“	**Computing division**		
	Created: 27.6.2016	***Modified: 14.02.2017***	***Rev. No.: 1.2***

ALBA Em# Specification

Abstract

This document describes the user and designer specifications to design and use the ALBA Em#.

Prepared by:	*Checked by:*	*Approved by:*
Manuel Broseta	**Guifré Cuní** **Jose Ávila** **Oscar Matilla** **Sergi Blanch** **Xavier Serra**	

History of Changes			
Rev. No.	*Date*	*Pages*	*Description of changes*
1.0	27.06.2016		Initial Document
1.1	13.07.2016		First official review
1.2	14.02.2017		Software Versions and point 1 review Added new SCPI commads: SUPPly, ODAC, FVCTrl. Added AUTO as a CABO:RANG option Added point 7 Dispaly Table summary review Web server review (new pictures)

Index

1 Description

1.1 Introduction

High accuracy low current readout is an extensively demanded technique in 3rd generation synchrotrons. They comprise a need both for diagnostics such as for data acquisition in today's photon labs. In order to tackle the problem of measuring from various sources of different nature and magnitude synchronously, while remaining flexible at the same time, ALBA developed years ago a 4 independent channel electrometer, the ALBA Em, based on trans impedance amplifiers with high resolution ADC converters integrated and an Ethernet communication port.

The new ALBA Em# is a 4-channel electrometer evolution of that ALBA Em widely used in at ALBA from 2011. This new redesign intends to be more versatile and customizable equipment than the original model, providing a wide range of functionalities to fulfil unique and complex experiments by means of configuration changes instead of having specific instruments.

1.2 Software Versions

This document has been written with the following software versions, which are included in the first official version for production.

Module	Version	Description
LINUX	4.4.16	Own linux distribution generated with Yocto tool
ALIN	0.3.39	Main control software
SCPI	0.3.4.0	SCPI driver
FPGA bin	1.08	FPGA binary software

1.3 Electric Design

The Em# project uses a Simple PCI Express FMC Carrier (SPEC) board to command a FPGA Mezzanine Card (FMC), designed to work as a 4-channel ADC and transfer the processed data to a Single Board Computer (SBC), the Intel NUC DE3815, using its high-speed serial computer expansion bus (PCIe).

The SPEC, it is a cost-optimized design developed by the CERN under the Open Hardware License (OHL) that mainly works as FMC carrier. It is powered by Xilinx Spartan 6 FPGA for custom gateware designs using High-speed serial connectivity, a PCIe interface and an FMC slot.

This hardware configuration allows Em# implement its own FPGA control code, to manage a 4-channel ADC converter in the FMC card. The communication and data sharing with the main control software in the SBC are also part of the control code routines implemented in the FPGA. But apart form the SPEC and SBC boards, there

are other hardware boards included in this equipment. ***Figure 1*** shows the hardware diagram block of the Em# project.

