UNIVERSITY OF THESSALY

DEPARTMENT OF COMPUTER & COMMUNICATION ENGINEERING

Design and implementation of a client-server system for acquiring beam intensity data from high energy accelerators at CERN

Thesis Advisor:
assistant professor
Christos D. ANTONOPOULOS

Author:
Athanasios TOPALOUDIS

Thesis Co-Advisor:
associate professor
Nikolaos BELLAS

CERN Supervisor:
Lars JENSEN

5th October 2012

Acknowledgements

I would like to thank my CERN supervisor Lars Jensen for the professional guidance and confidence he showcased in my skillset during my studentship at CERN.

I would also like to thank the project leader of the fast beam intensity measurements (FBCT) for the LHC, Dr. David Belohrad for his guidance and excellent cooperation on the FBCT project for the LHC.

Furthermore, I would like to thank Michael Ludwig for the time he spent talking with me about the existing implementation of the FBCT servers in the LHC – system A and B – along with other general matters that helped me evolve personally.

In addition, I would like to thank my supervisors of this Thesis Christos D. Antonopoulos and Nikolaos Bellas for taking over the difficult part of the remote supervision of this Thesis, their excellent cooperation and their suggestions and corrections on this work.

Last but not least, I would like to thank my family, Theodoros, Chrysi, Christos and Alexandros who helped me form a solid ground to evolve my personality and Eirini who showed me how to do it.

Table of Contents

Table of Figures

Table of Tables

Abstract

The world’s largest research center in the domain of High Energy Physics (HEP) is the European Organization for Nuclear Research (CERN) whose main goal is to accelerate particles through a sequence of accelerators – accelerator complex – and bring them into collision in order to study the fundamental elements of matter and the forces acting between them. For controlling the accelerator complex, CERN needs several diagnostic tools to provide information about the beam’s attributes and one such system is the Fast Beam Current Transformer (FBCT) measuring system that provides bunch-by-bunch and total beam intensity information.

The current hardware and firmware of the FBCT system has certain issues and lacks diagnostics as a lot of the calculations are done in an FPGA. In order to improve on this, the firmware was redesigned and simplified in order to increase its capabilities and provide the base of a unified FBCT measuring system that could be installed in several of CERN’s accelerator complex’s parts. Following the above changes, this Thesis proposes the implementation of an operational client-server software solution to control the FBCT installation in the Super Proton Synchrotron (SPS) accelerator, as well as studying the design and implementation of a unified client-server software scheme that can replace the operational ones in the Large Hadron Collider (LHC) and can eventually allow further installations of the FBCT measuring system, elsewhere in the CERN accelerator complex.

1 Introduction

The European Organization for Nuclear Research or CERN Laboratory is the largest research center in the domain of particle physics [1]. Its main activity is to accelerate ion or proton particles through its accelerator complex to their nominal energies and make them collide at one of the four collision points [2] in order to study the fundamental constituents of matter as well the forces acting between them.

The acceleration of the particles can only be achieved if the Radio Frequency (RF) field is correctly oriented with the accelerating cavity as they pass through it. Since this happens at specific moments of the RF cycle (sine-wave), particles travel around the accelerator complex in well-defined bunches [2].

For an accelerator's control to be effective, numerous diagnostic tools are needed to provide information about the beam's attributes [3]. Several measurement techniques exist providing such information and thus making the control of the CERN accelerator complex effectively feasible. One such technique uses AC-coupled Fast Beam Current Transformers (FBCTs) at first stage to integrate the current of each individual bunch inside a bunch-synchronized integration window and provide continuously 40MHz ADC values (in bits) [4], whereas, at second stage it implements data treatment in a Field-Programmable Gate Array (FPGA).

The FPGA firmware is used to store and/or reload at any time the device configuration in order to implement four acquisition modes, single *capture* – which measures the intensity for the specified bunch slots over a specified number of turns, *turn sum* – which measures the total intensity of all bunch slots available (depending on the accelerator) over one turn, *slot sum* – which measures the total intensity for a given bunch slot over a specified number of turns – and finally *sum of sums* – which measures a *turn sum* and then sums up these values using the *slot sum* measurement mode in order to produce one total intensity value [5].

In addition to the hardware part, there is also the software layer, responsible for controlling the device and to implement any data processing required that cannot be done by the firmware (floating-point calculations). Such data processing may be, averaging, data calibration – the transformation of the data from the measured values in number of bins to number of charges – and data publishing.

There is one FBCT system installed in the Super Proton Synchrotron (SPS) accelerator and two per beam in the Large Hadron Collider (LHC) that provide both bunch-by-bunch and total (per turn) beam intensity information. The FBCTs in the SPS ring are widely used at beam injection time to observe the beam losses at that critical part of its journey as well at specific times during the beam cycle to analyse the bunch-by-bunch losses. As for the LHC ring, only one – system A – out of the three FBCT installations is currently operational and is used by a large number of clients interested in bunch-by-bunch and turn-by-turn intensity information.

The original FBCT firmware was designed and developed by several people using different technologies. As a result, several design errors worsens the mean time between failures – MTBF – of the entire measurement system, complicating its maintenance. In order to properly develop the new FBCT system C, it was decided that a clean-up was necessary, moving a large part of the data treatment from the hardware to the software side. Therefore a new version of the firmware was designed and developed implementing only the *capture* acquisition mode leaving the software controlling the FBCT installations, responsible for all the data processing.

The whole idea behind this migration is to implement one data acquisition system – both hardware and software – that can be installed in the CERN accelerator complex and will be independent of the ring installed, which is not the current case, in order to make it generic and more easily maintainable.

As the new version of the firmware is already implemented, this Thesis describes the software solutions that need to accompany the hardware changes as well to propose new ideas as far as the data treatment is concerned. This document is divided in two large blocks: the first one introduces the theoretical and technical background whereas the second describes the proposed software implementation and outlines its performance evaluation.

In the first part, a brief introduction to the Organization and some fundamental knowledge concerning the FBCTs is given in chapter 2.1. In addition, in chapter 2.2 we describe the hardware architecture and in chapters 2.3 and 2.4, the existing software implementations for the FBCTs in the SPS and LHC accordingly.

In the second part, we provide our software design in chapter 3.1 and its technical implementation in chapter 3.2, along with the results of our proposals in chapter 4.

Finally, chapter 5 presents the conclusions of this work and directions for future work.

2 Background

In the previous section the need for the software design and implementation that controls the FBCT systems for SPS and LHC accelerators at CERN was discussed. In order to develop our suggestions, the analysis of some basic ideas related to the FBCTs, is needed.

Hence, we begin with the general information about CERN and other key aspects needed for the rest of this document and we continue with the hardware architecture where all the details relative to the hardware are given and finish this section with the description of the software implementations for the SPS and LHC rings that used to be or are operational.

2.1 General Background

In this section we analyse from scratch the basic information about CERN, its structure and accelerator complex because we are going to use this information for the deployment of our solution. Furthermore, we briefly describe the CERN Control Center and how the particles travel through the rings. Subsequently, we analyse the need of measuring the beam's attributes as well the different ways to do it. Lastly, we introduce the design framework that was used for the existing and the previous software implementations as well as ours.

2.1.1 CERN

The "Conseil Européen pour la Recherche Nucléaire" or "European Organization for Nuclear Research", well known as the "CERN Laboratory" is the largest scientific research center whose main area of research is high energy particle physics - the study of the fundamental constituents of matter and the forces acting between them.

It was founded in 1954 as one of Europe's first joint ventures and now it counts 20 member states. It is placed on the Franco-Swiss border near Geneva and it uses the world's largest and most complex scientific instruments in order to accelerate the particles, almost to the speed of light, before cause them to collide and study the fundamental laws of Nature [1].

Figure 2-1: CERN's Logo [6]

2.1.2 CERN's structure

The highest authority in the Organization is the CERN Council. It is formed by two representatives of each member state, one as the government's administration representative and one to represent the national scientific interests. Each member state has one single vote and in most of the cases a simple majority is needed for a decision to be taken.

The Council is responsible for all the important decisions that have to do with scientific, administrative and technical matters. It appoints the Director General who manages the CERN Laboratory through a structure of Departments which can be seen at fig.2-2. [7]

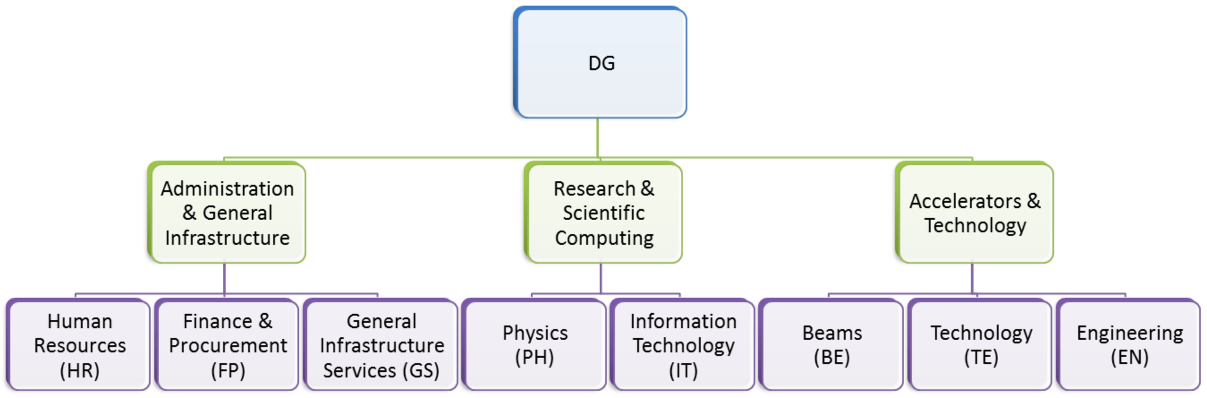

DG
Administration & General Infrastructure
Research & Scientific Computing
Accelerators & Technology
Human Resources (HR)
Finance & Procurement (FP)
General Infrastructure Services (GS)
Physics (PH)
Information Technology (IT)
Beams (BE)
Technology (TE)
Engineering (EN)

Figure 2-2: CERN's structure (source: Laura Saulnier, TECH induction 2012)

The author works in the Beams Department (BE) [8] and hence a little more emphasis will be given to it and its structure. The BE is responsible for everything that has to do with the production and acceleration of particle beams and their control while circulating through the CERN accelerator complex (see chapter: 2.1.3). The department is divided into six groups as can be seen in fig. 2-3.

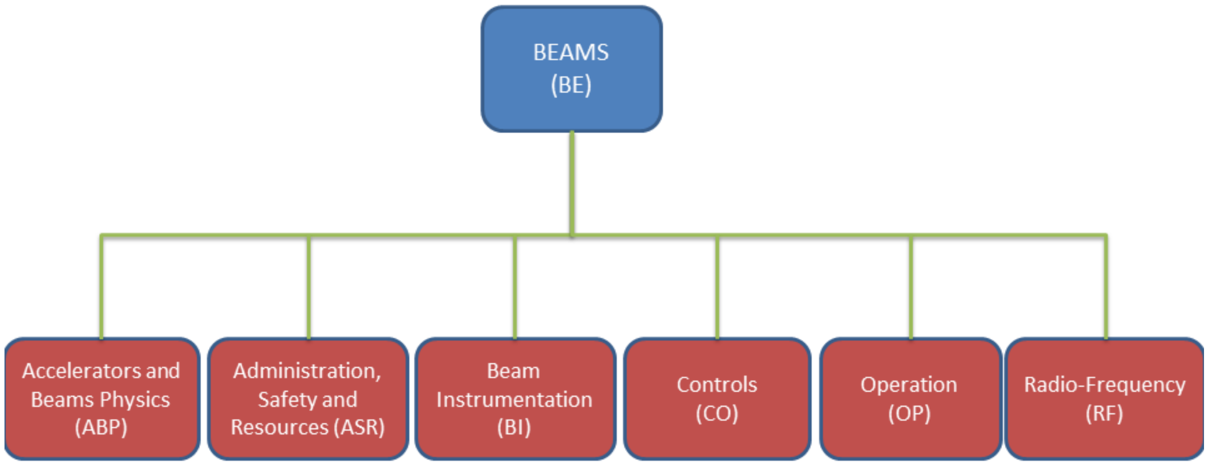

BEAMS (BE)
Accelerators and Beams Physics (ABP)
Administration, Safety and Resources (ASR)
Beam Instrumentation (BI)
Controls (CO)
Operation (OP)
Radio-Frequency (RF)

Figure 2-3: Beams Department's Structure

Each group is subdivided into sections. The focus of this Thesis is related to the Beam Instrumentation (BI) group which is responsible for studying, designing, building and maintaining all the instruments that allow the observation of the particle beams and its parameters which are important for its normal behaviour in the CERN accelerator complex.

[9] Its structure can be seen in fig. 2-4.The author works in the Software section (SW), responsible for providing the software needed for developing, testing, diagnosing, maintaining and controlling all the instruments provided by the group. [10]

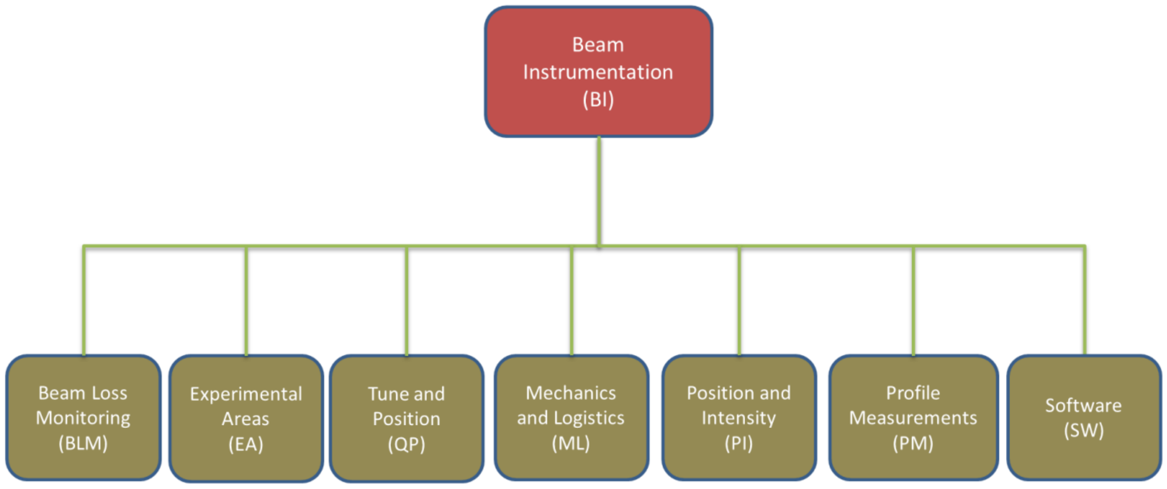

Beam Instrumentation (BI)
Beam Loss Monitoring (BLM)
Experimental Areas (EA)
Tune and Position (QP)
Mechanics and Logistics (ML)
Position and Intensity (PI)
Profile Measurements (PM)
Software (SW)

Figure 2-4: Beam Instrumentation's structure

2.1.3 The CERN accelerator complex

The accelerator complex at CERN is a succession of linear and circular accelerators through which the particles reach increasingly higher energies. Each accelerator receives the beam of particles from the previous part of the complex chain, boost its speed and energy and finally inject it to the next one in the sequence.

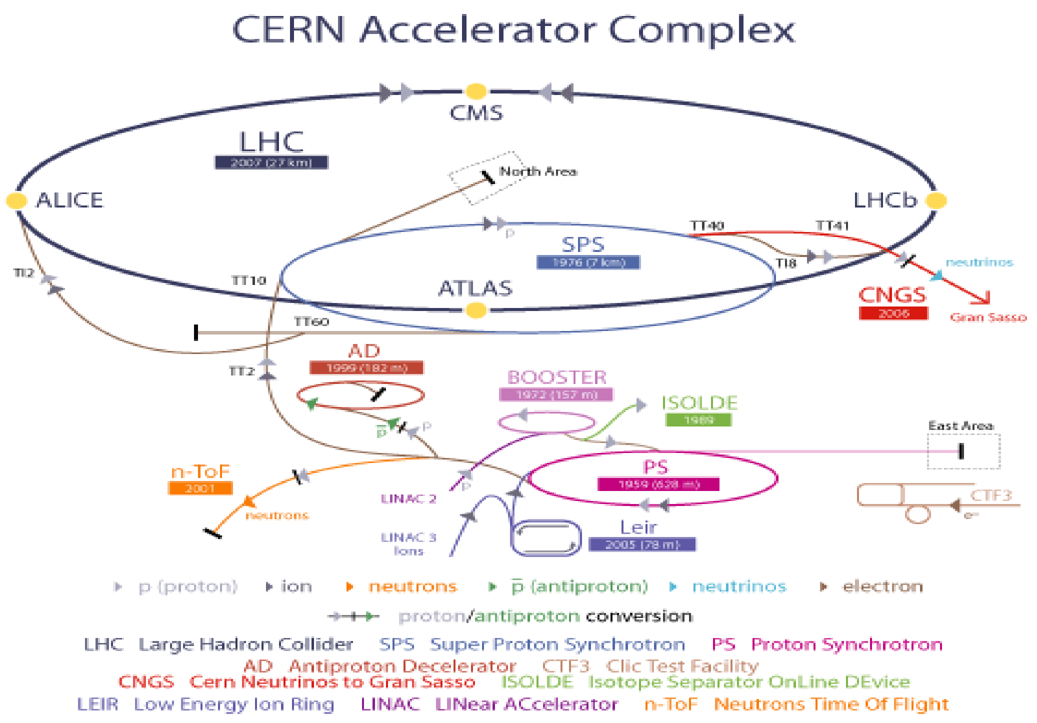


Figure 2-5: CERN Accelerator Complex [11]

There are two main types of particles that travel through the CERN accelerator complex: protons and ions.

The protons are obtained by stripping orbiting electrons from hydrogen atoms. They are accelerated in the linear accelerator (LINAC2) before they are injected into the PS Booster. They are then transferred to the Proton Synchrotron (PS) which is before Super Proton Synchrotron (SPS) in the complex sequence. Finally they are injected into the Large Hadron Collider (LHC) both in a clockwise and anticlockwise direction where they are accelerated to their top energy of 4TeV (nominally 7 TeV) before they start colliding with counter-rotating particles at one of the four collision points. [2]

The ions on the other hand, start from a source of vaporized lead and enter their own linear accelerator (LINAC3) before they are injected into the Low Energy Ion Ring (LEIR) from which they follow the same route as the protons to reach their maximum acceleration.

The complex also includes the Antiproton Decelerator (AD) which separates the antimatter particles while they are still in low energies, and the On-Line Isotope Mass Separator (ISOLDE) facility which is used as a unique source of low-energy beams of radioactive isotopes. The complex also feeds the CERN neutrinos to Grand Sasso (CNGS) project which creates and sends neutrino beams to Grand Sasso National Laboratory (LNGS) in Italy in order to detect the so called neutrino “oscillation”, the transformation from one type of neutrino to another. Last but not least is the Compact LInear Collider (CTF/CLIC) study, an international project working on a machine to collide electrons and positrons (anti-electrons). [11]

2.1.4 Control Center

The CERN Control Center (CCC) combines all the control rooms for the accelerator complex as well as the technical infrastructure under one roof. It consists of 39 operation stations organized in four different areas, the Large Hadron Collider, the Super Proton Synchrotron, the Proton Synchrotron complex and the technical infrastructure. [11]

2.1.5 Bunches

The particles travel around the CERN accelerator complex in well-defined bunches. That is because they can only be accelerated if the Radio Frequency (RF) field has a correct orientation when they pass through an accelerating cavity and that happens at well specified moments during the RF cycle. [2]

Under nominal operation, each LHC's proton beam can store up to almost 2800 bunches and SPS's 288, with each bunch containing about 10^{11} protons.

2.1.6 Beam Charge Measurements [3]

An effective accelerator control requires numerous types of diagnostic tools which provide information about the beam's attributes and they are commonly known as *beam diagnostics*. There are several measurement techniques which can be divided in two large categories, the intercepting and the non-intercepting measurements.

The first category, as it is revealed by its name, interacts with the beam in order to achieve the measurements and thus cause the destruction or deterioration of the beam, whereas the second group bases its measurements in the electric or magnetic field coupling of the beam to the measuring instrument.

The charge measurement, often called *beam intensity measurement*, is a process which integrates the actual measured quantity, the beam current, over a specific Region Of Interest (ROI) and divides that integral, the beam charge as it is called, by the elementary charge to result in the number of particle beam's charges.

The beam intensity measurement is vital for determining the intensity loss at injection, acceleration and extraction time or even the beam's lifetime while it is circulating in the accelerator. Furthermore, it enters the luminosity equation. [12]

What is important in this kind of measurements is the device that couples to the beam and provides the approximation of the beam's current. There are several different types of such devices. The most used of the intercepting DC devices are the Faraday cups. The non-intercepting AC devices are the electrostatic pickups, the Wall Current Monitors (WCMs) and the Fast Beam Current Transformers (FBCTs). The non-intercepting DC devices are the DC Current Transformers (DCCTs), the Superconducting QUantum Interference Devices (SQUIDs) and the Cryogenic Current Comparators (CCCs).

In this document we will focus only on the FBCTs, the devices that function in a bandwidth of few Hz up to GHz and on the contrary with all the other similar devices, can be absolutely calibrated. For more information see chapter 2.2 where the hardware is analysed in more detail.

2.1.7 FESA Framework

"The Front-End Software Architecture (FESA) is a comprehensive framework for designing, coding and maintaining LynxOS/Linux equipment-software that provides a stable functional abstraction of accelerator device." [13]

The Model of a FESA class is encoded as an XML Schema which enforces a specific grammar for the design of the class providing a partial yet generic solution for the equipment specialist. In this way and after the design of the class is well defined, the FESA user can generate a large part of the C++ code for his equipment saving a lot of time and effort. The FESA classes are identified by the combination of their name and version.

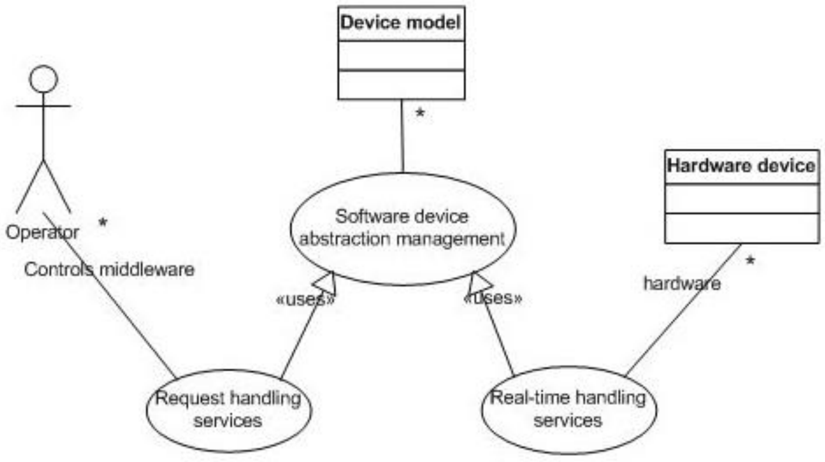

Device model
Operator
Controls middleware
Software device abstraction management
Hardware device
hardware
«uses»
«uses»
Request handling services
Real-time handling services
*
*
*

Figure 2-6: FESA's service supplies [13]

The Interface is a list of so-called Properties that defines the services that are available to the outside world and are remotely accessible by the clients of the FESA class, for example clients from the control room as well as middle-tier software layer. The Properties are attached to a server action (request) which can be of type GET or SET and either default, meaning that the code for that actions is auto generated, or complex for which the equipment specialist must provide the code himself.

The Data, the Device-Data and Global-Data, are defined in such a way to provide at any given time, a concrete snapshot of the device state. The data can be of any standard type that can be supported from both C++ and Java, scalars or arrays of up to two dimensions. There is also the possibility for the equipment specialist to define his own types, the persistency of the data or any multiplexing criterion for them.

C++ Scalar type	Array type
bool	bool
signed char (byte)	signed char
short	char
long	short
longlong	long
float	long long
double	float
	double

Table 2-1: FESA's data types [13]

Persistency	Purpose	Multiplexing	Purpose
FINAL	database constant	**NONE**	not multiplexed
PERSISTENT	periodic backup into persistent storage	**USER**	cycle user
VOLATILE	RAM data	**PARTICLE**	particle-type
		DESTINATION	beam-target

Table 2-2: FESA's data attributes [13]

The basic work-units of a FESA class are called actions and can be either of real-time or server type. The real-time actions are triggered by events which are synchronized with the CERN's central timing system or by hardware interrupts and they implement most of the equipment's functionality. They can also be attached to properties so that the latter can be notified at any update of the device's state. On the other hand, the server actions implement the client's request-handling and they are mostly responsible for the communication between the outside world and the device and that is exactly why they are attached most of the times with a property. For both real-time and server actions the equipment specialist must provide the C++ code himself, except for the default GET/SET server actions.

Once finished with the FESA design, one needs to declare all the instances his class would have. This is a very important part of the design procedure since a lot of work and duplicated code can be avoided. One instance means one module with its own initial values. All the instances (the modules the device can handle) are accessible inside the FESA class by iterating the *deviceCollection*, an array accessible everywhere in the class.

A FESA class, to which we will refer as 'server' from now on, is organized after its generation, in a directory structure containing: COMMON, GENERATED CODE, REALTIME, SERVER and TEST.

The REALTIME and the SERVER directory files are used to store and distinguish the actions based on their type as described above.

The GENERATED CODE directory files hold all the declaration of the fields that describe the device. Furthermore, all the generated code for the simple GET/SET actions is stored here.

The COMMON directory is used to store any custom made class that could be used by both real-time and server actions.

Last but not least is the TEST directory. In there, some diagnostic tests are stored as well as the executable files that allow testing the executable while developing.

2.2 Hardware Architecture

After giving the general information that is going to be needed in next sections, we are describing the hardware installation for the FBCTs. The latter consists of a detailed description of the ring installation as well as the one on the surface. Furthermore, we analyse the firmware – original and newer version – along with the driver needed to access it since they are widely used by the software and lots of the changes imposed to it derives from the changes of the firmware.

2.2.1 Fast Beam Current Transformer (FBCT) measurement system

Figure 2-7 depicts a simplified block schematic of the FBCT measurement system which consists of a Bergoz type transformer with a bandwidth from 400Hz to 1.2GHz (on the left). This transformer is followed by an RF front-end which consists of an analogue integrator, a Beam Circulating Flag (BCF) detector which detects the presence of the beam in the ring and an RF distributor which is responsible to split the analogue signal into two dynamic ranges, high and low gain and each dynamic range into two bandwidths, High (HBW) and Low (LBW). Finally a 14bit ADC digitizes and processes the signal. [14]

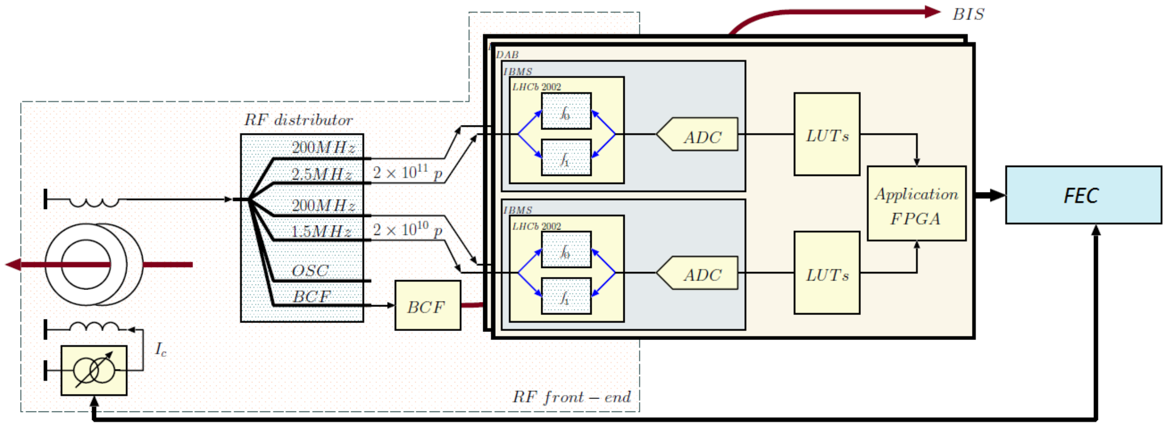


Figure 2-7: Block schematic of the FBCT measurement system [14]

The ADC module along with its analogue integration part is situated on a carrier called the Digital Acquisition Boards (DABs), a VME64x standard board developed by TRIUMF (Canada) for the LHC orbit and trajectory acquisition system [15]. It is equipped with two Individual Bunch Measurement System (IBMS) mezzanine cards [16]. Each mezzanine card uses a 40MHz integrator ASIC developed for the LHC-b preshower detector by the "Laboratoire de Physique Corpusculaire", UniversitéBlaise Pascal, Clermont-Ferrand [17], in order to integrate and sample the incoming signal before passing it to the DAB that processes it to provide bunch-by-bunch intensity values. All the logic of the DAB control is implemented in a large FPGA that can be reprogrammed at any time and its firmware is being discussed at chapter 2.2.2.

These DAB cards are installed on a VME64x crate along with the Beam Synchronous Timing Receiver Interface for the Beam Observation System (BOBR) – another VME format card that provides all the timing signals required to synchronize the different beam instrumentation systems [18]. What is more, all the cards installed in the VME64x crate are controlled by the Crate Central Processing Unit (CPU) – Front-End Computer (FEC) – an Intel® Core™ 2 Duo CPU board with 1.5GHz clock frequency, 4MB cache and no hard disc [19] that runs Scientific Linux CERN SLC release 5.7 (Boron) [20] and boots via network.

The following figure 2-8 depicts the VME64x crate installation for the SPS FBCT. The FEC is visible on the left of the crate with the green lights, whereas the DAB is just on the right of it and lastly, the BOBR in the middle of the crate.

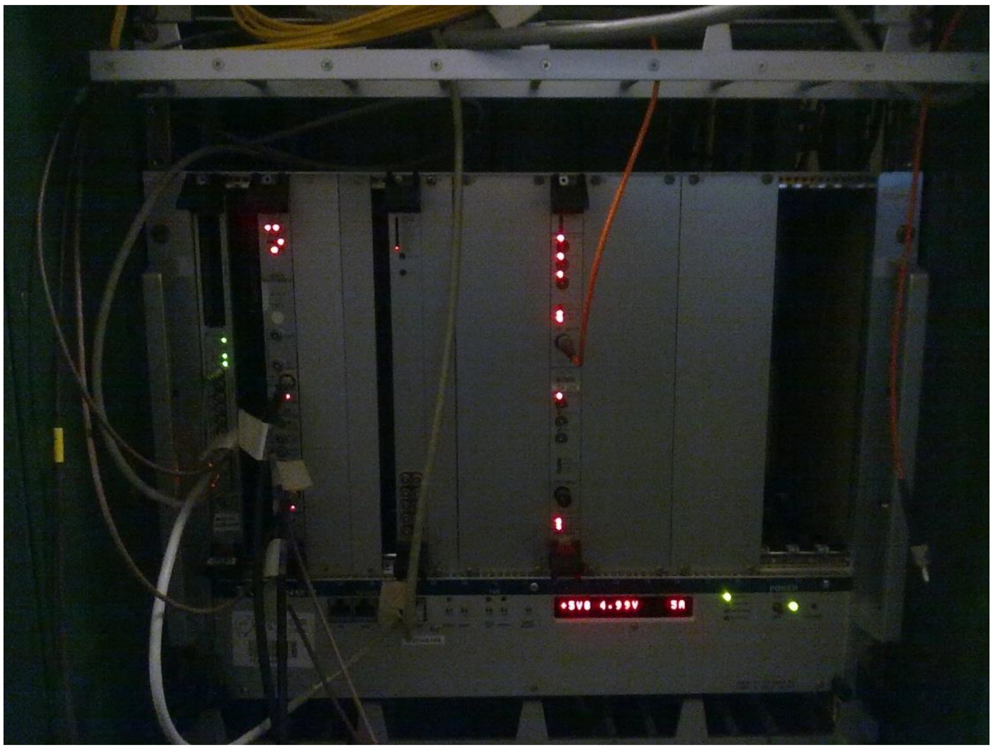


Figure 2-8: SPS FBCT's VME64x crate – on the left with green lights is the FEC, on the right of the FEC the DAB is installed and in the middle of the crate the BOBR is visible

As far as the SPS is concerned there is only a single DAB connected to the SPS type front-end amplifier. The former uses an external signal to switch between high and low gain measurements which is provided by the *sensitivity output* of each IBMS mezzanine.

In the LHC, things are different. There are two DABs per a measurement system, one for HBW and one for LBW measurements. Each DAB provides two dynamic range measurements using its different IBMS mezzanine and more specifically high gain (top mezzanine) and low gain (bottom mezzanine) measurements.

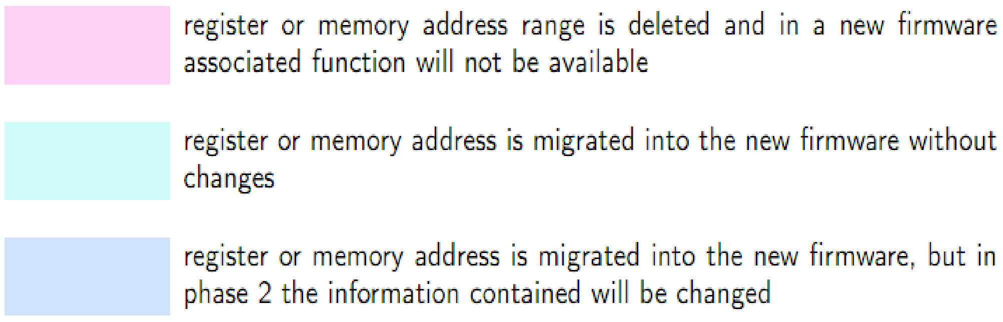
There are three such systems in the LHC, system A, B and C of which only A and B are operational while system C is now being developed with different technologies and with a different approach in the process of the data. Further discussion about this system will follow in chapter 3.

The FBCT measurement system is calibrated by a pulse of 25μs. The amplitude of this pulse differs from SPS – 128mA – and LHC which can be programmed. For the latter case though, the currently used system doesn't use direct calibration due to the fact that the LHC toroid exhibit beam position dependency and this can affect the transfer ratio between *beam – measurement turn* and *calibration turn – measurement turn*. Instead an indirect calibration is achieved by using DC current transformers (DCCTs) installed in the LHC [21].

2.2.2 Firmware

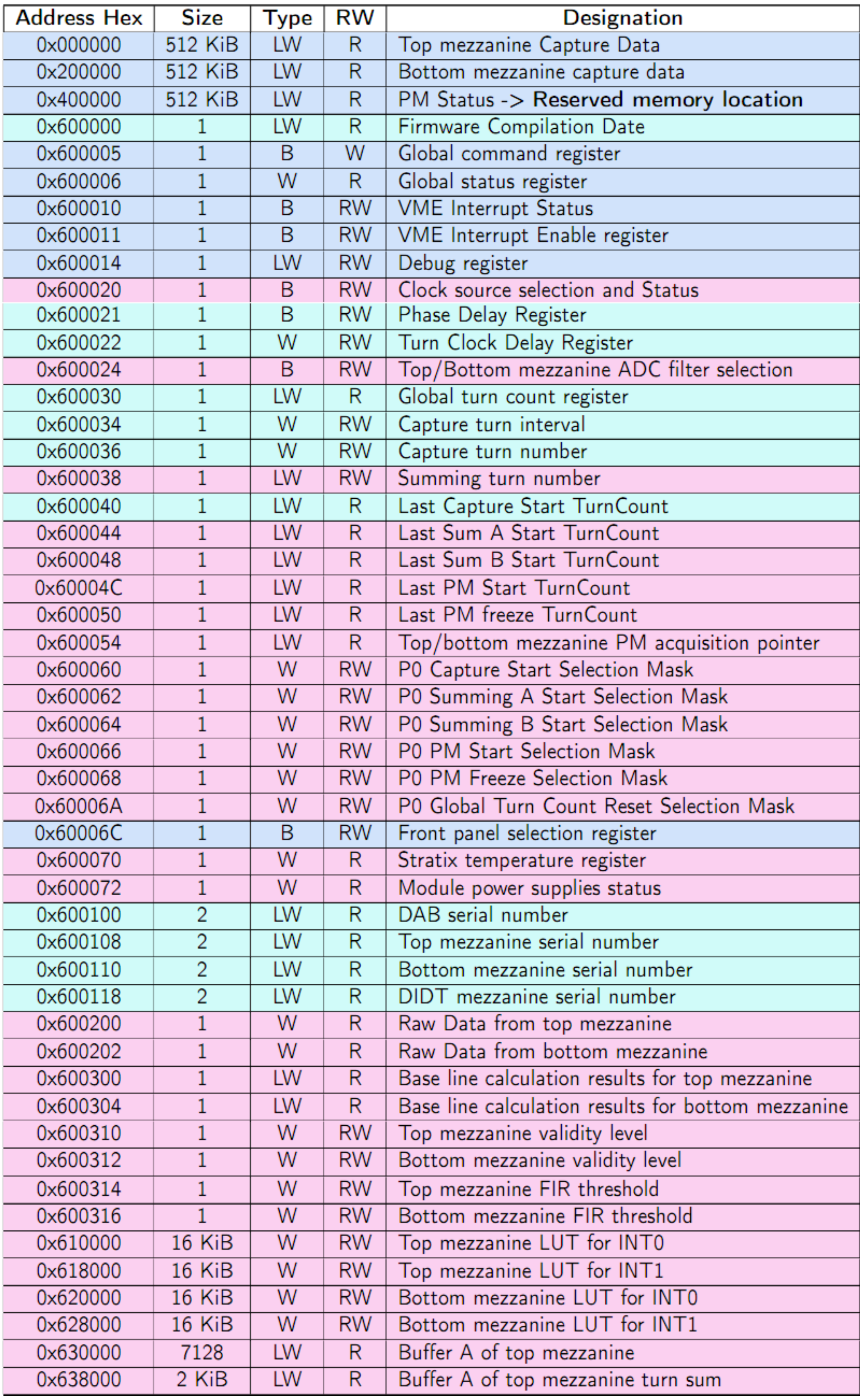
The Stratix-type FPGA stores the device configuration during operation at volatile SRAM cells, which must be reconfigured each time the device powers up. This is accomplished by its firmware (FIMDAB), which can be loaded over the VME bus as a Raw Binary Bile (RBF) via the EPM 3256 Complex Programmable Logic Device (CPLD). Software start-up scripts handle the FPGA start-up process and hence the FPGA is left unprogrammed after power up until the software layer is loaded. After the initial power-up process is complete, new configuration data can also be loaded at any time. [22]

As mentioned earlier for the new FBCTR system C, it was decided that a clean-up was necessary. The new firmware was also to be used for the FBCTR in SPS. The firmware registers migration is summarized in table 2-3, which uses the following colours to describe the state of the registers after the completion of the migration [23].



🟪	register or memory address range is deleted and in a new firmware associated function will not be available
🟩	register or memory address is migrated into the new firmware without changes
🟦	register or memory address is migrated into the new firmware, but in phase 2 the information contained will be changed

Figure 2-9: table's 2-3 Legend



Address Hex	Size	Type	RW	Designation
0x000000	512 KiB	LW	R	Top mezzanine Capture Data
0x200000	512 KiB	LW	R	Bottom mezzanine capture data
0x400000	512 KiB	LW	R	PM Status -> **Reserved memory location**
0x600000	1	LW	R	Firmware Compilation Date
0x600005	1	B	W	Global command register
0x600006	1	W	R	Global status register
0x600010	1	B	RW	VME Interrupt Status
0x600011	1	B	RW	VME Interrupt Enable register
0x600014	1	LW	RW	Debug register
0x600020	1	B	RW	Clock source selection and Status
0x600021	1	B	RW	Phase Delay Register
0x600022	1	W	RW	Turn Clock Delay Register
0x600024	1	B	RW	Top/Bottom mezzanine ADC filter selection
0x600030	1	LW	R	Global turn count register
0x600034	1	W	RW	Capture turn interval
0x600036	1	W	RW	Capture turn number
0x600038	1	LW	RW	Summing turn number
0x600040	1	LW	R	Last Capture Start TurnCount
0x600044	1	LW	R	Last Sum A Start TurnCount
0x600048	1	LW	R	Last Sum B Start TurnCount
0x60004C	1	LW	R	Last PM Start TurnCount
0x600050	1	LW	R	Last PM freeze TurnCount
0x600054	1	LW	R	Top/bottom mezzanine PM acquisition pointer
0x600060	1	W	RW	P0 Capture Start Selection Mask
0x600062	1	W	RW	P0 Summing A Start Selection Mask
0x600064	1	W	RW	P0 Summing B Start Selection Mask
0x600066	1	W	RW	P0 PM Start Selection Mask
0x600068	1	W	RW	P0 PM Freeze Selection Mask
0x60006A	1	W	RW	P0 Global Turn Count Reset Selection Mask
0x60006C	1	B	RW	Front panel selection register
0x600070	1	W	R	Stratix temperature register
0x600072	1	W	R	Module power supplies status
0x600100	2	LW	R	DAB serial number
0x600108	2	LW	R	Top mezzanine serial number
0x600110	2	LW	R	Bottom mezzanine serial number
0x600118	2	LW	R	DIDT mezzanine serial number
0x600200	1	W	R	Raw Data from top mezzanine
0x600202	1	W	R	Raw Data from bottom mezzanine
0x600300	1	LW	R	Base line calculation results for top mezzanine
0x600304	1	LW	R	Base line calculation results for bottom mezzanine
0x600310	1	W	RW	Top mezzanine validity level
0x600312	1	W	RW	Bottom mezzanine validity level
0x600314	1	W	RW	Top mezzanine FIR threshold
0x600316	1	W	RW	Bottom mezzanine FIR threshold
0x610000	16 KiB	W	RW	Top mezzanine LUT for INT0
0x618000	16 KiB	W	RW	Top mezzanine LUT for INT1
0x620000	16 KiB	W	RW	Bottom mezzanine LUT for INT0
0x628000	16 KiB	W	RW	Bottom mezzanine LUT for INT1
0x630000	7128	LW	R	Buffer A of top mezzanine
0x638000	2 KiB	LW	R	Buffer A of top mezzanine turn sum

Table 2-3: Original firmware register map [23]

Address Hex	Size	Type	RW	Designation
0x63C000	4	LW	R	Sum of Sum buffer A of top mezzanine
0x640000	7128	LW	R	Buffer A of bottom mezzanine
0x648000	2 KiB	LW	R	Buffer A of bottom mezzanine
0x64C000	4	LW	R	Sum of Sum buffer A of bottom mezzanine
0x650000	7128	LW	R	Buffer B of top mezzanine
0x658000	2 KiB	LW	R	Buffer B of top mezzanine
0x65C000	4	LW	R	Sum of Sum buffer B of top mezzanine
0x660000	7128	LW	R	Buffer B of bottom mezzanine
0x668000	2 KiB	LW	R	Buffer B of bottom mezzanine
0x66C000	4	LW	R	Sum of Sum buffer B of bottom mezzanine
0x670000	2 KiB	LW	R	Turn summing of top mezzanine channel data
0x672000	2 KiB	LW	R	Turn summing of bottom mezzanine channel data
0x674000	1	LW	R	Firmware revision
0x680000	1	LW	RW	BCF threshold register
0x680004	1	LW	R	BCF INT0 min/max register
0x680008	1	LW	R	BCF INT1 min/max register
0x68000C	1	LW	R	Extended status register
0x680010	1	LW	R	BCF difference register
0x680014	1	LW	R	Turn Clock Watchdog

Table 2-3: Continue from previous page

Following the table 2-3, table 2-4 summarize the minimum set of registers for the new proposed memory map. The table is organized in three categories. First group consists of the registers read directly from the DAB external static memories. Second group contains all the registers that are not specific to capture mode and third group contains registers only specific to capture mode. The latter two are separated by an unused address space, which makes a potential insertion of new registers simpler. All registers are 4-byte aligned and accessed by A32D32 transfer. Lastly, for non-single transfer registers, block transfer can be used improving the latency added when transferring huge amount of data. [23]

Address Hex	Size	Type	RW	Name	Designation
External sampled data memories					
0x000000	512 KiB	LW	R	CTopData	Top mezzanine Capture Data
0x200000	512 KiB	LW	R	CBottomData	Bottom mezzanine capture data
0x400000	512 KiB	LW	R	ExtendedData	PM Status -> **Reserved memory location**
Global registers					
0x600000	1	LW	R	FWDate	Firmware Compilation Date
0x600004	1	LW	R	FWRevision	Firmware revision
0x600008	1	LW	R	TClkWd	Turn Clock Watchdog
0x60000c	2	LW	R	SNDAB	DAB serial number
0x600014	2	LW	R	SNTop	Top mezzanine serial number
0x60001c	2	LW	R	SNBottom	Bottom mezzanine serial number
0x600024	2	LW	R	SNPIM	PIM mezzanine serial number
0x60002c	1	LW	RW	Debug	Debug register
0x600030	1	LW	RW	FPMux	Front panel selection register

Table 2-4: New firmware register map [23]

Address Hex	Size	Type	RW	Name	Designation
0x600034	1	LW	RW	Command	Extended/Global command/status register
0x600038	1	LW	RW	IRQ	VME Interrupt enable/status register
0x60003c	1	LW	R	GlobalTurnCount	Global turn count register
0x600040	1	LW	RW	TClkDelay	Turn Clock Delay Register
0x600044	4	LW	R	FWCodename	Firmware codename ASCII string
0x600054	1	LW	R	PSStatus	Power supplies status
0x600100	128	LW	R	DebugMem	General purpose debugging memory
Capture related registers					
0x610000	1	LW	RW	ClkPhase	Phase Delay Register
0x610004	1	LW	RW	CTurnInterval	Capture turn interval
0x610008	1	LW	RW	CTurnNumber	Capture turn number
0x61000c	1	LW	R	CStartingTurn	Last Capture Start TurnCount
0x610200	128	LW	RW	CBunchSelector	Bunch selection memory
0x610400	1	LW	R	CRollPointer	Rolling pointer
0x610404	1	LW	R	CCurrentBSlots	Number of currently selected bunch slots

Table 2-4: Continue from previous page

From the latter table, 6 major changes at the registers can be pointed out.

Firstly is the capture data organization. Using 32-bit storage, two 14-bit ADC samples can be stored per entry. Unfortunately this is not enough since additional information is needed to be stored with the stream, information about what integrator was used for acquiring the sample – the most significant bit of the sample (31 and 15) reveals the appropriate integrator (0 or 1) –, about whether the sample was saturated – bits 30 and 14 – and finally, about where the turn clock starts. Since there is no space left to store the latter information with the stream, a convention had to be declared: the turn always starts at the memory start address – 0x000000 for top mezzanine and 0x200000 for bottom. Hence, next turn can be easily calculated as following: <start_address> + (<number_of_bunches> / 2).

Such memory organization decreases the amount of external memories read from three to two, since the information stored in mezzanine three are now coded with the samples. It also increases the number of samples per mezzanine by factor of two, enabling at the same time the use of fast block transfer of the data, from the external memories to the CPU.

Furthermore, changes in register bit positions should also borne in mind. The original information of the *Turn Clock Delay* register is migrated from address 0x600022 to 0x600040, bits 12…0, whereas the information of the *Phase Delay* register from address 0x600021 to 0x610000, bits 7…0. As for the *Front Panel Selection* register, information about MUXA originally located at bits 7…4 is extended into bits 31…16, whereas information about MUXB, originally at bits location 3…0 is extended into bits 15…0. As far as the *IRQ* register is concerned, it behaves as *Interrupt Enable* register when written and returns the *Interrupt Status* register when read.

Last but not least is the *Command* register which combines the original locations at 0x600005, 0x600006 and 0x68000c and acts as *Command* register when written, keeping all the original properties and as *Status* register when read. The meaning of all bits read is changed though, due to the differences between the two versions of the firmware and for a full description of this meaning refer to full technical documentation. [22]

2.2.3 Driver Background

There are more than one ways to access the device's register and hence, we had to find which one is more suitable for us. The most common way is to use the *ioctl* module-specific library that comes with the driver and is automatically generated from the description of the module in the CO Data Base. This is a simple library that uses only one method to access the hardware, IOCTL. This library is good for individual values or short amount of data, since it is already high level and fast enough for single accesses.

If the performance is one of the main characteristics of the project, one should consider another library that comes with the same auto generated driver and that is *dal* (Driver Access Library). The *dal* library has three ways of accessing the hardware and these are IOCTL, same as before, IOMMAP and IODMA. Now as for the last two, the IOMMAP method uses the CPU to access the hardware while the IODMA does this directly.

We have been experimenting with these three ways, only to find out that there is a significant difference between IODMA and the other two. Generally we could summarize our conclusions as this: faster: IODMA < IOMMAP < IOCTL. As we saw in chapter 2.2.2, only three of our registers contain a considerable amount of data (512.00 KiB) and from those, only two are being currently used. All the others are either single valued or short amount of arrays. Thus we've decided to use the *ioctl* library for all the registers except the two capture memories for which we've used the *dal* library with the IODMA method.

2.3 The FBCTs in the SPS

As described in the previous sections there is only one FBCT system installed in the SPS and this consists of only one DAB card on the VME crate, which is used to operate with the original version of the device's firmware (FIMDAB).