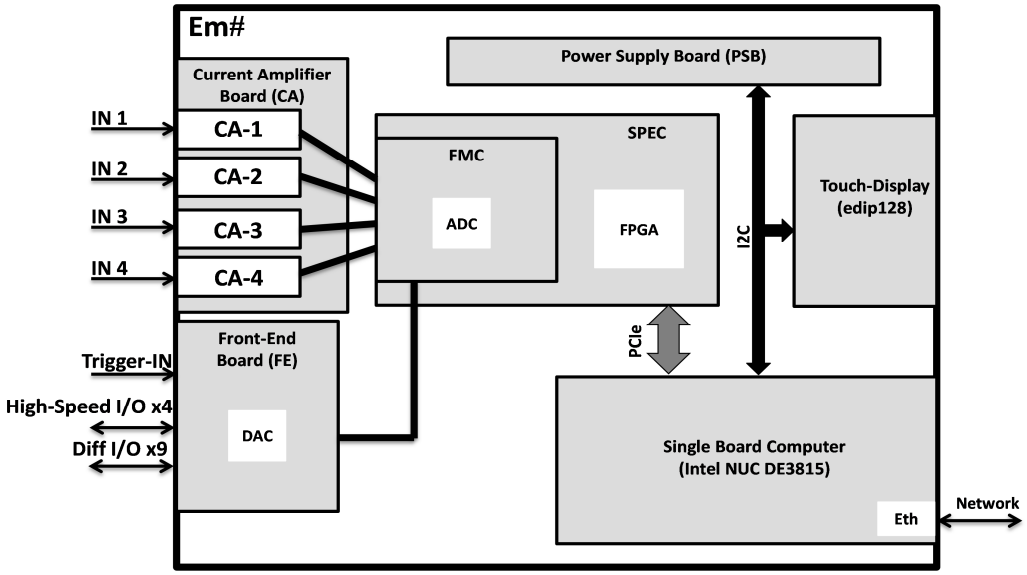

Figure 1. *Em# hardware diagram block.*

The Current Amplifier board (CA) contains the circuitry needed to communicate and control the 4 current amplifiers (CA-X). The Front-End board (FE) manages the trigger input and the different IO ports (4 High-Speed I/O ports and 9 Differential I/O ports). The Power Supply Board (PSB) supplies the equipment with different voltages needed by each module or board. A Touch-Screen (edip128) monitors the status and lets a general configuration of the equipment.

1.4 Software Design

The software development has been divided into three software projects, all together distributed in a single software package:

- The Linux OS
- The gateware (FPGA software)
- The main control software in the SBC (ALIN)

The Em# features complex embedded control software based on Linux OS. For that reason, a light Embedded Linux has been customized specially for this equipment, with only the necessary drivers to control the available hardware and with enough functionality to run the required applications. This Embedded Linux has been created using Yocto [4], an open source collaboration project that provides a set of recipes, tools and methods that allow building custom Linux-based systems to be embedded, regardless of the hardware architecture. Among the different recipes or set of recipes used for this project, these are the most significant ones:

- meta-e3815: Recipes to build the Embedded Linux OS distribution for the Intel Atom Processor E3815, used for this project.

- meta-spec: Main recipe used to compile and install the Spec Linux drivers in the corresponding Linux kernel selected
- meta-python: This is the set of recipes that provides python support.
- meta-alba: Contains the set of recipes to install the main control software (ALIN) and its drivers in the SBC

The gateware software [5] runs in the SPEC FPGA. It is written in VHDL and the design focuses on fast acquisition and data sharing between the different modules inside the FPGA as well as with the software running in the SBC. The binary, is reprogrammed in the FPGA every time the system boots. ***Figure 2*** shows the FPGA block diagram.