In the following sections, we will describe how the server used to be organized and which were its basic functionalities that made it operational.

2.3.1 Software Architecture

The server was designed[1] to operate a full acquisition (1-924 bunches) for every different active cycle (a typical cycle's length is 20sec). A sequence of real-time actions were used to accomplish that by preparing the device, starting the acquisition, reading back the acquired data, processing them, storing them temporarily, starting the acquisition again and repeating this sequence until the cycle was over.

All these functionalities were implemented in different real-time actions, rtPrepare, rtStart, endCapture and rtStop whose technical specifications will be discussed in the following chapter 2.3.2. The scheduling of these actions was the key for the proper operation of the server.

A warning of the beam's injection was used as an event that comes 20 msec before every different cycle's injection. This event was being used by the rtPrepare to set the appropriate settings to the device as well as calibrate it, before the acquisition could start.

Another event, specifying the beam's injection – cycle's beginning, was being used by the rtStart to initially start the acquisition. After that, an event coming every 40msec, was

[1] The server was first created by Lars Jensen

being used by the endCapture to read back the acquired data, process and store them in temporary buffers and finally start the acquisition again. This procedure was being repeated as many times as it could fit in every cycle’s lifetime.

Lastly, an event specifying the cycle’s end was being used by the rtStop to gather all data from the temporary buffers and store them in the shared memory of the server so that it could be fetched to the users.

The FESA properties that are used to interface the server were Setting – where the user could enter the settings relative to the acquisition, Expert Setting – where the user could specify the settings relative to the calibration of the device, Acquisition – where the user could see the desired data after all steps of their process, User Data – where the user could see the intermediate steps of the processed data and Calib Data – where the user could see and set the calibration factors of the data, either on his own or with respect to the calculated ones by rtPrepare. The generic FESA Navigator GUI was used for controlling and visualizing the above properties and an example showing acquisition data can be seen in figure 2-10.

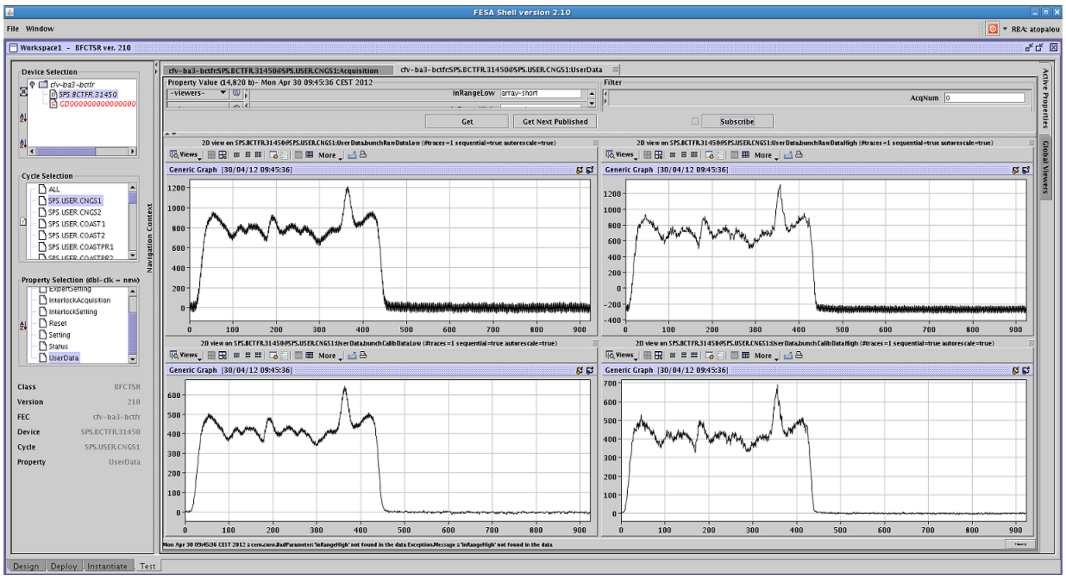


Figure 2-10: FESA Framework Interface

2.3.2 Previous Implementation

The previous implementation of the server used to access the device directly from its classes using the IOCTL library.

More specifically the rtPrepare action used to set the full bunch range (bunches 1-924) to the device as well as the number of turns for the acquisition which was always 1. After that, it would start an acquisition along with a calibration pulse in order to calibrate the device. This is achievable due to the fact that the rtPrepare operates when there is no beam. In this way and by firing a calibration pulse, whose current is known in advance, the appropriate calibration factors could be specified to take away all the additional noise that is being added to the data by the electronic equipment. Following the calibration, the rtPrepare would reset all the intermediate temporary buffers that were going to be used by the endCapture.

For the rtStart action, things used to be much simpler, since its only responsibility was to start a normal acquisition which means without the calibration pulse.

Furthermore, the endCapture action was the most critical one as far as the time constrains is concerned. In this action, the data would be fetched from the device and be processed before been stored to the temporary buffers. By processing the data, we mean to restore their base line as well as apply the calibration factors that were calculated before by the rtPrepare. The base line restoration is by far the most difficult stage of their process since its main goal is to take away the beam's AC dependency with the measuring device, restoring the level of the acquired noise to 0 in the y (intensity) axis, and this procedure is non-trivial at all when taking into account all beam types.

The existing implementation was using the Magic Imperial algorithm to restore the data's base line. This algorithm was based on the statistics from previous operational experience and its basic idea was the following:

- Iterate the acquired data and find minimum and maximum value.
- Using this information, determine the noise region as the (minimum value + (0.05 * maximum value)).
- Iterate again the acquired data and find a mean value for any sample that is below the just specified threshold.
- Finally iterate all the acquired data and take away this just calculated mean value.

In this way, all the noise samples would reach the 0 area in the y axis, while the original shape of the data would stay unchanged.

Last but not least, the rtStop action stored the intermediate buffers to the shared memory (device fields). This was accomplished by declaring the above buffers with the C++ key word *extern* and hence they were visible by more than one C++ class in the server.

The server actions that served the Setting and Expert Setting interfaces were implemented as simple actions. What is more and only for the Setting property, partial setting was allowed. As for the Calib Data property complex GET/SET actions were implemented with the partial setting enabled. Lastly, for the Acquisition and User Data, complex GET actions copied the contexts of the shared memory (fields) to the interface memory in order to be properly presented.

At this point, it's worth mentioning few words about the buffers holding the data, intermediate and final. The acquisition data were stored in two dimensional arrays; first dimension for the different measurements made by the endCapture and second dimension for the acquisition itself –intensity values for bunch slots 1-924. Unfortunately, there was no useful way to present these values with FESA interface and thus filters were being used. Hence, under User Data property, the user had to specify in the filter which measurement desired to observe. Using this filter in the server action, only one row of the 2D arrays was returned (924 values in total). In this way, data were quite uncomfortable to be studied, since the filters apply in the acquired data only once and thus one should wait for the next acquisition to see another measurement. One such example can be seen at Figure 2-10.

2.4 The FBCTs in the LHC

In the LHC ring there are three FBCT systems, each consisting of 4 DABs as described in chapter 2.2.1. System A and B use the original version of the FBCT's firmware which used to have 4 measurement modes [5]:

- **Capture** – the intensity measurement in each bunch slot for a specified number of turns
- **Turn Sum** – a total intensity measured from a full bunch acquisition (3564 bunches) over a single turn
- **Slot Sum** – a total intensity measured for a specified bunch slot over specified number of turns
- **Sum Sum** – the combination of *Turn Sum* and *Slot Sum*. By this we mean to make a *Turn Sum* for each acquired turn and then, sum all these sums as they were a single bunch slot measurement

2.4.1 Software Architecture

The FESA class that serves LHC's A and B FBCT measurement systems is BCTFRLHC v31[2]. The server of both systems is identical and has two instances, serving the FBCT installation for each circulating beam.

The version 31 of the BCTFRLHC FESA class is designed to provide LBW total intensities averaged over 225 consecutive turns at 1Hz. In addition, it provides HBW total intensities per turn with time resolution up to one turn (89μs) as well as HBW individual bunch intensities averaged over 900 turns as input for the post-mortem system for analysing the causes of machine protection beam dumps. [4]

2.4.2 Existing Implementation

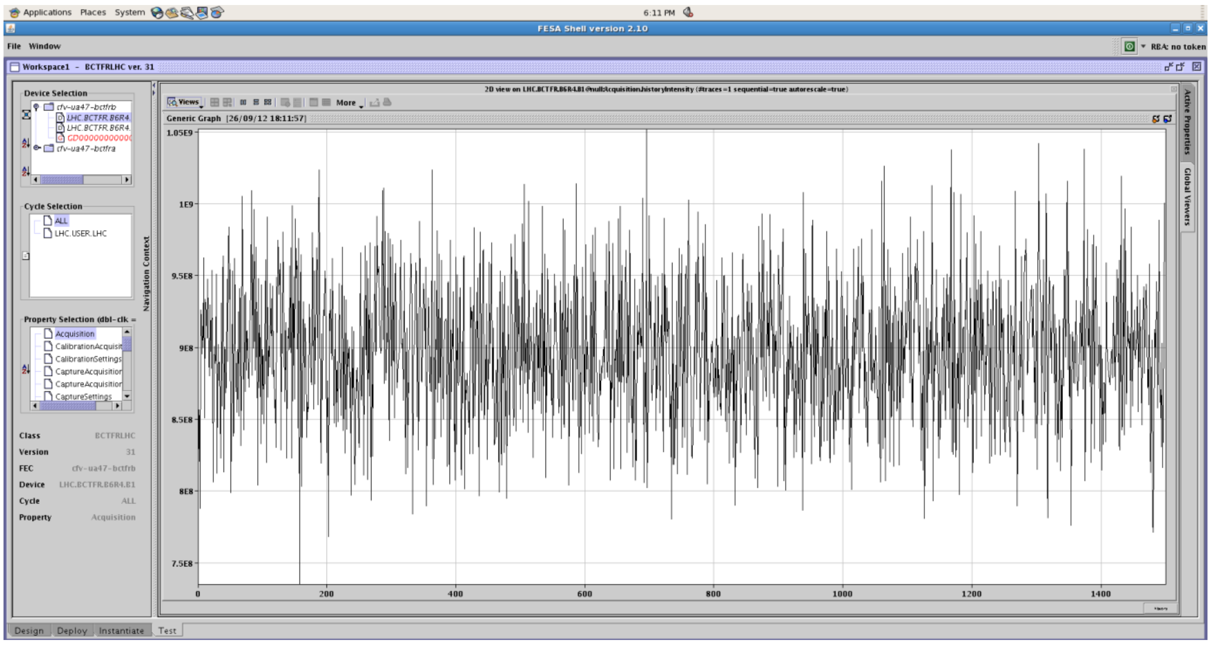
The server uses the LBW channel to make full bunch acquisitions over 225 consecutive turns – to suppress the noise at 50 Hz – using firmware's *Sum Sum* measurement method and it continuously updates them every second for operational displays. Additionally, it keeps the values from the last 30 seconds in a rolling data buffer, which also updates every second.

As for the HBW channel, the server uses the firmware's *Turn Sum* measurement mode to produce and publish the turn intensities – the total intensities per turn – and the *Slot Sum* measurement for the average individual bunch intensities. Both measurements are updated every second.

In order to suppress errors in the calculation of the noise mean value at the baseline restoration (BLR) procedure, the summing of empty buckets must be avoided. This is achieved by applying a minimum beam threshold set by the user. The BLR is based on the presence of empty buckets in each turn at least at the 3μs abort gap and hence, the calculation of the minimum integrated value of one turn can be used as offset correction for the next one. Subsequently, the lowest measurable turn-sum and bunch-average intensity is given by the

[2] Created by Michael Ludwig

noise suppression peak threshold – 10^8 number of charges for high gain and $5*10^8$ number of charges for low gain for both bandwidths. [4]



2D view on LHC.BCTFR.B6R4.B1@null:Acquisition.historyIntensity (#traces=1 sequential=true autorescale=true)

Generic Graph [26/09/12 18:11:57]

Class BCTFRLHC
Version 31
FEC cfv-ua47-bctfrb
Device LHC.BCTFR.B6R4.B1
Cycle ALL
Property Acquisition

Figure 2-11: Total Intensity History from beam1 of the LHC, System B

The above figure 2-11 depicts the rolling data buffer of the total intensity of beam 1 as it was measured by the FBCT in the system B. This buffer holds the calculated total intensities of the last 30 seconds – 1 acquisition over 225 turns takes 20ms hence 50 values per second and 1500 per 30 seconds. Up until recently no expert GUI was developed, so the client application that was being used to control the servers was the FESA Navigator interface.

3 New Implementation

In the previous section we described all the theoretical and technical background needed to better understand the previous software implementation for the FBCTs in the SPS and the existing one for the LHC. In this section we analyse our proposal for both systems separating the design from the technical part.

3.1 High level diagnostics

As the development of the two systems was ongoing, several decisions had to be made in order to proceed. This chapter is dedicated to such decisions that helped structuring the work and providing useful tools for our implementations.

3.1.1 Wrapper - Design

Since the firmware changed, a new way of accessing the device was needed. As the new firmware was to be deployed in both SPS and LHC FBCTs, we decided to create a common wrapper class, DABBFCTSRWrapper, which abstracts the device communication with the server. Additionally, such class is ideal for implementing functionalities irrelevant to the accelerator that hosts the FBCTs.

The DABBFCTSRWrapper is designed to have public methods for accessing all the device's registers using the IOCTL library, as well as processing some of the data that need to be read from or written to it, while there are also some other private methods for that scope as well.

Finally the header file of the wrapper seemed the perfect place for implementing the hash table with the different commands that the device can handle since it is imported every time we want to use it in the project for accessing the hardware and hence to instruct it to do something. In this way we've implemented it once being sure that is always visible in our general implementation.

3.1.2 Tester - Design

Another decision that was taken in the early days of our implementation was to create an additional tester class for testing the proper communication with the device. This class tries to write all the writable registers of the device and then read them back. In this way, several errors in the firmware were revealed early on and fixed.

While progressing with the implementation, the tester was changed to fit our testing needs. Hence, the tester ended up asking the user to select the bunches and the number of turns for acquisition, then firing the acquisition, reading back the data and printing them in the console as raw ADC values, just as they were read from the device. This procedure was found incredibly useful for studying, testing and assuring the decoding process of the data (look at chapter 2.2.2 – last paragraph / change of the data capture organization).

Furthermore, additional timing routines were added in order to study the different driver solutions for fetching the data from the device to the CPU, as well as some performance issues, especially as the server in LHC is concerned. These issues are being discussed in greater detail in chapter 3.2.5.

3.1.3 DabInfo - Design

As described in chapter 2.2.2 and table 2-4, there are some registers in firmware related to the DAB's information such as serial numbers and so on. Hence, it was found useful to have a console application that would retrieve and present this information. In this way, we were able to check the identification of the firmware, the mezzanines as well as the DABs themselves installed in the SPS, the LHC or the lab.

3.1.4 SPS

Our implementation is based on the existing one. We used this version and updated it so that it can access the new hardware and have one different acquisition mode the TURN_BY_TURN as we called it, as well as improving some troublesome behaviour relative to base line restoration. Our main goal, beside the proper functionality of the server of course, was to keep as much backwards compatibility as we could by changing the design as less as possible.

Hence, a new real time action was introduced; the rtTurnAcq which implements the new acquisition mode, while the rtPrepare remained the same, at least as far as the design is concerned.

The main difference to the existing classes was at the rtStart and endCapture class which were not needed if the acquisition mode was TURN_BY_TURN, and thus should exit immediately. The same idea was introduced to the new rtTurnAcq class but the other way round, it would return if the acquisition mode was REPETITVE. The event that wakes the rtTurnAcq is a warning of the beam's injection which come 20msec in advance. The new class is responsible to start the acquisition with 18msec delay, read the data, process them and transform them from ADC bins to number of charges, restore the DC baseline and finally store them in the appropriate buffers.

We kept the rtStop class the same which only copies the data from the buffers to the shared memory when the cycle is over. This is common for both acquisition modes and so, it made sense to try and keep it the same. In order to do that though, we had to change the buffers visibility through the server classes. In that sense, the variables that should be common to both acquisition modes and thus the appropriate classes, are now being created and initialized in the rtPrepare class and are visible by the endCapture, rtTurnAcq and rtStop by using again the keyword *extern*.

3.1.4.1 Baseline Restoration

The existing algorithm that used to correct the baseline was working quite well but unfortunately not always. It was observed that whenever there was a negative spike in the integrated signal, the algorithm didn't work. Since the algorithm was taking into account the ratio between the minimum and maximum value within an acquisition to determine the noise region, in case of this so called "undershoot" this region would typically contain a single measurement point, the minimum. As a result the minimum would be considered as noise and thus, after the BLR it would end up to be 0 and everything, including the actual noise, to be in the positive side of the graph. This can be easily seen at the following graphs:

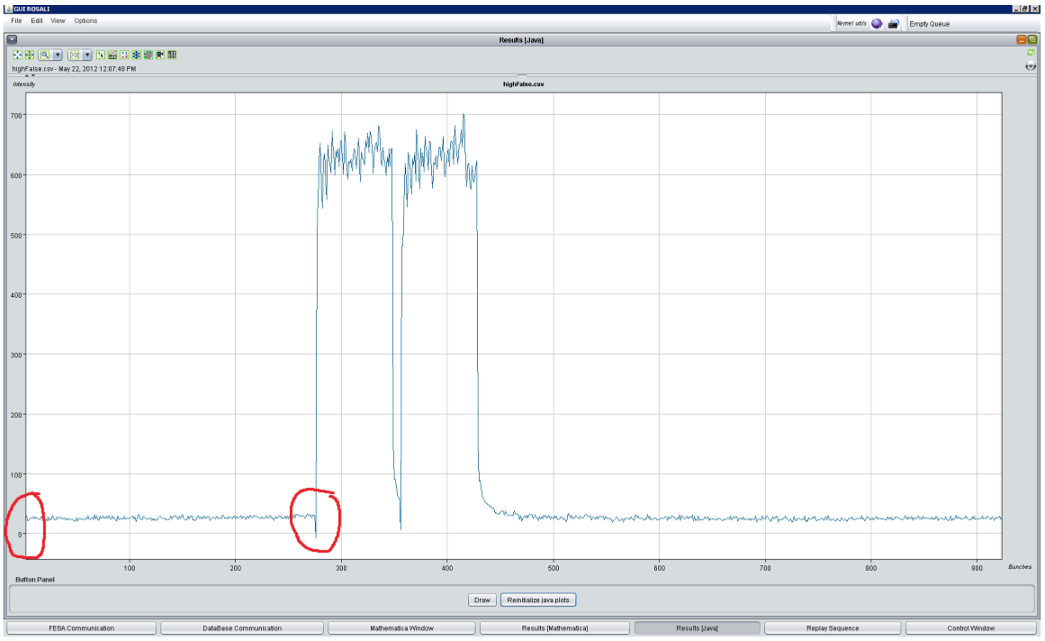


Figure 3-1: Bunch-by-bunch intensity measurement plot with error in BLR from SPS FBCT

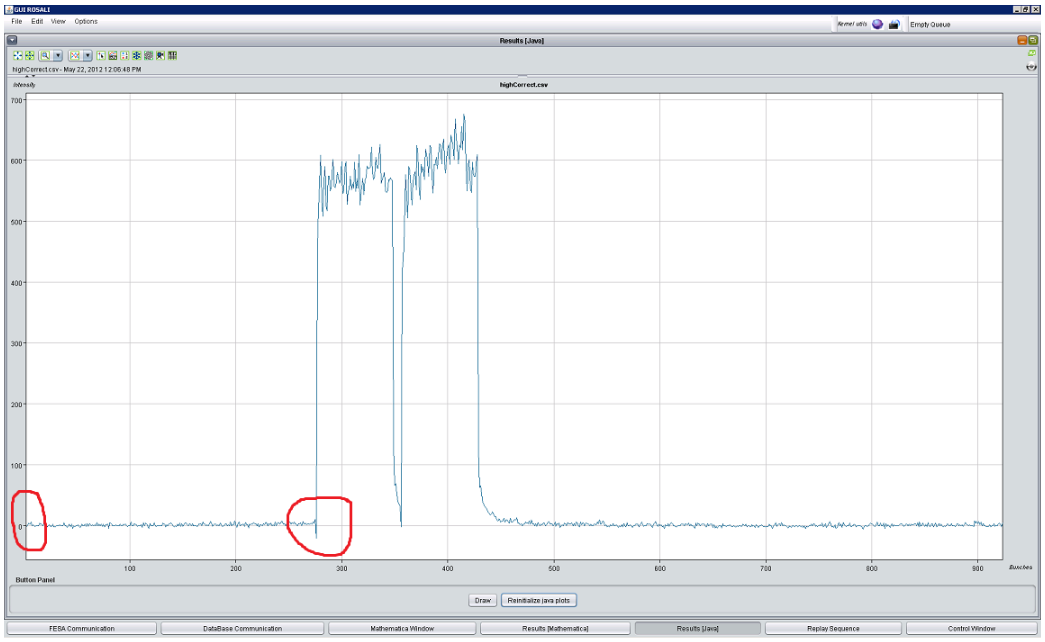


Figure 3-2: Bunch-by-bunch intensity measurement plot with correct BLR from SPS FBCT

These “undershoots” arrive infrequently, but when they come the BLR doesn’t work as it should be. We therefore considered changing the existing algorithm for restoring the BLR to another one much simpler and more stable.

We’ve decided not to take into account the min – max difference to specify the noise region, since this can change from cycle to cycle and from time to time. The hard coding percentage of that difference wasn’t flexible enough when those differences appeared. Hence, we search only for the minimum value of an acquisition and noise area is determined by a user setting. In this way, the BLR is much more flexible and dynamic.

Of course this does not erase the “undershoot” problem, since they don’t come in a deterministic way and thus one cannot specify a well-defined noise area and trust that would work for a longer period of time. In addition, an “undershoot” identifier had to be designed in order to help us ignore this kind of extreme values. To do that though, the user should provide another setting specifying the distance between two consecutive points that would identify the most negative as an “undershoot”.

3.1.4.2 TURN_BY_TURN acquisition mode

The most important change to the server was to add the new acquisition mode. As it was mentioned before, a full bunch acquisition (bunches: 1-924) over one turn, is repeated every 40msec until the end of every cycle. This mode of acquisition, REPETITIVE, covers the whole cycle and it was being used until now.

The new acquisition mode, TURN_BY_TURN, is again a full bunch acquisition but for as many consecutive turns as the data storage permits. This limitation comes from our effort to keep the backwards compatibility and hence by the fact that we use the same intermediate buffers in software as the REPETITIVE mode. For more details about the implementation of these buffers and their limitations please refer to chapter 3.2.4.

3.1.4.3 Client – Interface

The BFCTSR_ExpertGUI was developed in Java, organized in 5 packets for clearer separation of its classes. The *Constants* packet hosts all the classes that consist of constant data such as enumerations, names and converters. In the *expertGUI* packet, all the classes that implement the application interface are stored. Furthermore, there are the *factories* and *listeners* packets which host the homonyms classes. Last but not least is the *Data* packet where all the classes that are data specific are stored.

For the communication with the server, we used the *communication* library that was developed by BI/SW section and establishes a communication flow per device. We kept the communication and subscription mechanism over the network separated to one class called *DataProvider* and the data storage per FESA property to another called *Property*. Both classes are abstract since only few methods are domain specific and had to be separated.