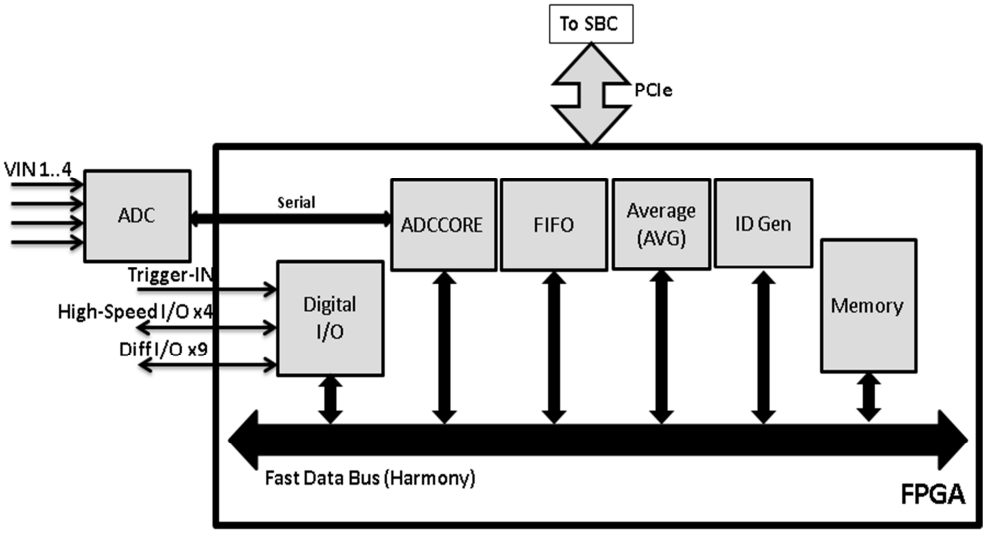

Figure 2. *FPGA block diagram.*

The Em# has been designed to acquire data at 400KSamples/second per channel. The main software is not fast enough to get the 4-channels data acquired at that sample rate via PCI bus. Therefore, the acquisition is carried out by the gateware software through a fast data acquisition bus, designed and implemented to share data between the different FPGA blocks. That fast data bus name is the Harmony Bus. Acquired data is stored in an FPGA memory block. The frames sent through this bus contains: the ID of the block which generates the frame, the data and the timestamp which indicates when the data was generated. The main software in the SBC configures the different acquisition types and reads the acquired data stored in the FPGA memory. Other slow and low priority data such as the information displayed in the touch-screen, are also transmitted through the PCI bus

In the SBC resides and runs the main software (ALIN) that has been designed aiming for high versatility in the application design and easy user control of the equipment. Versatility means that the software is easily adaptable to new features, just modifying the configuration of the FPGA, or to hardware changes. ALIN is multipurpose software customized to work as an electrometer. Regarding the easy user control, it offers both remote and local control interfaces. Remote control is available via telnet (using the SCPI protocol [6]) or via web through a webserver. Local control is available through navigation menus using the touch-screen display.

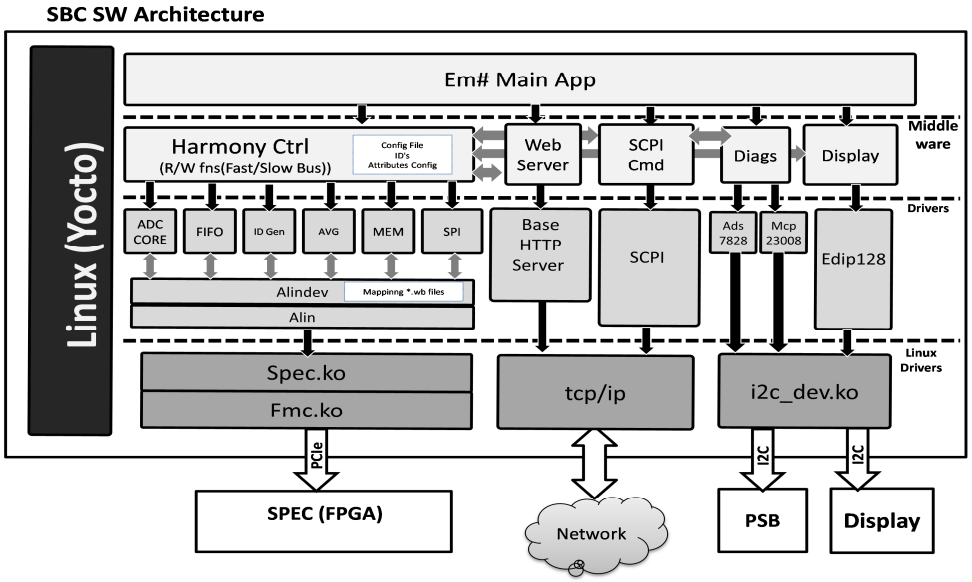

Figure 3. *SBC software architecture.*

Written in Python has the profits from the clean and straightforward syntax, while still performing well. The ***Figure 3*** shows the software architecture of ALIN divided in different functional layers. The communication between modules in different layers is always from top to bottom, except for the middleware layer where a cross communication is allowed to share data between the different modules. The different layers are, from top to bottom:

- Applications: The main application is in this layer. It allows the equipment control, remote or locally. It starts when the equipment boots and it is responsible to initialize and configure the middleware modules that contain the control and diagnostics functions.
- Middleware: In this layer resides the logic that makes this equipment work as an electrometer, or as any other equipment. It also provides the functionality to interact with the equipment through the touch-screen, or through a communications port.
- Drivers: Software modules to control physical or logical devices are in this layer. There is a driver for each FPGA module, a driver for each I2C device or a driver for the SCPI that contains the protocol to remotely control the equipment.
- Linux drivers: At the bottom, there are the Linux drivers needed for the Em# project. They have been previously compiled for the Embedded Linux distribution. Examples are the SPEC driver to communicate with the SPEC board via PCI or the I2C driver to control the display and the Power Supply Board (PSB).

In the left-bottom side of ***Figure 3***, the Spec and FMC Linux drivers are the kernel modules to control the SPEC and FMC cards via PCI bus. These Linux drivers are used by others like alin and alindev which implement read/write operations on the FPGA using the Self Describing Bus (SDB). This is a framework that helps to self-detect and manage the FPGA contents. It describes a series of structures used to provide metadata about the FPGA logic blocks, allowing the main software to automatically discover and configure them at runtime, via PCI bus. These SDB data structures are divided in records of 256 bytes size, where the first 64-byte are common between records and provide information about the FPGA blocks like the type of record, product, vendor, name, date, version and also the first-last address of the virtual memory space where the block data is located.

On top the alin drivers; there are specific drivers for each FPGA block. They provide functions to initialize write default values or read/write registers. These drivers use an external file that contains the register mapping. This file is auto generated using the same definition file which is used to define the FPGA block in the gateware software.

In the middleware layer, the Harmony Control middleware module uses these drivers to implement the electrometer Em# functionality. It configures the acquisition that will go through the Harmony bus in the FPGA, starts/stops the acquisition and process the acquired data in the FPGA memory. Acquired data and configuration parameters are shared to the rest of the middleware modules. External configuration of some predefined main software parameters, it is possible through a configuration file, which is loaded every time the system starts or after user request, by a command execution.

Remote control of the equipment is possible through a simple set of ASCII commands. These user control commands are implemented following a standard protocol for programmable instruments; the SCPI protocol. SCPI commands are ASCII textual strings that can contain one or more keywords, many of which take parameters. Responses to query commands are typically ASCII strings. Detailed explanation of SCPI is available in 2.***CONTROL*** section. The SCPI middleware module contains the list of control and configuration commands and their associated read/write call-back Em# functions. The SCPI driver, used by this middleware module is where the SCPI protocol is implemented.

The Em# also includes a web server, based on WebSocket protocol, for remote monitoring and overall control as detailed in section 6.***WEB SERVER***. The web offers a general equipment status, the current and voltages values read from the 4 channels, the configuration of the channels, the status and configuration of the acquisition, the last data acquired, the status and configuration of the 16 I/O ports, general diagnostics, etc.

It is also possible to do a more specific remote control of the FPGA through the tools available via SSH. There is a handful set of tools that help designers to check/configure the status of the FGPA, to get the SDB structures, write binary file or read/write to the FPGA devices or to configure some general parameters of the equipment

Local equipment control is possible through to the touch-screen display, detailed in…... The Display middleware module allows the user control through navigation menus, to locally configure or check the status of the equipment via the touch-screen display. The display is programmed by means of a protocol of high-level language graphic commands via I2C. The edip128 implements the communication protocol with the display.

ALIN also includes functions to control and self-detect its own diagnostic status. That is done in the Diagnostics middleware module. Check the harmony bus stability, self-detect the consumption of the power supplies in the Power Supply Board (PSB) board, among other tasks are some of the diagnostics examples done by this module. To control the PSB board is done using the Ads7828 (ADC) and Mcp23008 (Port-Expander) I2C drivers.

2 CONTROL

The ALBA Em# commands uses the ***Standard Commands for Programmable Instrumets (SCPI)*** as an control standard that specifies a common syntax, command structure, and data formats, used with all instruments.