The general idea of the design is the following: the *DataProvider* communicates via subscription to the server that runs on the front-end. Each time new data are produced, the *DataProvider* informs the *Properties* which process them if needed and store them to buffers. Then, they inform their interfaces to update their view with the new data. This data flow is depicted in the following figure:

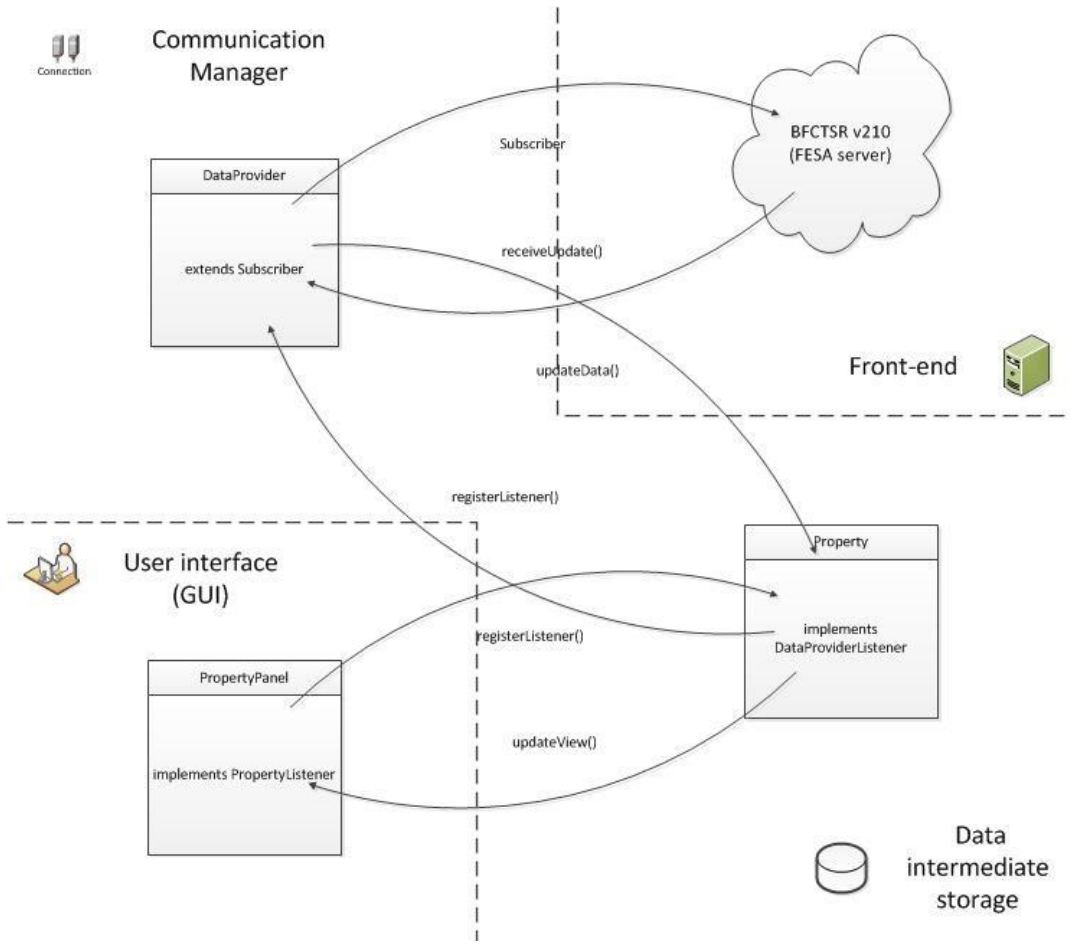


Figure 3-3: Data flow between front-end server and GUI client

We decided to split the frame into three areas. The top one hosts the TimingPanel component which shows which cycle is active per accelerator so that the users can choose an appropriate one. The left one hosts the setting and expert setting panels as tabs while the right one hosts the acquisition, UserData and BunchAcquisition panels as tabs. The representation of the data is on the right area of the frame and more specifically the acquisition tab is a graph of the total intensities as acquired and calculated from BFCTSR as well as two more devices for cross-checking, BCTDC3 and BCTDC4. The UserData tab hosts a graph of the individual bunch intensity measurements – one measurement at a time, while the BunchAcquisition tab hosts a 3D graph of the individual bunch intensity measurements – all together.

In figure 3-4 the Unified Modeling Language (UML) class diagram of our expert GUI is depicted according to entity separation of figure 3-3. The communication between two classes from a different group (Communication Manager, Intermediate Data Storage and GUI) is achieved with separated interfaces.

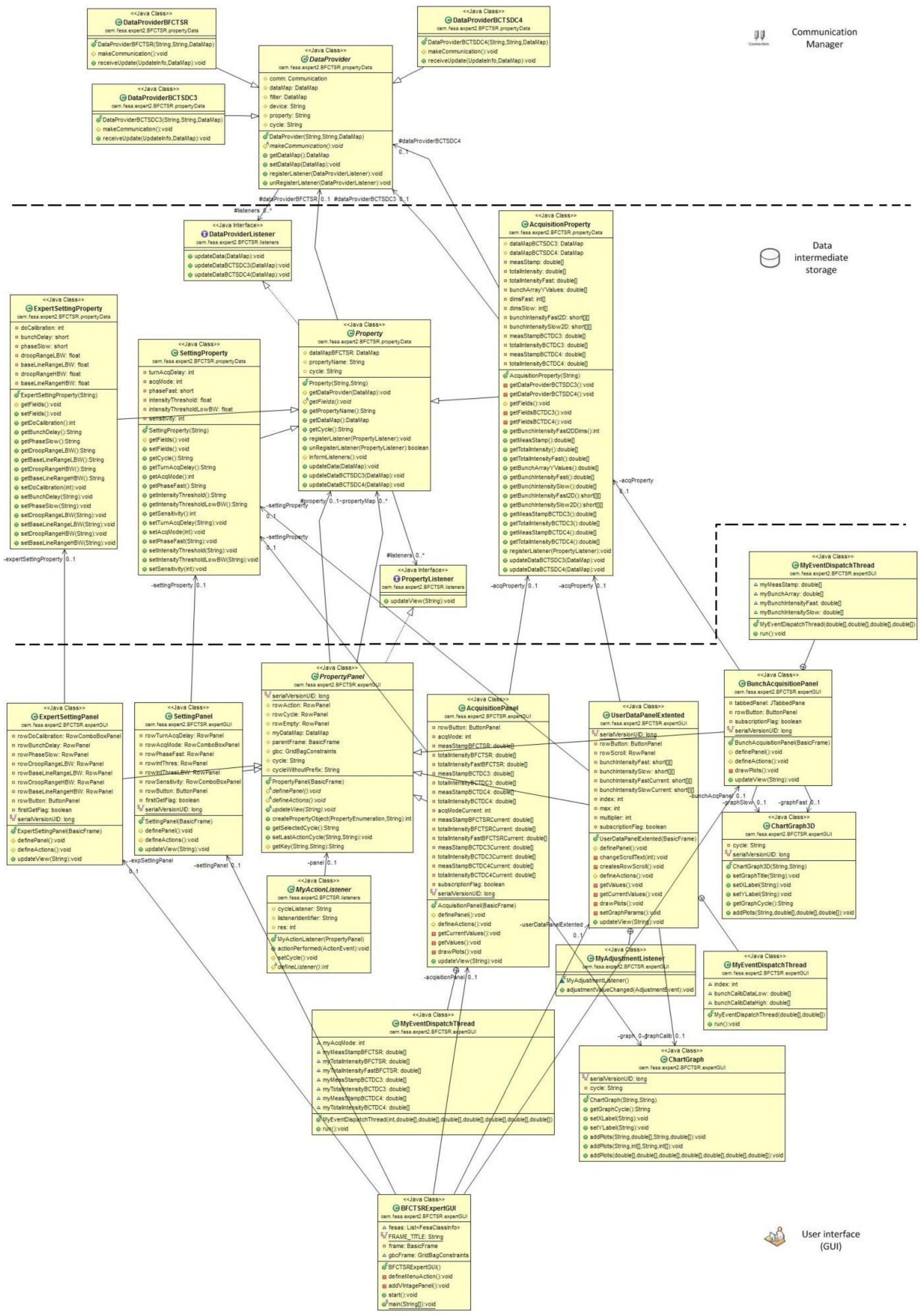


Figure 3-4: BFCTSR_ExpertGUI UML Class Diagram

3.1.5 LHC

In order to improve the performance of the FBCT measurements in the LHC while keeping the same update frequency of 1Hz (new values every second), it was decided to implement another approach as for the acquisition and calibration of the data using system C FBCT's new firmware. In this way, the acquisition is a simple ***Capture*** of the requested data (number of bunches for a specified number of turns) and all the computations for their process is done in the software. This approach allows us a degree of freedom in choosing which algorithms we use for the BLR, trying to achieve better accuracy when comparing this system with the other two.

The main idea of this approach is to make a full bunch acquisition for 25 turns with 224 turn interval. This means acquire 3564 bunch slots every 224 turns for 25 times as it can be seen in figure 3-5 and leads to a 25mA sampling over half a second. [21]

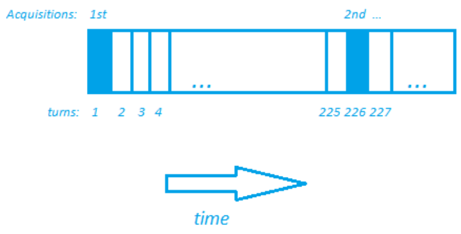


Figure 3-5: Acquisition schedule in respect with number of turn and turn interval

Since we have 4 cards and each one measures data for half a second, it would be impossible to implement a sequential scheduling and keep the 1 Hz publishing frequency. On the other hand, having one VME bus for communicating with all four cards makes it impossible to parallelize the parts of the process that consists of any kind of communication with the cards.

Hence, we decided to start the acquisition to all four cards almost at the same time and benefit of the acquisition's parallel nature. In this way, we spend half of a second for acquiring the data to all cards and keep the other half for processing them before publishing the total intensities. The process sequence of the data depicts in figure 3-6.

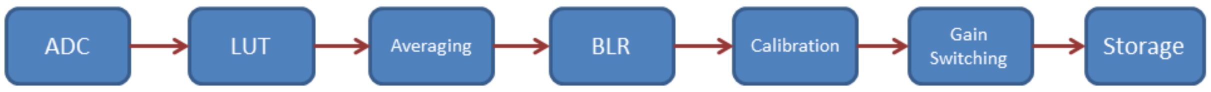


Figure 3-6: Data Process Sequence

3.1.5.1 Look Up Tables (LUT)

The integrator itself as well as the difference between the two integrators in the system is the main source of the overall non-optimal performance. In order to comprehend with this and treat both integrators as a black box, we performed a set of measurements in the laboratory analysing the linearity of the data. The results of this analysis can be summed as follows [21]:

- All measured integrators exhibit non-linear behaviour, which is not the same for each one and thus if corrected, it should be corrected per integrator
- An additional non-linear behaviour is exhibited in between each two integrators, due to the difference of their individual non-linear behaviour
- A linear approximation of the integrators' output is not enough to erase these non-linear components and thus higher order polynomial must be used instead

We decided that a reasonable approximation that would correct the non-linear behaviour quite decently – relative to the other two systems – is a polynomial of 5th order. Of course this would lead to long delays in the process of the data and hence we decided to measure each ADC approximation for each integrator and store these values to a unique comma-separated values (CSV) text file. Each file is unique per mezzanine and is named out of its serial number. It consists of 16384 text lines – the possible ADC values since they are 14 bits long – and each line consists of one integer – raw ADC value – and two floating values –corrected value for integrator 0 and 1 accordingly. Lastly, all LUTs are stored in our NFS section's directory so that they can be accessible from any FEC.

3.1.5.2 Averaging and Base Line Restoration (BLR)

Averaging the samples per bunch slot, as they come out of the LUTs, reduces the fluctuation of the signal caused by noise dramatically; this is due to the fact that the useful signal – beam – always comes at well specified moments during the RF cycle [2]. Hence, the more data we have to average, the clearer the result is.

Furthermore and for restoring the data's base line, we introduce a new algorithm based on the measurement of pure noise in the 3μs abort gap[3] as well of the noise at each empty bunch slot. Hence, we can summarize the algorithm for the BLR as follows:

- Find minimum after the LUT correction and averaging
- Specify the noise samples out of the 3564 which satisfy the following criteria:
    - The measured value falls in the interval of <min; min + TH>, where TH is a threshold value specified by the user
    - The position – bunch slot – of the measured sample is at least VS samples away from a non-noise sample, where the VS value is set by the user including 0
- Calculate the mean value of the selected noise samples
- Take away the calculated mean value from all the 3564 samples

An example of the above algorithm is depicted in figure 3-7. For this example the VS is 3, while the TH is of no significance. The samples that are considered as noise and thus are used for the calculation of their mean value are specified by the yellow regions.

[3] This is not actually true, since a limited amount of particles is always present and this can disturb the measurement [21]

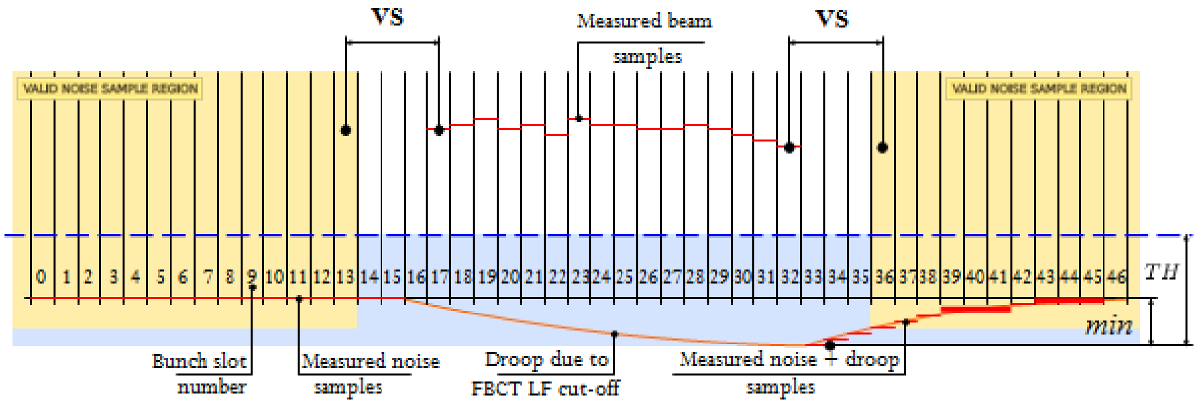


Figure 3-7: An example of the BLR algorithm with VS=3. Only the yellow region is considered as noise [21]

3.1.5.3 Calibration of the data

Calibration of the data is called the transformation of the ADC corrected values to the number of charges. This is done by applying a simple linear equation to the measured data:

$$NP = k * ADCcorrected + q \tag{3.1}$$

Where k is the calibration coefficient and q is the calibration offset, which both are normally found by calibration [2.2.1].

3.1.5.4 Gain Switching

As explained in chapter 2.2.1 both bandwidth channels provide two dynamic range measurements. Our software is responsible for the proper and automatic setting of the correct dynamic range, which depends on whether a bunch slot measurement exceeded a defined threshold. In order to avoid switching between gains when a measurement approaches the threshold we implemented a hysteresis in the switching thresholds. Hence, instead of one, we introduce two switching thresholds, settable by the user in ADC bins:

- *CHTH(high)* – this threshold is applied when the current measurement was performed by high gain measurement channel to switch to the low one, if at least one of the measured data exceeded it
- *CHTH(low)* – this threshold is applied when the current measurement was performed by low gain measurement channel to switch to the high one, if none of the measurement data exceeded it

3.1.5.5 Phase Scan

Phase scan is the observation of one bunch intensity – the maximum one – with its four neighbours (two from each side) when applying by brute force all 16 possible values for the phase delay expert setting. By changing the phase delay, the user can change the signal's amplitude and that is why this procedure is very important. The graph that comes out of this procedure can help the user to determine the appropriate phase delay setting in order to maximize the signal's amplitude. The following figure depicts one example of such a procedure that was performed[4] at system C, using a python script.

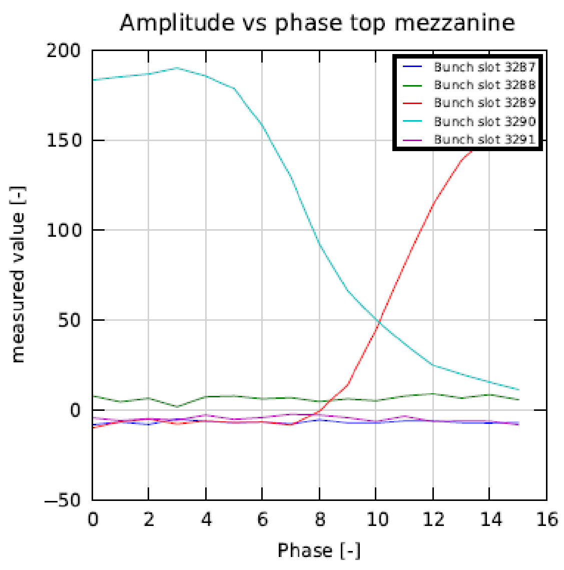


Figure 3-8: phase scan

3.1.5.6 *Server Architecture*

Four DAB cards are installed in LHC system C that measure the intensity of the beams using the FBCTs, one for the High and one for the Low Bandwidth measurements for each beam. Hence, we created a FESA class, BCTFRLHC v6, with four instances – one per card.

This server has two real-time actions:

- *Acquire* – where all the functionality of the server is implemented, such as data acquisition, process, BLR and storage. It operates every second.
- *XpocAction* – which is responsible to copy the history of the last 1000 total intensities as calculated by Acquire, as well as their time stamps to a different server at any beam dump event for diagnosing a possible reason for it

The properties that interface the server are:

- *Setting* – where the user can specify/observe the settings relative to the acquisition
- *CalibrationSetting* – where the user can specify/observe all the settings which are not relative to the acquisition
- *LoadLUT* – where the user can upload and clear the LUT for each mezzanine
- *Acquisition* – where the user can observe the total, bunch and history intensities for both mezzanines as well as the selected ones

[4] Performed by D. Belohrad

- *ExpertAcquisition* – where the user can observe the intermediate values before reaching the desired total intensities, such as the data after the LUT and BLR
- *XpocData* – where the user can observe the data copied from the XpocAction before being transferred to the server

3.1.5.7 Client – Interface

We developed the BCTFRLHC_v6 expert GUI in Java and organized it in five packages just as the BFCTSR_ExpertGUI [3.1.4.3], figure 3-3. The class diagram of the main part of the expert GUI is depicted in figure 3-9. This is the part that interfaces the server's properties *Setting*, *CalibrationSetting*, *LoadLUT*, *Acquisition* and *ExpertAcquisition* as well the graph from *Phase Scan*. These properties are organized in two areas – left and right. All the setting related panels – *Setting*, *CalibrationSetting* and *LoadLUT* – are placed at the left area as tabs whereas all the graph related panels – *Acquisition*, *ExpertAcquisition* and *Phase Scan* – are placed at the right side again as tabs.

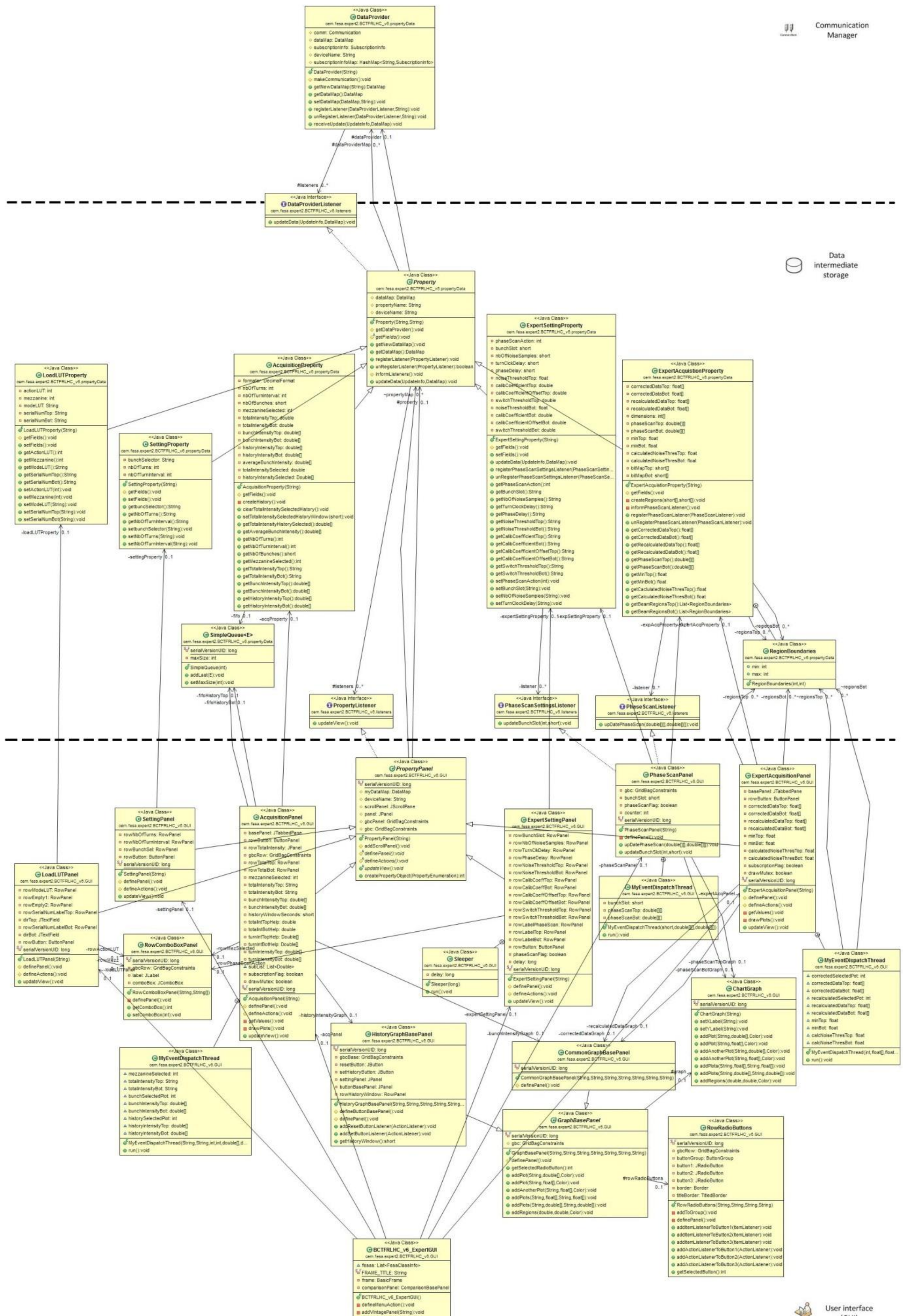


Figure 3-9: BCTFRLHC_v6 expert GUI UML class diagram (without comparison window)

In addition, we implemented a comparison window among the three systems – A, B and C – comparing the bunch intensities among the FBCTs of these systems and the total intensities among the FBCTs of these systems as well as the DCCTs of system A and B. Due to the lack of the calibration mechanism, we decided to implement this comparison window as part of the expert GUI for the FBCTs in system C, in order to ease the setting of the calibration coefficients and their monitoring. This comparison window's main purpose is to calibrate our FBCT's implementation of system C, relative to the existing implementations in system A and B, as well to cross check the accuracy of the data that our implementation provides. The class diagram of this comparison window is depicted in figure 3-10.

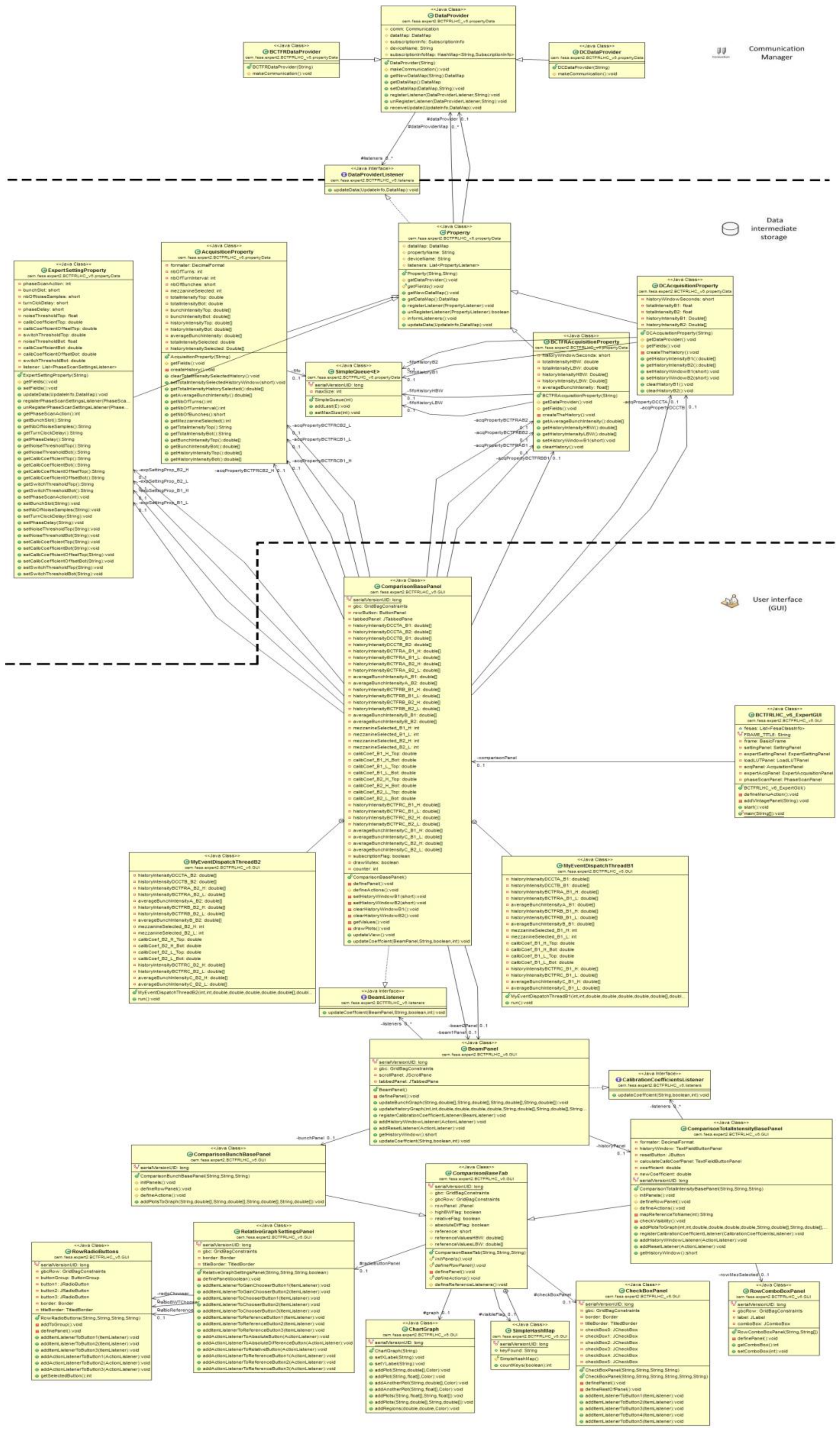


Figure 3-10: Comparison Window UML class diagram (part of the BCTFRLHC_v6 expert GUI)

3.2 Technical Implementation

After analysing the high level of our implementation for both systems – FBCTs in the SPS and LHC – we will try and give all the technical details that concern the implementation of the common tools – such as the rapper, tester and dabInfo – used by both systems independently as well the specific details by both systems individually.