The physical communications link is not defined by SCPI. While originally created for IEEE-488 (GPIB), it can also be used with RS-232, Ethernet, USB, VXIbus, HiSLIP, etc. In our development, we use ***Ethernet,*** and more specifically, we use ***telnet*** which is the application layer that provides a bidirectional interactive text-oriented communication facility using a virtual terminal connection used on internet or local area networks.

To connect with the electrometer execute telnet from Linux or Windows commad line and then open the communications port with the Alba Em# as follows:

telnet> open <*electrometers_hostname_or_ipaddress*> 5025

Remember that a telnet communication is closed pressing **CTRL+]** , write **quit** and then press ENTER to close the connection.

2.1 *SCPI Command Syntax*

SCPI commands are ASCII textual strings, which are sent to the instrument over the physical layer. Commands are a series of one or more keywords, many of which take parameters. In the specification, keywords are written in capital letters and use to be the first four letters of a command. The entire keyword can be used, or it can be abbreviated to just the uppercase portion. Responses to query commands are typically ASCII strings. However, for bulk data, binary formats can be used.

SCPI commands to an instrument may either perform a set operation (e.g. switching a power supply on) or a query operation (e.g. reading a voltage). Queries are issued to an instrument by appending a question-mark to the end of a command. Some commands can be used for both setting and querying an instrument. For example:

:MEASure::VOLTage?

*CAL?

Similar commands are grouped into a hierarchy or "tree" structure, while specific sub-commands within the hierarchy are nested with a colon (:) character.

Some commands require an additional argument. Arguments are given after the command, and are separated by a space. For example, the command to set the trigger mode:

"TRIGger:MODe SOFTware

Here, the word "NORMal" is used as the argument to the "TRIGger:MODe" command.

Multiple commands can be issued to an instrument in a single string. They are made of simple commands separated by a semicolon character (;). For example, the command to "Measure a DC voltage then measure an AC current" would be issued as MEASure:VOLTage:DC?;:MEASure:CURRent:AC?.

Simple commands which start with a colon (:) are interpreted with respect to the root of the command tree. Otherwise, they refer implicitly to the last node of the previous command (unless they already begin with an asterisk).

The command syntax shows some characters in a mixture of upper and lower case. Abbreviating the command to only sending the upper case has the same meaning as sending the upper and lower case command.

3 COMMANDS

This section describes the list of commands implemented in the ALBA Em# using SCPI. As pointed in section 2, these commands are sent using the *Telnet protocol through port 5025.*

They are group by its purpose, general commands wich affect to overall EM# or channel specific commands which are related to a particular channel. In this development we are using four independent channels named as CHAN01, CHAN02, CHAN03 and CHAN04.

3.1 GENERAL COMMANDS

These general commands are used to configure and check the general status of the instrument or common modules to all channels, like the frontend board or the carrier board.

They are used to get information about the status of the instrument or perform global actions that affect to the complete instrument

They aren't group inside any component, so they can be directly executed.

3.1.1 *IDN

This is an special and ***mandatory*** command that returns the identification information of tis instrument. IT returns information about the equipment name, the serial number and the software version.

Usage:	*IDN?
Type:	READ_ONLY
Returns:	<string> that contains general equipment information.
Ex.:	**IDN?* *ALBASYNCHROTRON,Electrometer2,000000000,0.2.2*

Technical description

This command returns the information about Manufacurer, Instrument name, Serial Number and software version. The first three items are got from the configuration file which is used to configure the instrument. These parameters can be modified during production without changing the software.

3.1.2 *MAC

This is another special and ***mandatory*** command that returns the mac address of this equipment

Usage:	*MAC?
Type:	READ_ONLY
Returns:	<string> that contains mac address.

Ex.:	**MAC?* c0:3f:d5:65:73:89

Technical description

This instrument runs under its own Linux distribution. To get the MAC address we use standard Linux commands.

*3.1.3 *HLP*

This commands shows short information about the list of available commands.

Usage:	*HLP?
Type:	READ_ONLY
Returns:	<string> that contains the list of available commands
Ex.:	**HLP?* *IDN* *MAC* *RST* *FWVER* *CHANnelNN:INSCURRENT* *CHANnelNN:AVGCURRENT* *CHANnelNN:CABO:TEMP* *CHANnelNN:CABO:INIT* *….*

*3.1.4 *RST*

This command performs a general reset of the instrument. It stops the application that controls the FPGA and executes a reset of the OS.

Usage:	*RST
Type:	WRITE_ONLY
Returns:	None
Ex.:	*RST

Technical description

This instrument runs under its own Linux distribution. When a reset is executed, the main application which is running to control the equipment is ended, closing the harmony application, which is the one that controls the electrometer logic, the display and killing all possible open control threads and the executes the Linux command to do a system reset:

 # reboot -f -t 0

3.1.5 *SHT

This command shutdowns the instrument remotely. It stops the application that controls the FPGA and executes a shutdown of the OS.

Usage:	*SHT
Type:	WRITE_ONLY
Returns:	None
Ex.:	*SHT

Technical description

Same as the reset, when this command is launched the main application which is running to control the equipment is ended, , closing the harmony application, which is the one that controls the electrometer logic, the display y and killing all possible open control threads and the executes the Linux command to do a system shutdows:

 # shutdown now

3.1.6 RECOnfigure

There is a configuration file used for designers to configure this instrument (see Appendix A for file details). This config file is read during the startup and configures the software to work with the parameters defined in it. This command is used to reconfigure again the complete project without restart.

Usage:	RECO <value> Or RECOnfigure <value>
Type:	WRITE_ONLY
Set value:	Fixed to True (not used but required)
Returns:	nan
Ex.:	*RECO True*

Technical description

This command reapplies all parameters declared in the config file, shuch as debug levels, debug enable, i2c addresses, etc..without restarting the main control application. For more details about the config file contents, refer to Appendix A.

3.1.7 FWVErsion

It returns the version of the firmware which is reuning in the FPGA.

Usage:	FWVE? *or* FWVErsion?
Type:	READ_ONLY
Returns:	<string> that contains the FW version
Ex.:	*FWVE?* 00.01

Technical description

This returns the FPGA software version, written in the FPGA though the SDB structure.

FPGA blocks	SDB structure: SDB INTEGRATION RECORD
Registers	Version

3.2 DIAGNOSTICS COMMANDS

These commands are used to check the diagnostics status of the hardware. They are grouped under the ***DIAGnostics*** component. So to execute these commands you need to enter the component first, then follow by ':' and then tne command.

By default, if no additional command is send, it executes the ***VSENse*** command.

3.2.1 CBTEmperature

This diagnostics returns information about the temperature of the carrier board.

Usage:	DIAG:CBTE? *Or* DIAGnostics:CBTEmperature?
Type:	READ_ONLY
Returns:	<string> with the temperature of the carrier board
Ex.:	*DIAG:CBTE?* 28.8125

Technical description

SPI bus control the low speed HW included in the instrument and placed in the Front-End Board (FEB), the current amplifiers carrier board or the four current amplifiers. There are up to 25 devices connected to the SPI. This SPI bus is controlled by the FPGA. In order to have multiple lines to control these devices, a 3-to-8 active low decoding system as been implemented to control the chip select lines of those devices and to select the appropriate one when writing to a device. So to read a particular device connected to the SPI bus, it is important

to select to the appropriate Chip select and then use the register to transmit or receive data to the selected device.

For more details see ALBA Em# Configuration SPI specification document.

FPGA blocks	SPI
Registers	*ctl_spi_addr:* 0x14 ➔ Chip Select for the Temperature Sensor of CA Carrier Board Other register used to transmit/receive data though SPI: *ctl_tx, ctl_tx_data, ctl_tx_data_length, ctl_rx, ctl_rx_data_length, ctl_start, ctl_rx_data*
Formula	Read_value*0.0625

3.2.2 FETEmperature

This diagnostics returns information about the temperature of the front end board.