3.2.1 Wrapper – Common Implementation

3.2.1.1 Constructor

Since we are using two libraries to access our hardware, they should be initialized somehow and this is done in the constructor of the wrapper class of our implementation. There, the *ioctl*'s function to open the device driver node is being called with two arguments, the Logical Unit Number (lun -- Logical Unit Number assigned to the module) and the Minor Device Number (chanN -- Minor Device Number. There can be several entry points for current Logical Unit Number (ChannelNumber)). It returns the file descriptor with which all the library's functions are called.

The *dal*'s function to enable the access to the device is being called with four arguments, the name of the device (as specified in the Data Base), and the method that will be used for the access (IOCTL, IOMMAP and IODMA), the LUN and chanN. It returns as well a file descriptor which is used when any of the library's methods are called.

3.2.1.2 Single-value Registers

For reading the single-value registers one can call the appropriate wrapper's method and pass a pointer to an integer as argument. The method is calling the *ioctl*'s function to get the register's value which returns a result of that action, if succeeded or failed. This result is stored in the address that was passed as argument to the wrapper's method, while the value of the register is being returned as unsigned long at the end of the method.

For writing a value to the single-value registers, the mechanism is quite similar with the above, with the difference that the value to be set is passed as an unsigned long argument along with a pointer to an integer. The method is using the *ioctl*'s function to write the value to the register and stores the result of that action (succeeded or failed) in the address passed as argument. The method doesn't return anything.

3.2.1.3 Multiple-value Registers

As for the multiple-value registers, we've implemented two ways of reading them. First is the type of methods that expect two arguments, one pointer to unsigned long and second to integer. This type of methods read the whole register and store it to the memory where the first pointer points and the result of that action to the second one.

The other type of methods that reads multiple values, take three arguments. One pointer to unsigned long for the result, one to an integer for the action's result as before and one additional integer to specify how many values to be read.

For writing this kind of registers, we've used the exact same implementation as above, with the only difference that we've used the appropriate libraries' functions for writing instead. Of course now, the first pointer points to the address where the values would be read and not written, meaning that became the source from destination.

3.2.1.4 Setting processing

There are also some methods to process the data that need to be set to the device before any action. These are the bunch selection which comes as a string from user's input. A parser was needed to be implemented in order to transform the user-friendly string to the array of hexadecimals that the device can take as setting through the CBunchSelector register.

The parser takes the string as argument and splits it to ' ' and ',' to find different selections. Then, it calls a private method to define if there is a region requested or a single bunch by searching the '-' character. And finally another private method is called to do the appropriate calculation and set the corresponding hexadecimals to the CBunchSelector register. The procedure is repeating itself until it reaches the end of the string.

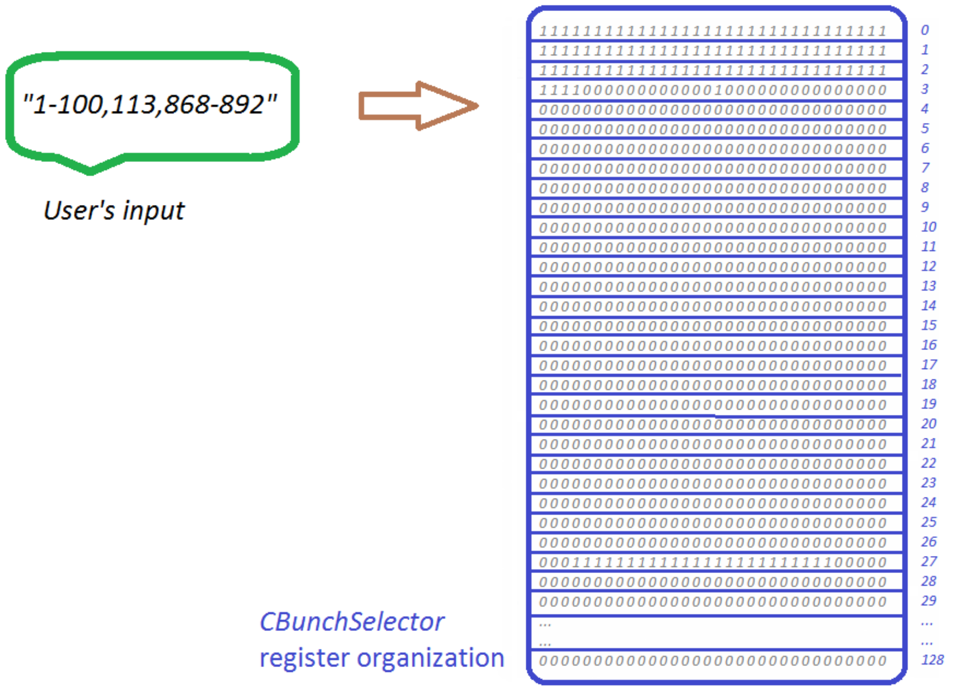


Figure 3-11: Bunch Selection Transformation from string to a set of hexadecimal

3.2.2 Tester – Common Implementation

The tester class was created at first to test the communication with the device. At those days, it did nothing more but to read and write the registers in order to make sure that every one of them behaves the way it should.

In the meanwhile, and as the project evolved, we found the need to develop new tests more relative to the acquisition behaviour of the device. Hence we implemented a loop that asks the user to enter the number of bunches and turns for acquisition while checking if this input is reasonable – no zero bunch selection for example. As it was described in the previous chapter, setting the bunch selector register is something that has to be done with great care, since errors in that procedure can mess the data and are extremely difficult to be spotted. That is why we implemented a CBunchSelector "parser" in the tester (which was moved later on to the wrapper). This parser is iterating the CBunchSelector memory (128 of doubles) and prints them as hex, so that we can debug its setting procedure.

Furthermore, the acquisition starts in a loop so that we can simulate real time conditions and the data are fetched from the device before passed to a method that decodes and prints them. The selection of the data is usually big enough and thus very uncomfortable to be printed in the console, hence the routine that does this job can take two arguments that specify two limits in order to print only the specified first and last samples.

The decoding of the data, which was moved to the wrapper later on, has to split the data as it was read from the device in the middle. Take the left part first (16 MSB) and apply a sign correction after striping the 14 less significant bits as follows:

```
temp = sample & 0x3FFF;           //keep only the 14 LSB

if(temp < 0x2000)                 //
      data = temp ;               // sign correction
else                              //
      data = temp - 0x4000;       //

integrator = (temp & 0x8000) >> 15 //MSB of the sample
saturation = (temp & 0x4000) >> 14 //2nd MSB of the sample
```

Figure 3-12: Sign correction of the data in the code

The same procedure must be followed to the right sample as well (16 LSB) before moving to the next element in the CBunchSelector memory. Special care should be taken when the number of samples – number of bunches * number of turns – is odd, in the sense that we keep only the desired and correct data. We achieved that by repeating the above procedure of splitting, striping and correcting the sign of the data one time less than is needed and taking modulo of the number of samples with 2 into account. In this way, we repeat the procedure for the left sample (16 MSB) and the right one (16 LSB) only if the modulo is 0.

3.2.3 DabInfo – Implementation

For the implementation of the dabInfo utility, we need the user to specify the LUN number of the DAB that he wishes to retrieve the information. After taking our Hardware expert's request under consideration, we agreed on having two ways to do that. If no argument was passed while running the application, a loop would ask the user to provide an appropriate LUN number. On the other hand, the user can directly pass this information with the running command.

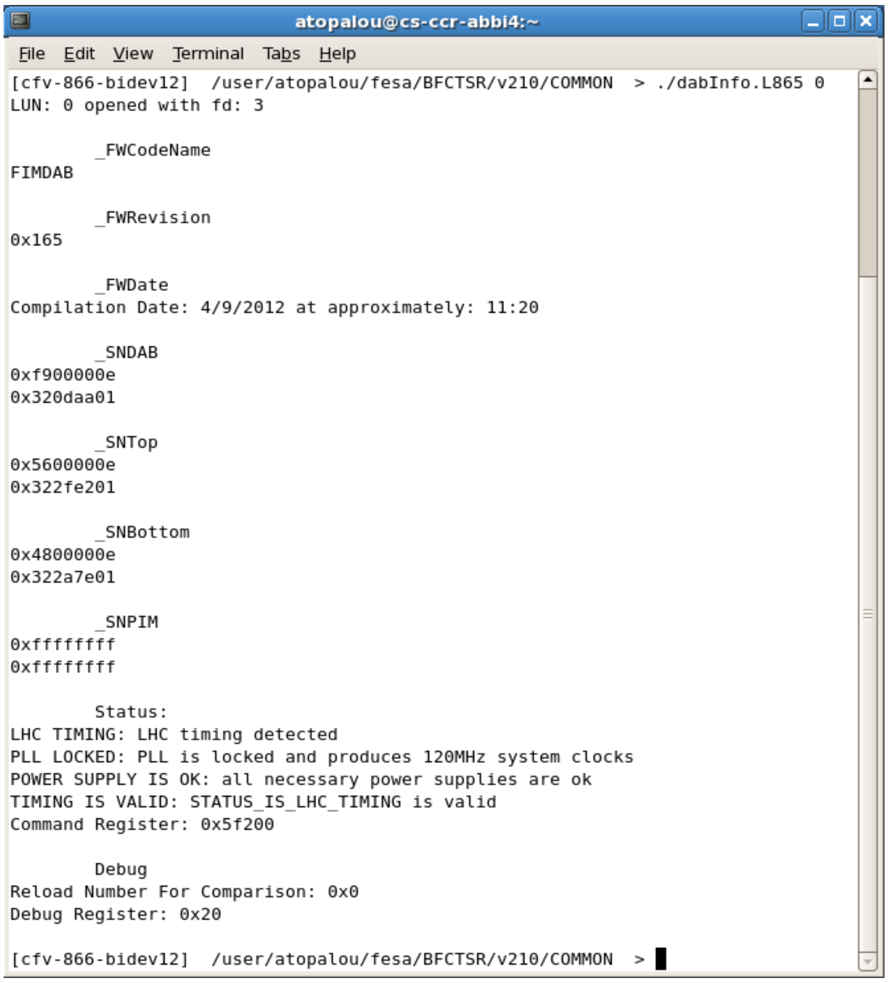
DabInfo does nothing more than reading directly (without using the wrapper class) 9 registers relative to the firmware, serial numbers and the status of the device – FWCodename, FWRevision, FWDate, SNDAB, SNTop, SNBottom, SNPIM, Command and Debug – and present their contexts in a meaningful way after processing them if needed.

For example, for printing in ASCII format the firmware codename, we split every element of the register at 4 pieces of 8 bits each and print each one of them as character. A code example is the following:

```
char c;                                        // temporary variable for printing
for (int i=0; i < 4; i++) {                    // register's length is 4 long words
        for (int j=0; j<4; j++) {              // 4 sets of 8 bits for every ASCII character
                c = FWCodeName[i] & 0xFF;      // keep only the 8 LSB of every long word
                printf ("%c", c);              // print them as character
                FWCodeName[i] = FWCodeName[i] >> 8;    // move to the next set of 8 bits
        }
}
```

Figure 3-13: example code for ASCII parsing

In a similar way, the FWDate has to be processed in order to extract the information about the day, month, year and time of the firmware compilation. Furthermore, and for the status (Command) and debug register we had to implement two hash tables, one for each register, with the possible status and debug states and print the corresponding message depending on the contexts of the appropriate register. An example of the output information when running dabInfo at the lab is the following:



```
[cfv-866-bidev12]  /user/atopalou/fesa/BFCTSR/v210/COMMON  > ./dabInfo.L865 0
LUN: 0 opened with fd: 3

        _FWCodeName
FIMDAB

        _FWRevision
0x165

        _FWDate
Compilation Date: 4/9/2012 at approximately: 11:20

        _SNDAB
0xf900000e
0x320daa01

        _SNTop
0x5600000e
0x322fe201

        _SNBottom
0x4800000e
0x322a7e01

        _SNPIM
0xffffffff
0xffffffff

        Status:
LHC TIMING: LHC timing detected
PLL LOCKED: PLL is locked and produces 120MHz system clocks
POWER SUPPLY IS OK: all necessary power supplies are ok
TIMING IS VALID: STATUS_IS_LHC_TIMING is valid
Command Register: 0x5f200

        Debug
Reload Number For Comparison: 0x0
Debug Register: 0x20

[cfv-866-bidev12]  /user/atopalou/fesa/BFCTSR/v210/COMMON  >
```

Figure 3-14: example run of the dabInfo in the lab

3.2.4 SPS

In this section we are focusing on the technical implementation details of the FBCTs in the SPS ring. We describe what changed in the software and in what way. Finally we describe the expert GUI that did not exist before.

3.2.4.1 Baseline Restoration (BLR)

The implementation of the new algorithm for the baseline restoration searches the acquired data for the minimum value. In order to detect and ignore extreme values, this is not enough. Hence, in the same loop, the minimum neighbour is determined so that its distance with the currently examined value can be tested and then decided if it will be considered as valid value or an extreme one.

In this way and within a single loop the minimum value of an acquisition, ignoring any "undershoots" is determined. Then the user setting that specifies the noise area is added to it in order to create a threshold that determines the samples below it to be considered as noise. Continuing in the second loop the average value of these noise samples is calculated, which is then removed from any sample in the acquisition. In this way, what is considered as noise moves to the zero area of the y axis. Figure 3-2 above shows such case.

3.2.4.2 TURN_BY_TURN acquisition

For the implementation of the new real time action rtTurnAcq, we basically combined the rtStart and endCapture into one new real time action with different settings. The main idea is the same; the rtTurnAcq starts the acquisition with the settings that are already in the device, reads the data back, decodes and calibrates them before exiting.

This acquisition mode acquires a full bunch selection for 500 consecutive turns (instead of 1 for the REPETIVE mode). This number is the limit of the first dimension of the intermediate and final buffers (number of measurements for the REPETIVE mode) which we also use in rtTurnAcq but storing the turn instead of the measurement in their first dimension. For the REPETIVE mode, 500 measurements every 40msec is more than enough and is never actually reached. As for the TURN_BY_TURN mode though, this number is really limiting the amount of data acquired, hence the precision of the measurement, when the capacity of the device storage exceeds this limitation by a factor of 2.

The main compatibility problem about this issue comes from our clients, people in the CCC who develop their own GUI applications to interface our servers. Their main request is to change their applications as less as possible to preserve stable releases of their software solutions. That is why we decided not to increase the maximum number of measurements/turns at developing time, but later on in the future and after we assure that the new version of the server works fine and stably.

Another implementation issue that appeared was the synchronization of the starting point of the real time action. The warning that starts the rtTurnAcq is 20msec earlier than the beam's injection. If we started the acquisition at this moment, we would acquire mostly noise and only a small fracture of the actual beam's intensity. Taking the limitation in our acquisition data that was introduced before under consideration, this would turn our new acquisition mode useless. To make things worse, this is the same event that wakes rtPrepare and serious problems would appear if both real time actions tried to communicate with the device since there is only one bus for this communication.

To avoid these problems, we had to wait some time – 18msec – just to assure the non-simultaneous device access as well as the acquisition of meaningful data. We implemented this delay using another FESA class that was created by our group for abstracting the global timing events, named LTIM, which gives us the opportunity to specify such settings as delay. We choose to implement this mechanism rather than using simple sleep commands, in order to reduce the useless CPU usage as well as preserving the wright synchronization among the real time actions.

3.2.4.3 Client – Interface

For the implementation of the expert GUI, we used the BasicFrameBuilder which was created from our section for abstracting the creation of certain useful toolkits such as the RBA toolbar as well as the device iterator. The latter – visible on the left side of figure 3-15 – creates a thread of the application for each device (instance of a FESA class) whiles the former – visible on the right side of the same figure – takes care of the privileges each user has for accessing each server.

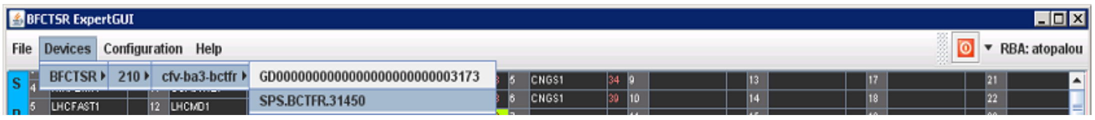
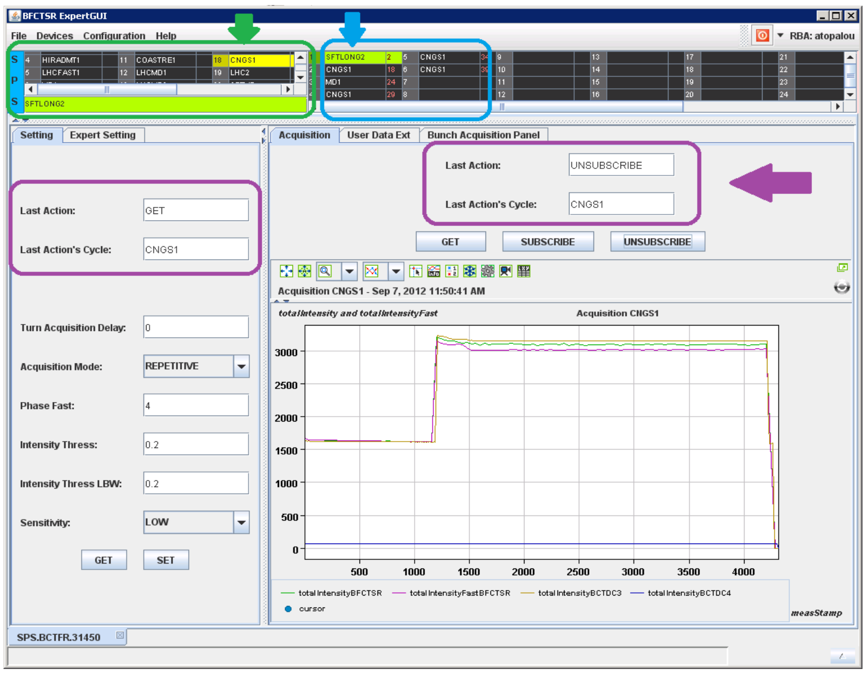


Figure 3-15: Upper part of the BFCTSR_EpertGUI

The TimingPanel is implemented by our section and its main purpose is to abstract the cycle multiplex for each accelerator. In this panel and at the right side, the user can see which cycle is active at any moment as well as the sequence of all active cycles for a given accelerator. At the left side of this panel, the user can choose by a simple click, which cycle's intensities he wants to observe. This information, as well as the type of the action the user requested (GET, SET, SUBSCRIBE and UNSUBSCRIBE), is visible in every panel of our application since things can complicate quite fast, if more than one cycle are observed at the same time.

In the figure 3-16 the cycle selection is visible inside the green box, where the green arrow points, while the sequence of the active cycles are inside the light blue box, pointed by the light blue arrow. Inside that box and with a green colour is the active cycle for that specific moment while the red numbers on the right side of each active cycle is its duration in seconds. Lastly and inside the purple boxes is the last action as well as the cycle for which it was operated. In the same figure the Setting as well as the Acquisition panel is visible.

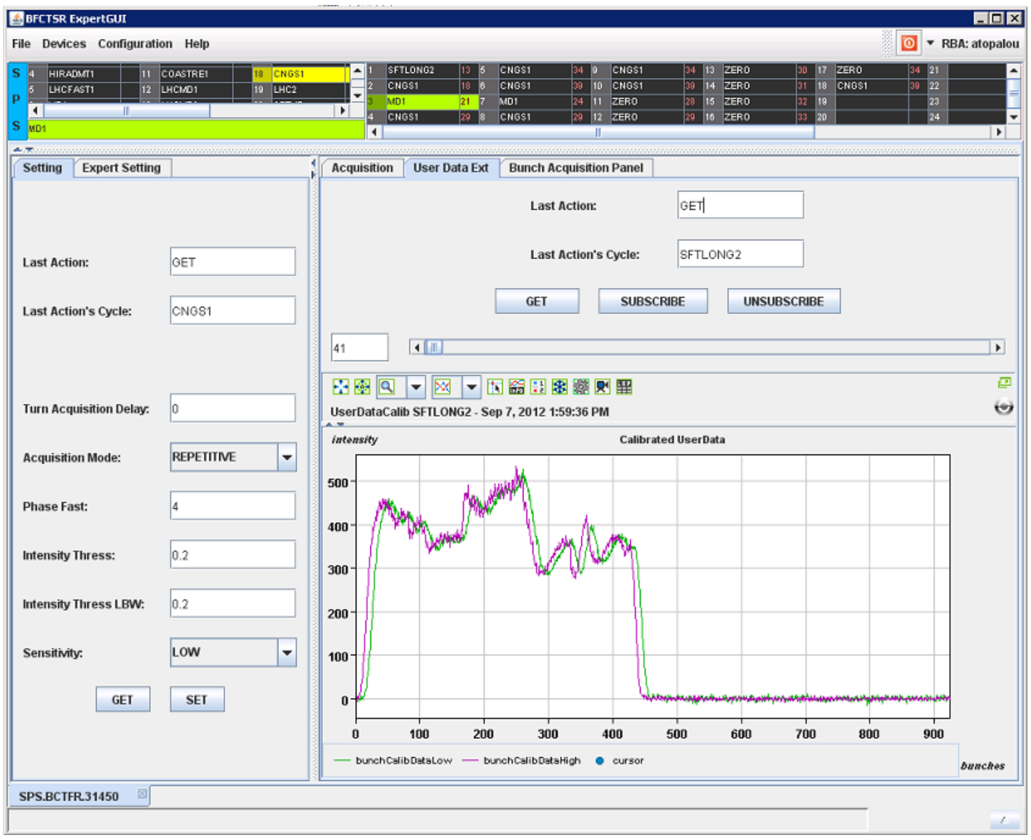


Last Action: GET
Last Action's Cycle: CNGS1
Turn Acquisition Delay: 0
Acquisition Mode: REPETITIVE
Phase Fast: 4
Intensity Thress: 0.2
Intensity Thress LBW: 0.2
Sensitivity: LOW

Last Action: UNSUBSCRIBE
Last Action's Cycle: CNGS1
Acquisition CNGS1 - Sep 7, 2012 11:50:41 AM

Figure 3-16: BFCTSR Expert GUI – Acquisition Tab

The UserData panel hosts a plot with the individual bunch intensities per measurement. There is also a scroll bar to iterate the different measurements as well as a text-field where the measurement offset in milliseconds is indicated. For example in figure 3-17 we can see the second measurement for the SFTLONG2 cycle with 41msec offset.


BFCTSR ExpertGUI

File Devices Configuration Help

RBA: atopalou

4	HIRADMT1	11	COASTRE1	18	CNGS1
5	LHCFAST1	12	LHCMD1	19	LHC2

MD1

1	SFTLONG2	13	5	CNGS1	34	9	CNGS1	34	13	ZERO	30	17	ZERO	34	21	
2	CNGS1	18	6	CNGS1	39	10	CNGS1	39	14	ZERO	31	18	CNGS1	39	22	
3	MD1	21	7	MD1	24	11	ZERO	28	15	ZERO	32	19			23	
4	CNGS1	29	8	CNGS1	29	12	ZERO	29	16	ZERO	33	20			24	

Setting | Expert Setting

Last Action: GET

Last Action's Cycle: CNGS1

Turn Acquisition Delay: 0

Acquisition Mode: REPETITIVE

Phase Fast: 4

Intensity Thress: 0.2

Intensity Thress LBW: 0.2

Sensitivity: LOW

GET SET

Acquisition | User Data Ext | Bunch Acquisition Panel

Last Action: GET

Last Action's Cycle: SFTLONG2

GET SUBSCRIBE UNSUBSCRIBE

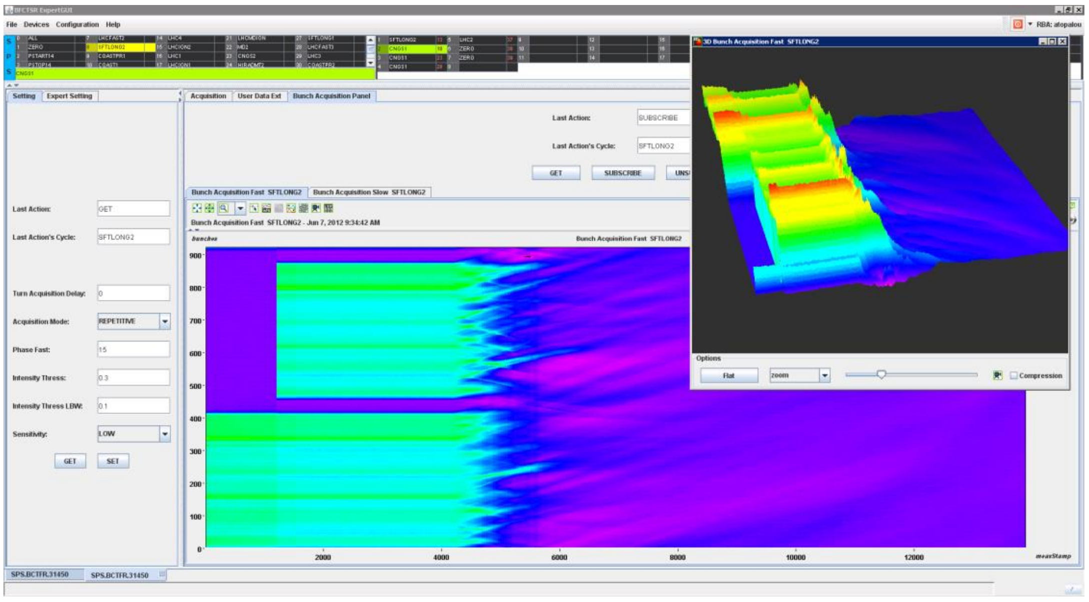
41

UserDataCalib SFTLONG2 - Sep 7, 2012 1:59:36 PM

SPS.BCTFR.31450

Figure 3-17: BFCTSR Expert GUI – UserData Tab

Lastly, the BunchAcquisition panel hosts two 3-Dimensional plots, one for each mezzanine. These 3D graph components were experimentally created by our group and found to be quite useful in our case, since we can have a global idea of the individual bunch intensity measurements in time at once. The data that are being presented by both BunchAcquisition and UserData panels are the same – the two dimensional arrays from the server – only with a different representation. The UserData panel is very useful for the individual study of the measurements whereas the BunchAcquisition is ideal for the whole picture of the measurement. An example of the latter panel can be seen in figure 3-18 along with the 3D pop-up graph.


Last Action: GET
Last Action's Cycle: SFTLONG2
Turn Acquisition Delay: 0
Acquisition Mode: REPETITIVE
Phase Fast: 15
Intensity Thress: 0.3
Intensity Thress LBW: 0.1
Sensitivity: LOW

Last Action: SUBSCRIBE
Last Action's Cycle: SFTLONG2

Bunch Acquisition Fast SFTLONG2 - Jun 7, 2012 9:34:42 AM

Figure 3-18: BFCTSR Expert GUI – BunchAcquisition Tab

3.2.5 LHC

For the server implementation in the LHC ring, we decided to keep the four pointers to the wrapper class – one per DAB card – apart from the shared memory. The design is such, that either way, we iterate through the device collection – four DABs – in order to start the acquisition, read back the data, set the settings and so on. This iteration is done always in the same order and it starts from the device in lun 0 – HBW for beam1 – and it goes up to the device in lun 3 – LBW for beam2. Hence, we create and initialize these four pointers to the wrapper class in the constructor of the real time classes, BCTFRLHCRealtime and after storing them to an array in the same order of the devices, we access them through our server classes using the keyword *extern*.

3.2.5.1 Look Up Tables

As described in chapter 3.1.5.1, there are two LUTs per DAB card – one for each mezzanine. The LUTs contain the signed corrected ADC values (-8192 – 8191) and the two corrected values – one for each integrator. We implemented the LUTs in software in two arrays of floats per LUT – one for integrator 0 and one for 1. We used the ADC values as indexes to each corrected floating value for each integrator's array, after eliminating the sign correction by subtracting the constant value 8192, in order to have proper positive array indexes. These arrays are stored in the device shared memory, so that they can be accessed by any server class at any time.

The implementation of the software LUTs is done in a custom class that is accessible by any class of our server. This class has hardcoded the path where the LUTs are placed and takes a pointer to a BCTFRLHC device as a constructor's argument. The array of the wrapper pointers is also visible in that class using the key word *extern*.

Furthermore, this class has to methods:

- clearLUT(int) – which clears the software LUTs for the specified mezzanine (0 for both, 1 for the top and 2 for the bottom)
- updateLUT() – which loads or reloads the LUTs according to the settings the user has provided in loadLUT property

By clearing the LUTs, we mean to make them (1:1) transparent in order to avoid our server from crashing. In other words, the LUTs return the same value that was used for indexing, without any non-linear correction. This is also very important to check the raw ADC values as they are read from the DABs, since they are not published at all to avoid making our properties "heavy".

Updating the LUTs at runtime, is a feature much appreciated by the users, since they can change them (clearing/updating) in order to observe, as said, the raw values if needed. In addition and if it is found that they need to be changed in the future, this can be done on the fly without spending too much time rebooting the server.

The LUTs are loaded for the first time to the shared memory at BCTFRLHCRealtime class which is responsible for any kind of initialization of the real-time classes when the server starts. If by any reason this operation fails, the ones that failed are being cleared.

3.2.5.2 Averaging, Base Line Restoration (BLR) and Calibration of the Data

Since there is the 1 second time restriction, we tried to condense as many of the data process steps as possible. Hence, when we iterate the acquired values <*number_of_turns* * *number_of_bunches*> and parse them through the appropriate LUTs, we also sum the corrected values per bunch slot. Furthermore, in a second iteration <*number_of_bunches*> we divide every sum with the <*number_of_turns*> to get the average bunch intensities after LUT correction. In this iteration, we also specify the minimum average bunch values to be used from the next steps of the data process.

For implementing the BLR as described in chapter 3.1.5.2, we decided to use two arrays of shorts – one per mezzanine – that we called *bitmaps* and specify if a bunch slot contains noise or beam signal – 1 or 0 accordingly. Obviously, these arrays' length is the maximum number of the bunch slots that can be acquired – 3564. In addition, these *bitmaps* are initialized with 1, assuming that every single bunch slot contains noise measurement which is the case when the beam is not present.

Subsequently, we iterate the averaged LUT corrected values from <*VS*> (see chapter 3.1.5.2) to <*number_of_bunches* – *VS*> checking if the value is above <min + TH>. If it is, then it means that this bunch slot measurement should be considered as beam signal and hence the corresponding entry of the *bitmap* is changed to 0. Then, we check the measured values just before and after the current one, to specify if this bunch slot is at the beginning – the previous value should be below <min + TH>, end – the next value should be below <min + TH> – or in the middle of the beam. If any of the two former cases appear, we also change the *bitmap* for the according bunch slots – previous or next – to 0. This is done for both top and bottom mezzanines.

Furthermore, we iterate the first *VS* values as well as the last ones in case there is beam signal at these bunch slots, in which case we change the *bitmap* for these bunch slots to 0. By the end of these iterations, we have all the information needed to calculate the mean value of the noise in the *bitmaps*.

Thus, we iterate once more the averaged LUT corrected values <*number_of_bunches*> and we sum the values that have 1 at the corresponding index of the *bitmaps*, increasing also a counter for every noise sample. In this way we specify the mean value per mezzanine by dividing the sum with the counter.

Lastly, we take away the just calculated noise mean value from every sample at the same time we transform them to number of charges by applying the calibration components. Hence, the equation 3.1 is transformed to the following:

$$NP = k * (ADCcorrected - noiseMeanValue) + q \tag{3.2}$$

In addition, this is the iteration where we sum the calibrated values – number of charges – and calculate the average total intensity for both mezzanines, that one of which will be published. We also find the maximum value as well its bunch slot that will potentially be used by the phase scan actions.

3.2.5.3 Gain Switching

In order to implement the gain switching in software, the user provides two switching thresholds in ADC bins. But these thresholds are applied to the data after their calibration – in number of charges – and thus, the same transformation (equation 3.2) must be applied to them.

After transforming the thresholds, we read back from the shared memory which was the previous selected gain, and apply the thresholds accordingly. If it was the top mezzanine, then we iterate the averaged calibrated values and if we find at least one value that exceeds the threshold, we break and we switch the gain to the bottom mezzanine. On the other hand, if the previous gain selection was the bottom mezzanine, we simply check if the maximum value that was already found from the calibration-BLR iteration exceeds the according threshold and if it does not, we switch to the top one.

3.2.5.4 Phase Scan

For the implementation of the phase scan, we use the settings that the user has provided at *CalibrationSetting* property and more specifically the phase scan action selection and the bunch slot. We support two actions and thus the *phaseScanAction* field has three possible states:

- DO_NOTHING – is the default state of that field and as its name reveals, is used for doing nothing as far as the phase scan procedure is concearned
- FIND_MAX_BUNCH_SLOT – is the state of that field that instructs the real-time action to store at *bunchSlot* field the bunch slot with the maximum value of the selected gain, as found from the calibration-BLR iteration, from the current measurement

- DO_PHASE_SCAN – is the state of that field that instructs the real-time action to apply the phase scan at the specified bunch slot, given by the *bunchSlot* field

The latter, needs 16 acquisitions – 16 seconds – to be completed. We keep the phase delay that was last used for the phase scan, in a private field so that it doesn't mess up with the phase delay the user provided in the *CalibrationSetting* property. The values of the 5 bunch slot measurements are stored in different 2D buffers whose first dimension is the 5 different bunch slots whereas the second one is the 16 values according to the 16 possible values of the phase delay. Each second, we increase the private phase delay by one and check if we reached the end, where we set it to its initial value (0) and the *phaseScanAction* field to its default value (DO_NOTHING).

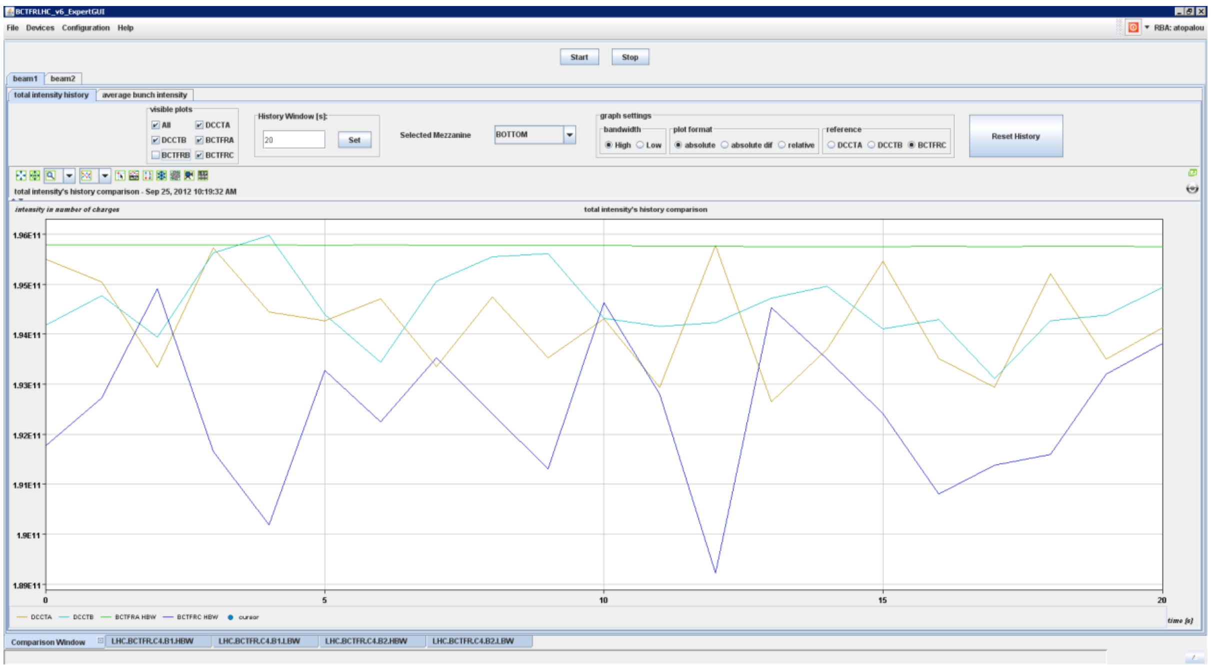
3.2.5.5 Client – Interface

We implemented the BCTFRLHC_v6_ExpertGUI, using the basic frame builder just as for the BFCTSR_ExpertGUI (see chapter 3.2.4.3) in order to take advantage of the automatic implementation of the device list as well as the RBAC toolbar.

The expert GUI consists of two main tabs:

- *Comparison Window* – which interfaces the comparison application described in chapter 3.1.5.7 – figure 3-10
- *Device Window* – which interfaces our expert GUI per device instance as it was described in chapter 3.1.5.7 – figure 3-9

The *Comparison Window* consists of a row of buttons on top – Start / Stop, and two tabs – one per beam. Each beam tab consists of two tabs as well – one for the history of the total intensities and one for the average bunch intensities. The latter two tabs consist of a toolbar on top and a graph at the remaining area. The toolbar is different per tab and that is because there are different settings depending on the type of the graph.



visible plots: [✓ All] [✓ DCCTA] [✓ DCCTB] [✓ BCTFRA] ☐ BCTFRB [✓ BCTFRC]

History Window [s]: 20 Set

Selected Mezzanine: BOTTOM

graph settings — bandwidth: [✓ High] Low; plot format: [✓ absolute] absolute dif relative; reference: DCCTA DCCTB [✓ BCTFRC]

Figure 3-19: Comparison Window - total intensity history for beam 1

Hence, the toolbar for the total intensities tab consists of a group of checkboxes where the user can specify the visibility of the available plots – these are the history of the total intensity as calculated from DCCTA and DCCTB as well from FBCTs in all three systems. Next to these checkboxes, lie a text-field and a button that allows the user to specify the depth of the history he desires. This is achieved by changing accordingly the length of the First-In-First-Out (FIFO) queues we use to create the history plots from all devices. In addition, a reset button clears these queues, in case the user wants to restart the history monitoring. Furthermore, we state which mezzanine was used to provide the total intensity as far as our server is concerned in the next component which consists of a label and a combo box. Subsequently, three sets of radio buttons lie next to the selected mezzanine that group the settings related to the graph. The first of these sets specifies which bandwidth to plot from each device – High or Low. The second set specifies the graph format – absolute, absolute difference and relative difference – and the third one the references – DCCTA, DCCTB and FBCTC.

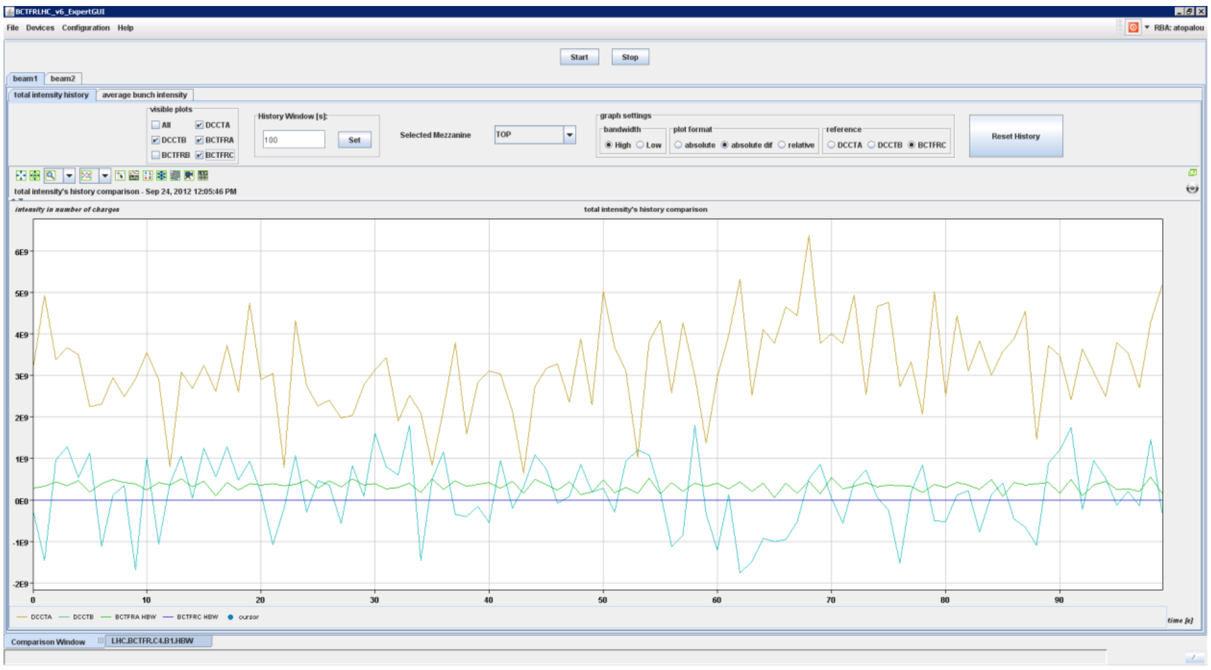


Figure 3-20: Comparison Window - total intensity history - absolute difference - for beam 1

By absolute, we mean that we plot the total intensity histories as we get them from the devices. For the other two formats – absolute and relative difference – we use the values from one device as reference – the user specifies which one he wants from the third set of radio buttons – and we calculate the difference of the visible plots relative to the reference ones. In the absolute difference format, we just subtract the reference values from the visible ones. On the other hand and for the relative difference format, we use the following equation to calculate the percentage difference between two systems:

$$difference = \left(\frac{visible\ values - reference\ values}{reference\ values}\right) * 100 \qquad (3.3)$$

The result of the absolute difference format is a graph of the difference between the visible systems relative to the specified one in number of charges, whereas in the case of relative difference is the percentage of this difference. In addition and only for the relative difference format, if there is only one visible plot and at least one of the two settings – visible and relative – is system C but without being the same to both settings, we make visible another component which consists of a text-field and two buttons. This component is used to calculate and apply the corresponding calibrating coefficient for system C in a way to eliminate the difference as much as possible. This is achieved by calculating the next equation using the values retrieved by equation 3.3 and the most recently used calibration coefficient:

$$new\ coefficient = \left(1 + \frac{difference}{100}\right) * old\ coefficient \qquad (3.4)$$

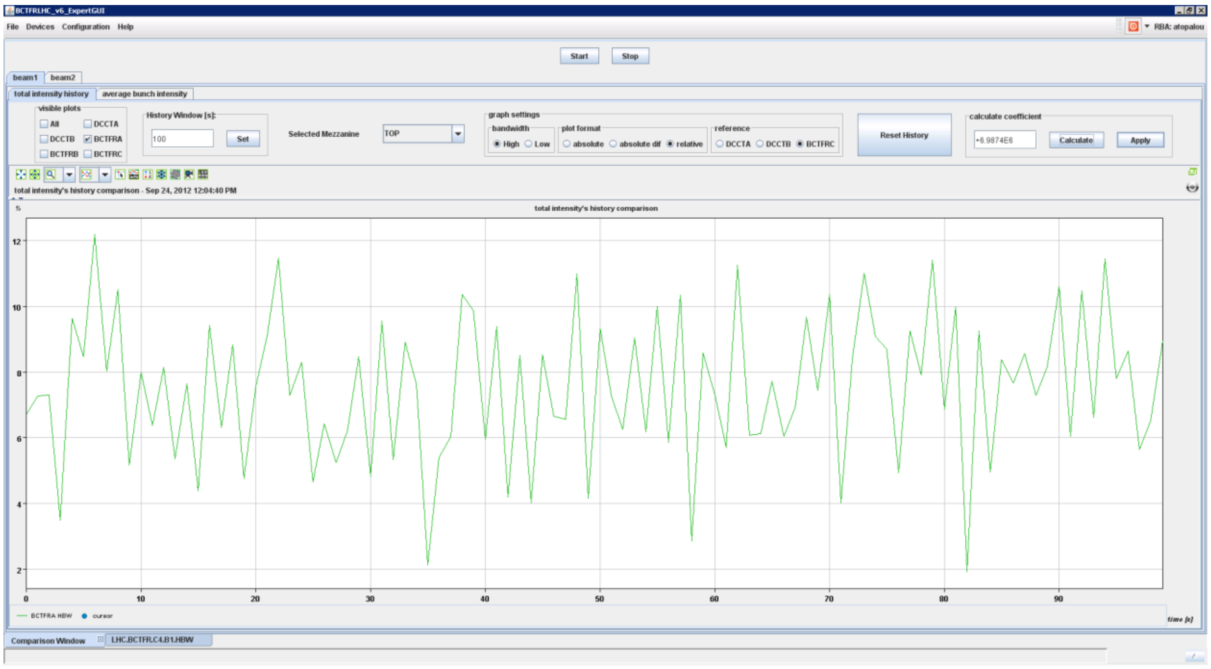


Figure 3-21: Comparison Window - total intensity history - relative difference for beam 1

As for the toolbar of the average bunch intensities tab, things are simpler since it consists only by a smaller group of checkboxes and two sets of radio buttons. The checkboxes are again to allow the user to specify which available plots he wishes to make visible – these are the average bunch intensities as calculated from the three FBCT systems. This is because the DCCTs do not provide bunch-to-bunch measurements. The radio buttons are again to specify the graph settings as in the total intensity history tab's toolbar but this time without the bandwidth chooser nor the additional coefficient calculator component. The lack of the former is due to the fact that system A and B do not provide bunch-to-bunch measurements for the LBW whereas the coefficient calculator is focused on the published values which are the total intensities.