Usage:	DIAG:FETE? *Or* DIAGnostics:FETEmperature?
Type:	READ_ONLY
Returns:	<string> with the temperature of the front end board
Ex.:	*DIAG:FETE?* 28.8125

Technical description

Same as explained in technical description of 3.2.1. CBTEmperature, the Front End board (FE) is also connected to the SPI bus. So any communication to this board is done through the SPI module in the FPGA by selecting the proper chip select.

FPGA blocks	SPI
Registers	*ctl_spi_addr:* 0x1b ➔ Chip Select for the Temperature Sensor of the Front End Board Other register used to transmit/receive data though SPI: *ctl_tx, ctl_tx_data, ctl_tx_data_length, ctl_rx, ctl_rx_data_length, ctl_start, ctl_rx_data*
Formula	Read_value*0.0625

3.2.3 VSENse

This diagnostics returns information about the sensed voltages for the different voltages supplies.

Usage:	DIAG:VSEN? *Or* DIAGnostics:VSENse?
Type:	READ_ONLY
Returns:	<string> that contains a joined list with the different voltage supplied and their sensed values

Ex.:	*DIAG:VSEN?* [['VSENSE_I_ISO', 0.033183750108771316], ['VSENSE_I_SPEC', 0.7891845703125], ['VSENSE_I_AUX', 0.0097868085683297159], ['VSENSE_I_VS', 0.058523337112464248], ['VSENSE_I_VCC', 0.16724363772455089]]

Technical description

The power supply board (PSB) is directly controlled by the NUC through I2C. To monitor the status of the voltages controlled by this PSB board, it is mount an ADC (ADS7828), which returns the voltage values sensed for the different voltages supplied. These voltages are the isolated power supply (VSENSE_I_ISO), the VCC power supply (VSENSE_I_VCC), the VS power supply (VSENSE_I_VS), the auxiliary power supply (VSENSE_I_AUX) and the 12V SPEC power supply (VSENSE_I_SPEC).

For more details see ALBA Em# Configuration I2C specification document.

HW blocks	ADC7828 (I2C: 0x48 7-bits address)
Registers	0x84: VSENSE_I_ISO 0x94: VSENSE_I_VS 0xA4: VSENSE_I_SPEC 0xC4: VSENSE_I_VCC 0xD4: VSENSE_I_AUX
Formula	Read_value*0.0625

3.3 TRIGGER COMMANDS

These commands are used to configure or check the trigger for an acquisition. They are grouped under the ***TRIGger*** component. So to execute these commands you need to enter the component first, then follow by ':' and then the command.

By default, if no additional parameter or command is send, it executes the ***STATe*** command

3.3.1 DELAy

This command is used to configure the delay in miliseconds of the acquisition after the trigger.

This can be read always, but can only be modified when the acquisition state is not ACQUIRING.

Usage:	TRIG:DELA <value> *or* TRIGger:DELAy <value> *Or* TRIG:DELA? *or* TRIGger:DELAy?
Type:	READ_WRITE
Set value:	Delay in milliseconds
Returns:	<string> with the corresponding trigger delay
Ex.:	*TRIG:DELA 1000* 1000 *TRIG:DELA?*

	1000

Technical description

Trigger commands are not able to be modified when an acquisition is running; they are just allowed to be modified before starting a new acquisition. At the same time, these commands don't change any value in the FPGA when they are executed, they just store the value in a local variable, in this case ***trigger_delay*** and the delay value will be really applied in the FPGA registers when a new acquisition starts.

3.3.2 INPUt

This command is used to configure the input used for the trigger. The parameter is integer value which corresponds to the trigger input listed below (see "Set Value")

This can be read always, but can only be modified when the acquisition state is not ACQUIRING.