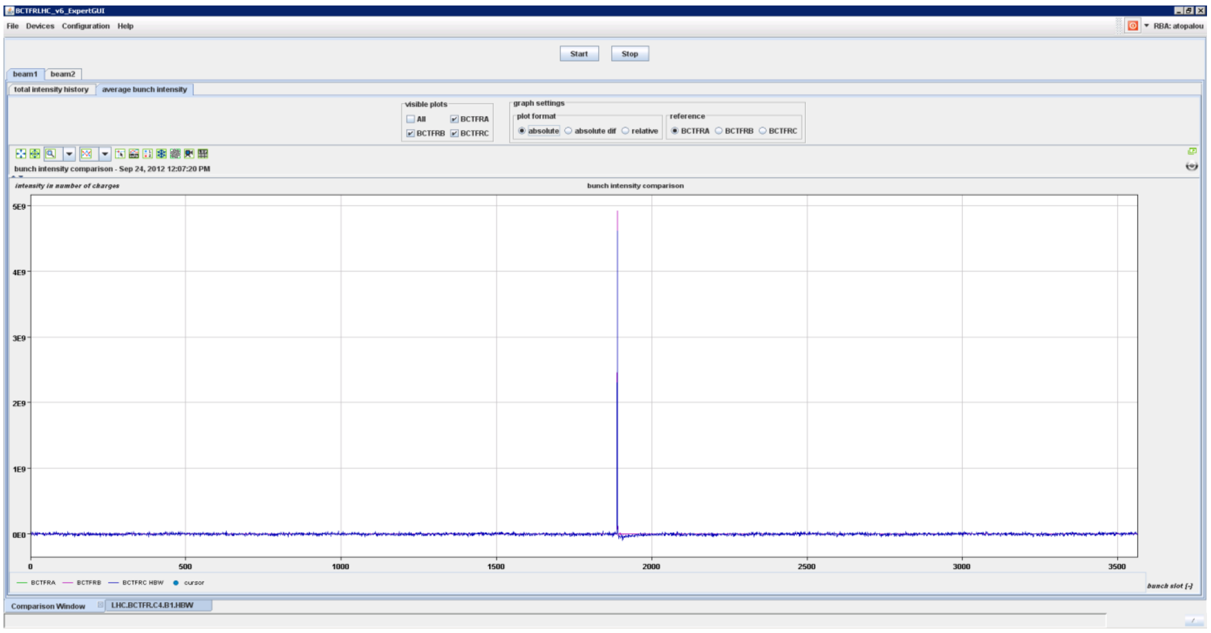
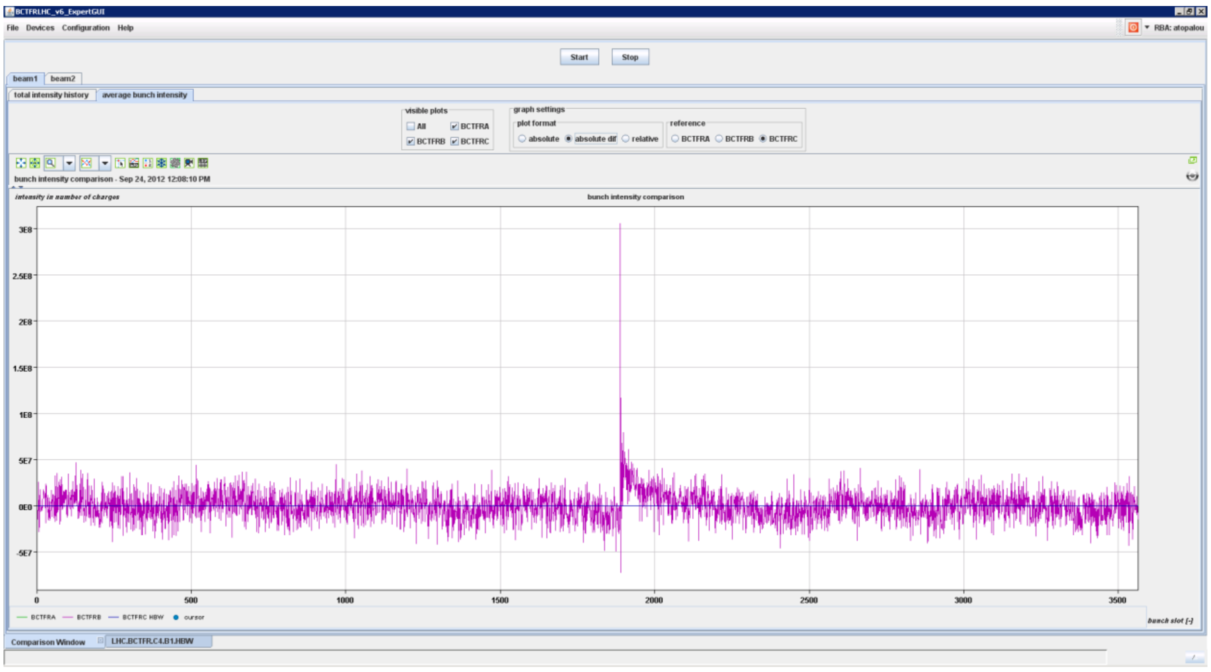


Figure 3-22: Comparison Window - average bunch intensity for beam 1

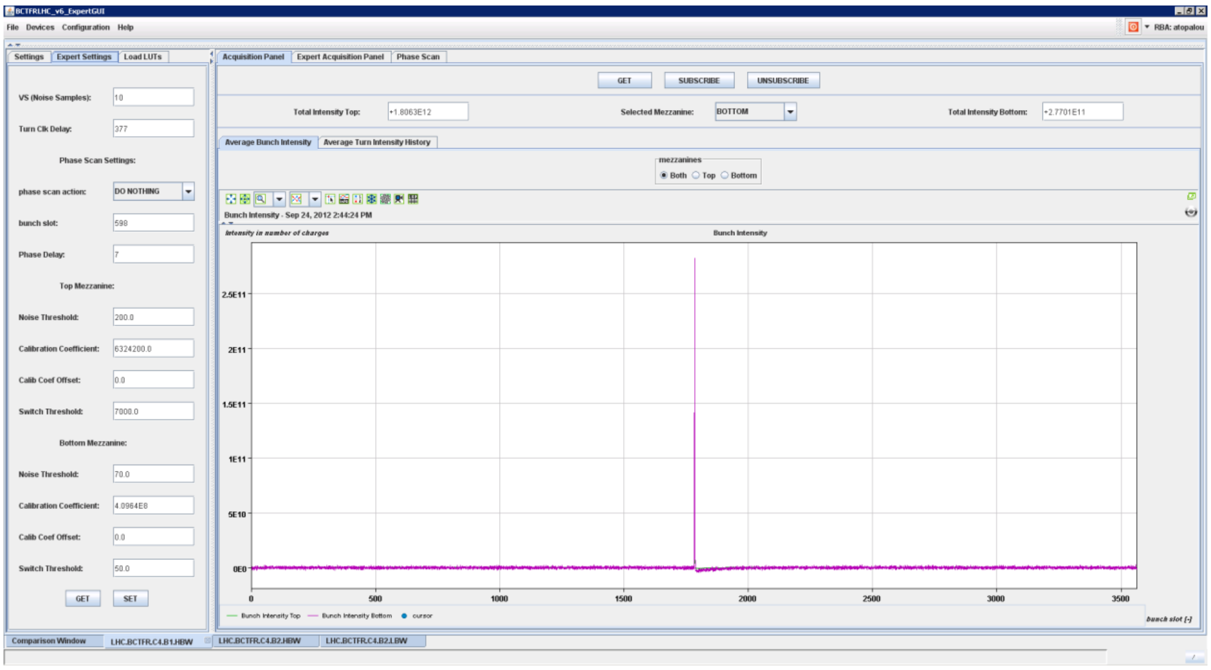


visible plots: All ☐ BCTFRA ☑ BCTFRB ☑ BCTFRC ☑

graph settings — plot format: absolute [✓ absolute dif] relative — reference: BCTFRA BCTFRB [✓ BCTFRC]

Figure 3-23: Comparison Window - average bunch intensity - absolute difference for beam 1

The *Device Window* consists of an area of setting – *Setting*, *ExpertSetting* and *LoadLUT* – panels on the left of the GUI, that interface the corresponding FESA properties and the graphics area on the right with acquisition panels – *Acquisition*, *ExpertAcquisition* and *PhaseScan*. In this way, the user is able to spot immediately the reaction of his settings to the data acquired.



VS (Noise Samples): 10
Turn Clk Delay: 377

Phase Scan Settings:
phase scan action: DO NOTHING
bunch slot: 598
Phase Delay: 7

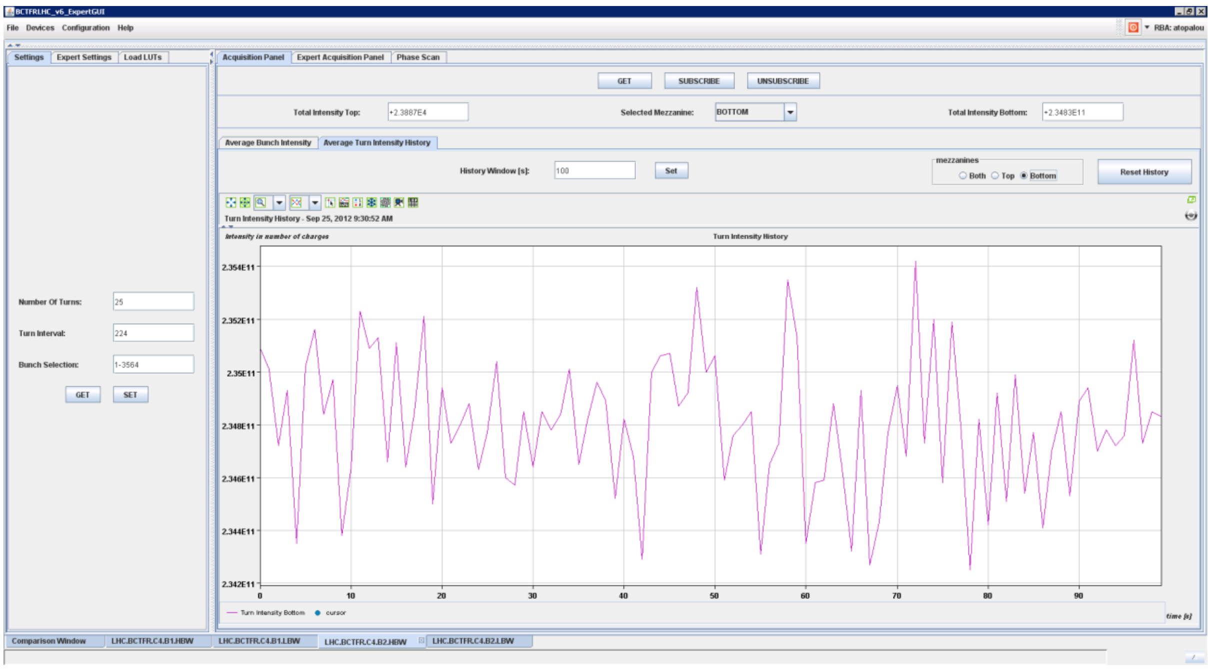
Top Mezzanine:
Noise Threshold: 200.0
Calibration Coefficient: 6324200.0
Calib Coef Offset: 0.0
Switch Threshold: 7000.0

Bottom Mezzanine:
Noise Threshold: 70.0
Calibration Coefficient: 4.0964E8
Calib Coef Offset: 0.0
Switch Threshold: 50.0

Total Intensity Top: +1.8063E12
Selected Mezzanine: BOTTOM
Total Intensity Bottom: +2.7701E11

mezzanines: [✓ Both] Top Bottom

Figure 3-24: BCTFRLHC_v6_ExpertGUI - Acquisition Panel / Average Bunch Intensity - Expert Settings Panel

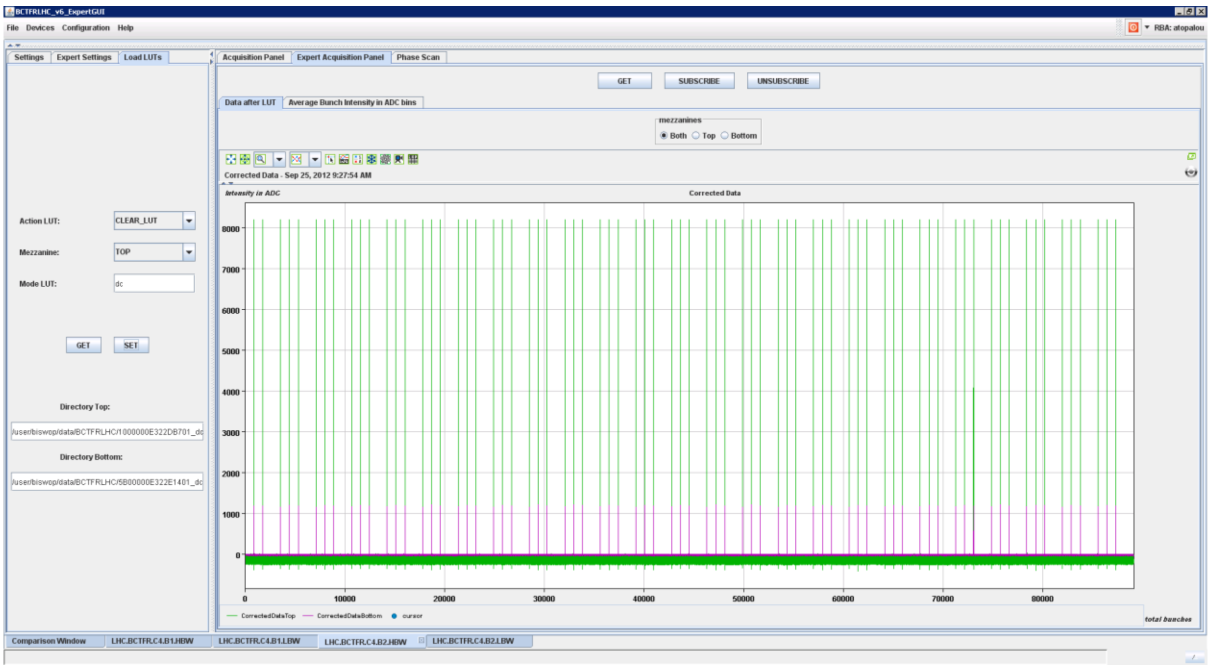

BCTFRLHC_v6_ExpertGUI
File Devices Configuration Help
RBA: atopalou
Settings Expert Settings Load LUTs
Acquisition Panel Expert Acquisition Panel Phase Scan
GET SUBSCRIBE UNSUBSCRIBE
Average Bunch Intensity Average Turn Intensity History
Set
Reset History
GET SET
Turn Intensity History
Intensity in number of charges
time [s]
Turn Intensity Bottom
cursor
Comparison Window LHC.BCTFR.C4.B1.HBW LHC.BCTFR.C4.B1.LBW LHC.BCTFR.C4.B2.HBW LHC.BCTFR.C4.B2.LBW

Number Of Turns: 25
Turn Interval: 224
Bunch Selection: 1-3564
Total Intensity Top: +2.3887E4
Selected Mezzanine: BOTTOM
Total Intensity Bottom: +2.3483E11
History Window [s]: 100
mezzanines: Both Top [✓ Bottom]
Turn Intensity History - Sep 25, 2012 9:30:52 AM

Figure 3-25: BCTFRLHC_v6_ExpertGUI - Acquisition Panel / Average Turn Intensity History - Settings Panel

The history tab of the average turn intensities under the *Acquisition* panel is exactly the same graph with the *total intensity history* tab in the *Comparison Window* if the user selects the appropriate settings from its toolbar. In the example shown in figure 3-25, one should choose to plot the BCTFRC values at the *beam 2* tab with HBW and absolute graph format as graph settings. And this is true, only if the currently selected mezzanine (GAIN) from FBCT in system C is bottom (Low).

The next two figures (3-26 and 3-27) depicts the impact of the LUTs at the data. For this reason we plot the data as soon as they are parsed from the LUTs in the *Expert Acquisition* panel, *Data After LUT* tab. In the first figure we cleared (1:1) the LUT for the top mezzanine only so that the difference between the actual and the cleared LUTs can be spotted easily. The second figure depicts the data after updating "on the fly" the top mezzanine's LUT.


BCTFRLHC_v6_ExpertGUI
File Devices Configuration Help
RBA: atopalou
Settings Expert Settings Load LUTs
Acquisition Panel Expert Acquisition Panel Phase Scan
GET SUBSCRIBE UNSUBSCRIBE
Data after LUT Average Bunch Intensity in ADC bins
GET SET
Intensity in ADC
Corrected Data
total bunches
CorrectedDataTop CorrectedDataBottom cursor
Comparison Window LHC.BCTFR.C4.B1.HBW LHC.BCTFR.C4.B1.LBW LHC.BCTFR.C4.B2.HBW LHC.BCTFR.C4.B2.LBW

Action LUT: CLEAR_LUT

Mezzanine: TOP

Mode LUT: dc

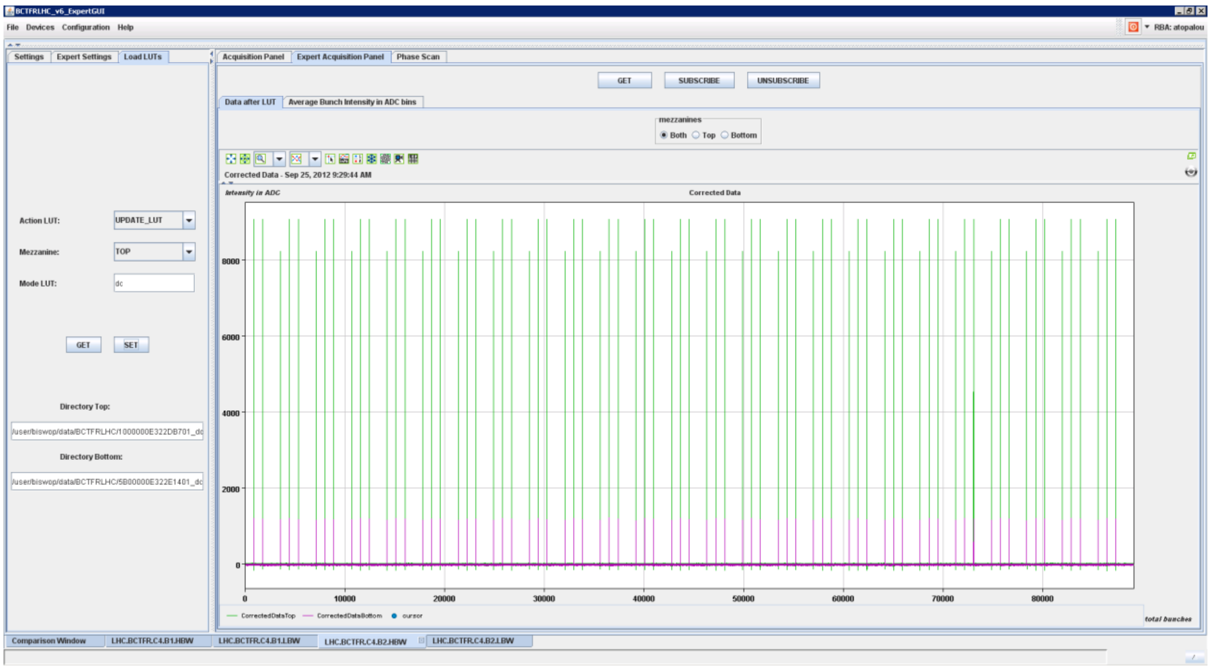
Directory Top: /user/biswop/data/BCTFRLHC/1000000E322DB701_dc

Directory Bottom: /user/biswop/data/BCTFRLHC/5B00000E322E1401_dc

mezzanines: [● Both] ○ Top ○ Bottom

Corrected Data - Sep 25, 2012 9:27:54 AM

Figure 3-26: BCTFRLHC_v6_ExpertGUI - Expert Acquisition Panel / Data after LUT - cleared LUT for top mezzanine


BCTFRLHC_v6_ExpertGUI
File Devices Configuration Help
RBA: atopalou
Settings Expert Settings Load LUTs
Acquisition Panel Expert Acquisition Panel Phase Scan
GET SUBSCRIBE UNSUBSCRIBE
Data after LUT Average Bunch Intensity in ADC bins
GET SET
Intensity in ADC
Corrected Data
total bunches
CorrectedDataTop CorrectedDataBottom cursor
Comparison Window LHC.BCTFR.C4.B1.HBW LHC.BCTFR.C4.B1.LBW LHC.BCTFR.C4.B2.HBW LHC.BCTFR.C4.B2.LBW

Action LUT: UPDATE_LUT

Mezzanine: TOP

Mode LUT: dc

Directory Top: /user/biswop/data/BCTFRLHC/1000000E322DB701_dc

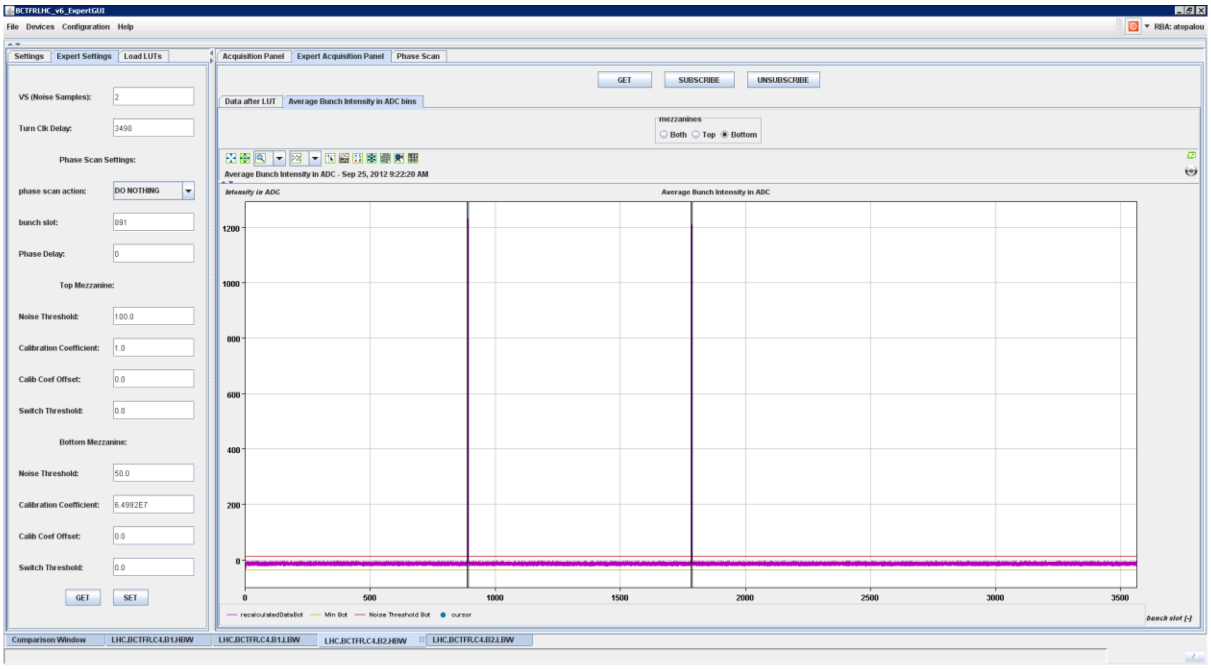
Directory Bottom: /user/biswop/data/BCTFRLHC/5B00000E322E1401_dc

mezzanines: [● Both] ○ Top ○ Bottom

Corrected Data - Sep 25, 2012 9:29:44 AM

Figure 3-27: BCTFRLHC_v6_ExpertGUI - Expert Acquisition Panel / Data after LUT - updated LUT for top mezzanine

In the average bunch intensity graphs in the *Expert Acquisition* panel, we plot the data after averaging them and before restoring their baseline or calibrate them. In addition we also plot the BLR components as they are calculated from the real-time action, in order to follow the BLR procedure and have a visual and immediate clue of the impact of our *Expert Settings* (figure 3-28). This is true only if the user chooses to plot one of the two plots (top/bottom mezzanine) since these components are specified per mezzanine.


BCTFRLHC_v6_ExpertGUI

VS (Noise Samples): 2

Turn Clk Delay: 3490

Phase Scan Settings:

phase scan action: DO NOTHING

bunch slot: 891

Phase Delay: 0

Top Mezzanine:

Noise Threshold: 100.0

Calibration Coefficient: 1.0

Calib Coef Offset: 0.0

Switch Threshold: 0.0

Bottom Mezzanine:

Noise Threshold: 50.0

Calibration Coefficient: 6.4992E7

Calib Coef Offset: 0.0

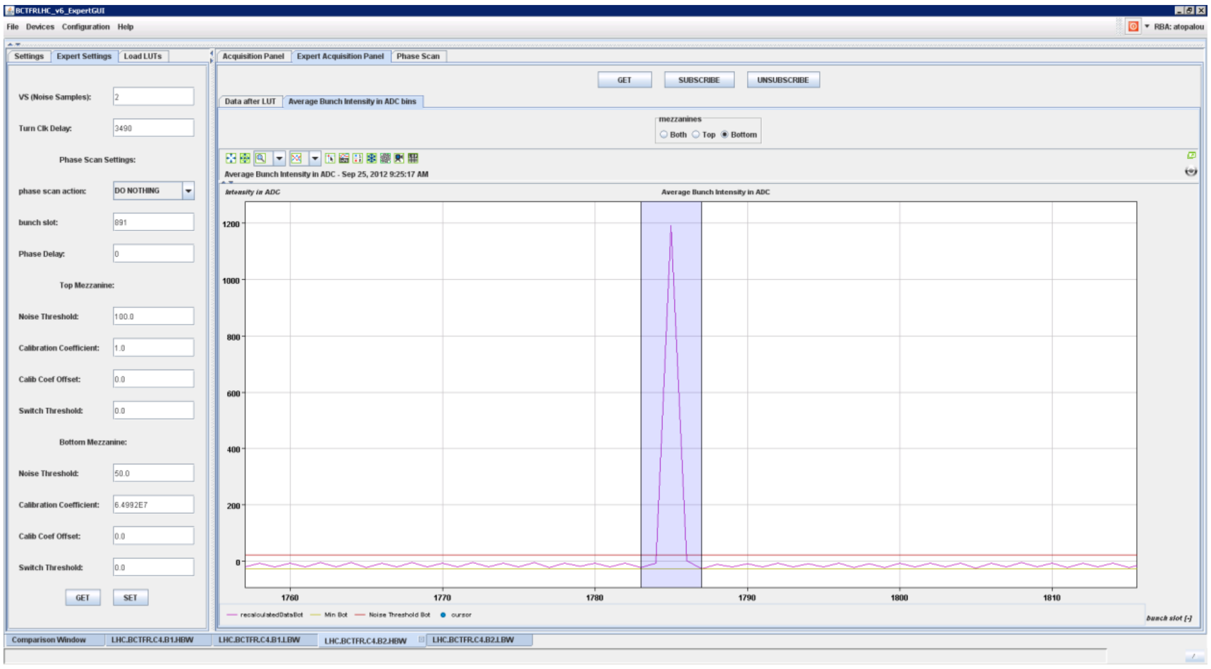
Switch Threshold: 0.0

mezzanines: Both Top [✓ Bottom]

Average Bunch Intensity in ADC - Sep 25, 2012 9:22:20 AM

Figure 3-28: BCTFRLHC_v6_Epxert Acquisition / Average Bunch Intensities in ADC bins - Expert Settings

In figure 3-29 a zoom of the same graph depicts the details of the BLR components for better understanding. In this figure the minimum value as it was calculated by *Acquire* real time action is visible with the yellow line as well the user setting TH with red. In addition the area that is considered to have useful signal is painted blue for better visualization.


BCTFRLHC_v6_ExpertGUI

VS (Noise Samples): 2

Turn Clk Delay: 3490

Phase Scan Settings:

phase scan action: DO NOTHING

bunch slot: 891

Phase Delay: 0

Top Mezzanine:

Noise Threshold: 100.0

Calibration Coefficient: 1.0

Calib Coef Offset: 0.0

Switch Threshold: 0.0

Bottom Mezzanine:

Noise Threshold: 50.0

Calibration Coefficient: 6.4992E7

Calib Coef Offset: 0.0

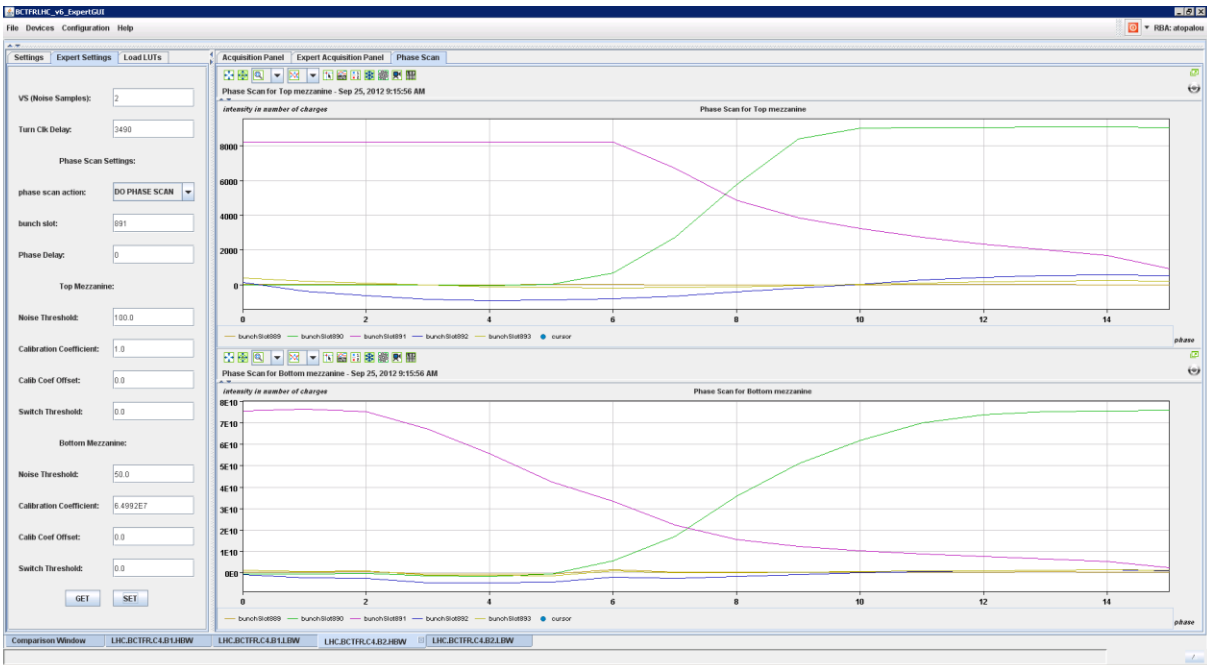
Switch Threshold: 0.0

mezzanines: Both Top [✓ Bottom]

Average Bunch Intensity in ADC - Sep 25, 2012 9:25:17 AM

Figure 3-29: BCTFRLHC_v6_EpxertGUI - Zoom at the Expert Acquisition panel / Average Bunch Intensity in ADC bins tab

Lastly, in figure 3-30 the phase scan procedure is depicted for the bunch slot that was found to have the maximum value.



VS (Noise Samples): 2

Turn Clk Delay: 3490

Phase Scan Settings:

phase scan action: DO PHASE SCAN

bunch slot: 891

Phase Delay: 0

Top Mezzanine:

Noise Threshold: 100.0

Calibration Coefficient: 1.0

Calib Coef Offset: 0.0

Switch Threshold: 0.0

Bottom Mezzanine:

Noise Threshold: 50.0

Calibration Coefficient: 6.4992E7

Calib Coef Offset: 0.0

Switch Threshold: 0.0

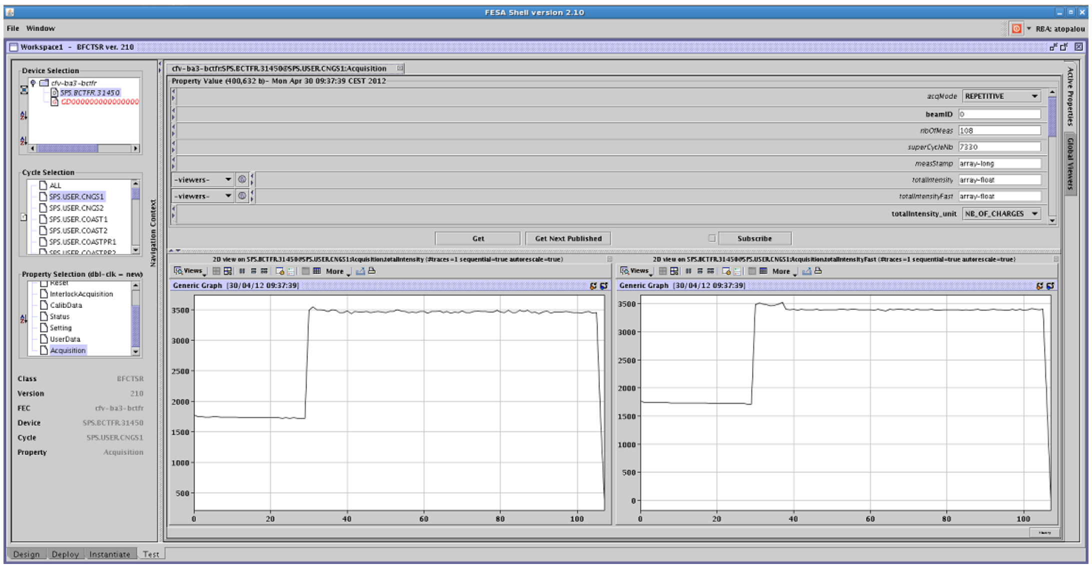
Figure 3-30: BCTFRLHC_v6_ExpertGUI - Phase Scan

4 Results

Having deployed our suggestions of the FBCT servers for both SPS and LHC rings is time to present and analyse the results of our implementations. This section is dedicated to that and is divided in two subsections, one per server. This is important since, the requirement for an operational FBCT system in the SPS ring was critical and hence, we couldn't wait in order to develop a unified system as it was first foreseen. Therefore, we developed this server first and in parallel we studied the ways – as it was described in the previous sections – in order to achieve the implementation of a unique FBCT measuring system.

4.1 SPS

The FESA class BFCTSR v210, our implementation of the server for the FBCT in SPS ring, was deployed and is operational since 22/05/2012. Until now no problems had occurred. On the contrary the CCC operators were happy to finally see this TURN_BY_TURN acquisition mode as well that the new implementation of the baseline restoration is working properly.



cfv-ba3-bctfr:SPS.BCTFR.31450@SPS.USER.CNGS1:Acquisition
Property Value (480,632 b)- Mon Apr 30 09:37:39 CEST 2012

Class	BFCTSR
Version	210
FEC	cfv-ba3-bctfr
Device	SPS.BCTFR.31450
Cycle	SPS.USER.CNGS1
Property	Acquisition

acqMode	REPETITIVE
beamID	0
nbOfMeas	108
superCycleNb	7330
measStamp	array-long
totalIntensity	array-float
totalIntensityFast	array-float
totalIntensity_unit	NB_OF_CHARGES

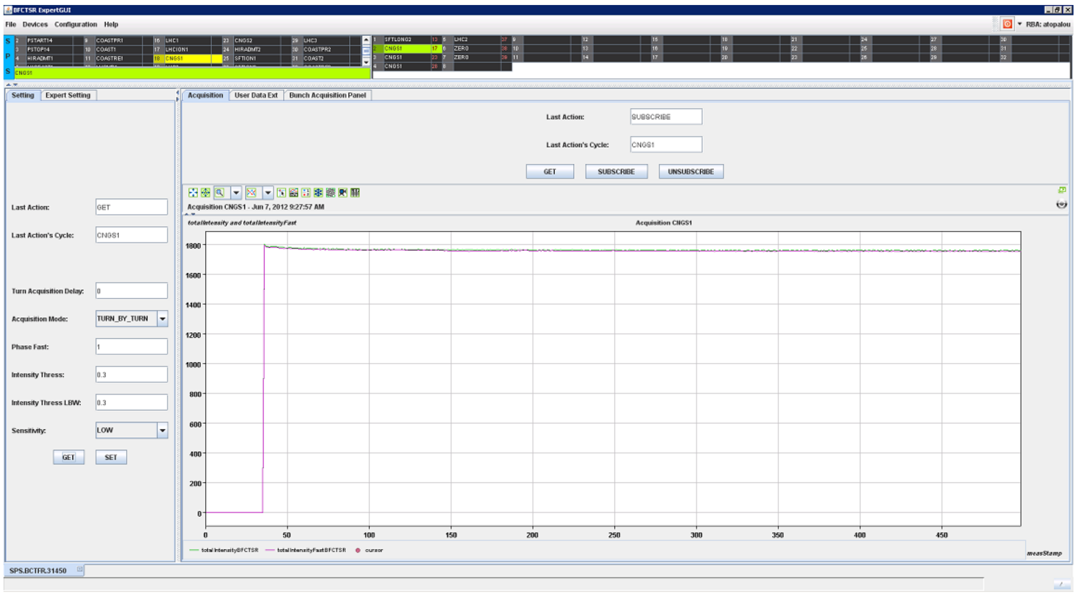
2D view on SPS.BCTFR.31450@SPS.USER.CNGS1:Acquisition.totalIntensity (#traces=1 sequential=true autorescale=true)

2D view on SPS.BCTFR.31450@SPS.USER.CNGS1:Acquisition.totalIntensityFast (#traces=1 sequential=true autorescale=true)

Figure 4-1: Total Intensity Measurement with FBCT for the SPS, CNGS1 cycle and REPETETIVE mode with the previous version of the server

Until now, the operators were only able to see the whole history of the beam's intensity during a cycle apart from the first injection, since the acquisition started the moment the beam was already present (see Figure 4-1). Having this history is useful but not if anyone wants to observe the behaviour of the beam's intensity at the injection time.

And that is what is renovating with our implementation, for the first time, the operators can see the intensity of the beam on the injection moment in great detail and thus they can easily calculate the additional intensity that actually took place during the injection. This is very important for the smooth operation of the SPS ring since several unpredicted behaviours of the beam can be detected early in the cycle allowing corrections to be made. What is more, and by specifying the delay of the execution of the acquisition in milliseconds, the operators (users) can actually choose how far they want to look in the cycle. In this way they can see a potential second, third, nth injection during a cycle in great detail.



Last Action: GET
Last Action's Cycle: CNGS1
Turn Acquisition Delay: 0
Acquisition Mode: TURN_BY_TURN
Phase Fast: 1
Intensity Thress: 0.3
Intensity Thress LBW: 0.3
Sensitivity: LOW
GET SET

Last Action: SUBSCRIBE
Last Action's Cycle: CNGS1
GET SUBSCRIBE UNSUBSCRIBE

Acquisition CNGS1 - Jun 7, 2012 9:27:57 AM

Figure 4-2: Total Intensity Measurement with the FBCT in the SPS, CNGS1 cycle, TURN_BY_TURN mode

Last but not least, the baseline restoration is now dynamically adjusted and thus is more precise and correct. This fact satisfied the users a lot, since they had many problems in the past with the reliability of the server and as a result they had to contact experts several times.

4.2 LHC

The large number of the client programs (Expert GUIs and logging) requesting data from the FBCT system C, requires an intermediate proxy software layer controlling the data flow between the server and the clients. In this way, low-level system load was minimized while the system's stability was gained.

In the following figures 4-3 and 4-4 a comparison of the beam's 2 total intensity as it was measured from system A and C is depicted. The first figure shows a low gain measurement and although the curves seem to follow each other quite nicely, the yellow one – system C – exhibits higher noise in terms of sigma than the other system.

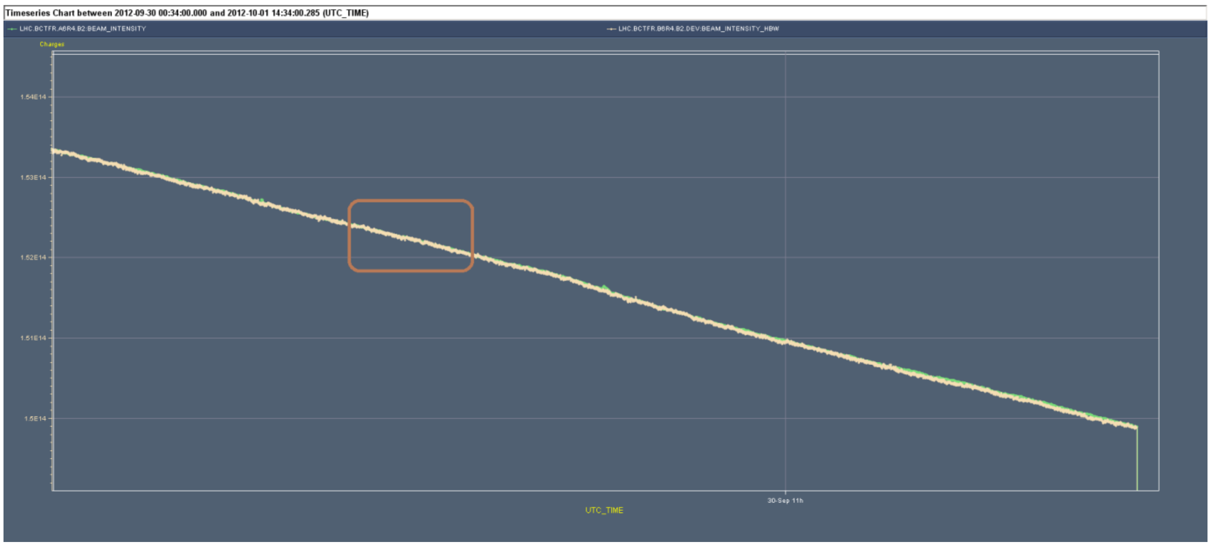


Figure 4-3: Beam's 2 low gain total intensity comparison among system A, B and C in the LHC

The next figure 4-4 is an enlargement of a small part of the previous measurement visible in figure 4-3 as a brown box.

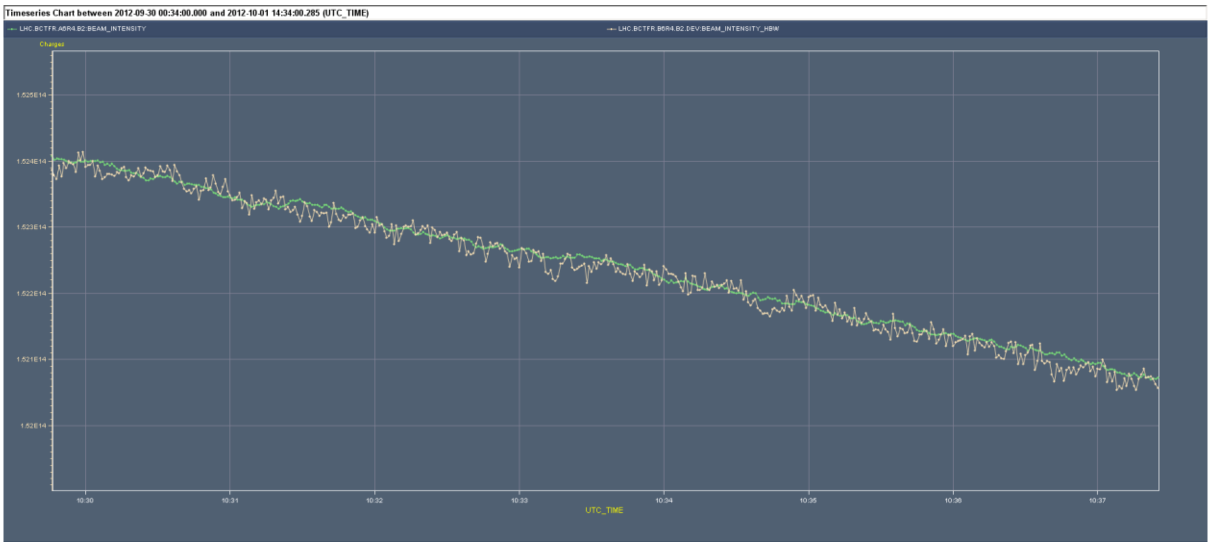


Figure 4-4: Beam's 2 high gain total intensity comparison among system A, B and C in the LHC

Already by the above figure, we observe that while the new system's measurement follows very nicely the already operational one, it is still noisier. This is mainly due to the number of turns both systems are acquiring data for and hence averaging over. It appears that having a 224 turn interval in order to suppress the white noise at system C, doesn't improve the resolution by much. The main difference as for the noise suppression comes from the number of averaging samples and therefore, system A provides smoother measurements than system C since the former acquires and averages over 900 turns whereas the latter over 25.

5 Conclusions and Future Work

After the presentation of our implementation of the two servers controlling the FBCTs in the SPS and LHC ring and analysing the results of these implementations, in this final section, we restate our observations, we propose future work and we conclude.

5.1 Conclusions

As we mentioned in chapter 2.1.6, out of the several technics that measure the beam's attributes, the FBCT measuring system is a very important one since it provides with great precision both bunch-by-bunch and total turn-by-turn intensity measurements. Additionally, it is the only system that can be absolutely calibrated although this is not the current state. In order to benefit the most out of this system though, several significant changes should be made and hence, new implementation solutions for the controlling software should be given.

5.1.1 SPS

In this direction the first contribution of this Thesis is the delivery of a complete software client-server scheme for the FBCT in the SPS ring. The server side of that scheme follows its predecessor's outline while benefiting from the new firmware's design and adding a complete new and renovating functionality – TURN_BY_TURN acquisition mode – that is proven very useful. In addition, it corrects former malfunctions as for the data treatment, making the server more dynamically adjustable to different use cases.

Furthermore, the client side of that scheme provides a different and more user-friendly interface for the server introducing new ways of presenting the data, such as 3D-graphs and 2D-graphs that can be easily scrolled at the same measurement, on the contrary of the graphical solutions that the previously used FESA interface provided.

5.1.2 LHC

An additional contribution of this Thesis is the study of another complete software client-server scheme for the FBCTs in the LHC ring that will be able to be used in any circulating beam installation in the future, including the already existing one of the SPS accelerator. The results of this study as they were presented in the previous section reveal that although this approach seems very promising, further work should be done in order to implement a unified FBCT measuring system. This matter will be explained in more details in chapter 5.2 but we can summarize here that only the averaging part of the data treatment was found insufficient and hence needs improving, whereas the LUTs, BLR and Gain Switching worked perfectly.

What is more, the client side of that scheme was found very helpful for the fast development of this system since it provided the direct comparison among the other systems of the same kind, in different ways. In addition and due to the lack of the calibration procedure of the system, the ability of setting directly the calibrating coefficients such that the measurements match the ones from the operational systems, improved the development speed as well.

5.2 Future Work

As an enhancement of this work we need to improve the averaging procedure of the data process in software. In order to do that, we will have to reduce dramatically the turn interval – even to 0 – since it doesn't contribute as much as expected to the noise suppression but impose a great delay in the acquisition time – a 224 turn interval impose approximately 20msec delay at every acquired turn. And this is actually the limiting factor to the number of turns acquired at our implementation since we agreed to perform a half second acquisition in order to have enough time to process the data, hence 25 acquired turns with a 224 turn interval.

On the other hand, performing a full bunch acquisition that would fill the memory – 294 consecutive turns lead to 1047816 acquired samples at almost 25msec – hits again the 1Hz restriction as it may take 25msec to make the acquisition but it takes almost 400msec to read the data from the DAB since there is only one VME bus.

As a result, we intend to move the averaging part of the data process to the hardware (FPGA) by changing the firmware again and adding a summing mode allowing us to perform full bunch acquisitions for a large number of consecutive turns removing the huge transfer delay in a sense that we will always be fetching 3564 values from memory. All the functionalities of the recently changed firmware – as they were analysed in chapter 2.2.2 – should remain unchanged if it is going to be used in other parts of the CERN's infrastructure such as linear accelerators and/or dump and transfer lines.

In addition, since we will be fetching averaged data from the DAB and not the integer acquired values, the parsing through the LUTs should be transformed to a linear approximation of LUT as it is described in chapter 3.1.2 of [21].

Last but not least, the proper calibration technique should be implemented in order to achieve the maximum of the FBCT measuring system performance.

Bibliography

[1] CERN, CERN in a Nutshell, http://public.web.cern.ch/public/en/About/About-en.html

[2] CERN, "CERN LHC: the guide", Geneva : CERN, 2006

[3] Belohrad, D, "Beam Charge Measurements", Geneva : CERN, 2011

[4] D. Belohrad, J. Gras, L. Jensen, R. Jones, M. Ludwig, P. Odier, J. Savioz, S. Thoulet, "Commissioning and First Performance of the LHC Beam Current Measurement Systems", IPAC'10, Kyoto, Japan, 2010

[5] D. Belohrad, L. Jensen, R. Jones, M. Ludwig, J. Savioz, "The LHC Fast BCT system: A comparison of Design Parameters with Initial Performance", BIW'10, Santa Fe, New Mexico, United States of America, 2010

[6] CERN, Design-Guidelines, logo-badge, http://design-guidelines.web.cern.ch/fr/logo-badge

[7] CERN, CERN Structure, http://www-dev.web.cern.ch/about/structure-cern

[8] CERN, Beams Department (BE), https://espace.cern.ch/be-dep/default.aspx

[9] CERN, Beams Department – Beam Instrumentation (BE-BI), https://espace.cern.ch/be-dep/BI/default.aspx

[10] CERN, Beams Department – Beam Instrumentation – Software Section (BE-BI-SW), http://project-beam-instr-sw.web.cern.ch/project-beam-instr-sw/Welcome.php

[11] CERN, The Accelerator Complex, http://public.web.cern.ch/public/en/Research/AccelComplex-en.html

[12] Ventura L, Gilardoni S, Migliorati M, Palumbo L, "Study of longitudinal multibunch instabilities for LHC-type beams at CERN Proton Synchrotron", Geneva : CERN, 2013

[13] the FESA team, "FESA Essentials", Geneva : CERN, 2004 (http://project-fesa.web.cern.ch/project-fesa/binaries/documents/FesaEssentialsBundle.pdf)

[14] D. Belohrad, R. Jones, M. Ludwig, J. Savioz, S. Thoulet, "Implementation of the Electronics Chain for the Bunch by Bunch Intensity Measurement Devices for the LHC", DIPAC'09, Basel, Switzerland, 2009

[15] D. Bishop, C. Boccard, E. Calvo-Giraldo, D. Cocq, L. Jensen, R. Jones, J. Savioz, G. Waters, "The LHC Orbit and Trajectory System", DIPAC'03, Mainz, Germany, 2003

[16] H. Jakob, L. Jensen, R. Jones, J. Savioz, "A 40MHz Bunch by Bunch Intensity Measurement for CERN SPS and LHC", DIPAC'03, Mainz, Germany, 2003

[17] G. Bohner, A. Falvard, J. Lecoq, P. Perret, C. Trouilleau, "Very front-end electronics for the LHCbpreshower", LHCb-2000-047, CERN, 2000

[18] Jean-Jacques Savioz "Engineering Specifications BOBR The Beam Synchronous Timing Receiver Interface For The Beam Observations", http://www.cern.ch/TTC/BOBRspec.pdf

[19] MEN Mikro Electronic GmbH A19/A20, http://www.men.de/products/01A020-.html#t=overview

[20] CERN, Scientific Linux CERN 5 (SLC5), http://linux.web.cern.ch/linux/scientific5/

[21] D. Belohrad, Technical Documentation "On the Fast Beam Intensity Measurements Algorithms and Correction Methods", http://svnweb.cern.ch/world/wsvn/fimdab/trunk/doc/on_the_beam_intensity_measurement_algorithms.pdf

[22] D. Belohrad, Technical Documentation "Digital Acquisition Firmware For The LHC Fast Beam Current Monitors, http://svnweb.cern.ch/world/wsvn/fimdab/trunk/doc/fimdab_technical_specification.pdf

[23] D. Belohrad, Development Documentation "Migration Guide", http://svnweb.cern.ch/world/wsvn/fimdab/trunk/doc/migration_guide.pdf