Usage:	TRIG:INPU <value> *or* TRIGger:INPUt <value> *Or* TRIG:INPU? *or* TRIGger:INPUt?
Type:	READ_WRITE
Set value:	0: DIO_1 1: DIO_2 2: DIO_3 3: DIO_4 4: DIFF_IO_1 5: DIFF_IO_2 6: DIFF_IO_3 7: DIFF_IO_4 8: DIFF_IO_5 9: DIFF_IO_6 10: DIFF_IO_7 11: DIFF_IO_8 12: DIFF_IO_9
Returns:	<string> with the corresponding trigger input name
Ex.:	*TRIG:INPU 4* DIFF_IO_1 *TRIG:INPUt?* DIO_4

Technical description

Trigger commands are not able to be modified when an acquisition is running; they are just allowed to be modified before starting a new acquisition. At the same time, these commands don't change any value in the FPGA when they are executed, they just store the value in a local variable, in this case ***trigger_input*** and the input will be really set in the FPGA registers when a new acquisition starts.

3.3.3 MODE

This command sets the trigger between the two available types SOFTWARE or HARDWARE. The parameter is integer value which corresponds to the trigger input listed below (see "Set Value")

This can be read always, but can only be modified when the acquisition state is not ACQUIRING.

Usage:	TRIG:MODE <value> o*r* TRIGger:MODE <value> *Or* TRIG:MODE? o*r* TRIGger:MODE?
Type:	READ_WRITE
Set value:	0: SOFTWARE 1: HARDWARE
Returns:	<string> with the corresponding trigger mode name
Ex.:	*TRIG:MODE 1* HARDWARE *TRIG:MODE?* HARDWARE

Technical description

Trigger commands are not able to be modified when an acquisition is running; they are just allowed to be modified before starting a new acquisition. At the same time, these commands don't change any value in the FPGA when they are executed, they just store the value in a local variable, in this case ***trigger_mode*** and the mode will be really set in the FPGA registers when a new acquisition starts.

3.3.4 POLArity

This command sets the polarity of the trigger at the input between "rising" (level change from 0V to 5V) or "falling" (level change form 5V to 0V). The parameter is integer value which corresponds to the trigger input listed below (see "Set Value")

This can be read always, but can only be modified when the acquisition state is not ACQUIRING.

Usage:	TRIG:POLA <value> o*r* TRIGger:POLArity <value> *Or* TRIG:POLA? o*r* TRIGger:POLArity?
Type:	READ_WRITE
Set value:	0: FALLING 1: RISING
Returns:	<string> with the corresponding trigger polarity name
Ex.:	*TRIG:POLA 1* RISING *TRIG:POLArity?* RISING

Technical description

Trigger commands are not able to be modified when an acquisition is running; they are just allowed to be modified before starting a new acquisition. At the same time, these commands don't change any value in the FPGA when they are executed, they just store the value in a local variable, in this case ***trigger_polarity*** and the polarity will be really set in the FPGA registers when a new acquisition starts.

3.3.5 SWSEt

This command is used to force an acquisition only when the trigger mode is set to SOFTWARE and the acquisition state is ACQUIRING

Usage:	TRIG:SWSE <value> *or* TRIGger:SWSEt <value>
Type:	WRITE_ONLY
Set value:	Fixed to True (not used but required)
Returns:	Nan
Ex.:	*TRIG:SWSE True* nan

Technical description

This command lets to trigger an acquisition, when the acquisition is running and the trigger mode is set to SOFTWARE. In any other case this command has no sense and will return error.

The SW trigger is generated using the ID_Generator FPGA block. When a SW trigger is requested by the user, an ID_GEN ID message is generated in the harmony bus (with no data and timestamp set by the FPGA))and this, triggers the FPGA memory to store the corresponding data, that can come form the AVERAGE module or the FIFO or the ADC, depending on the configuration set. To allow this, when the acquisition starts, the FPGA Memory has to be properly configured, to listen the ID GEN ID and store the selected data when this message is sent.

FPGA blocks	ID_GEN	
Registers	*DATA_GEN_DATA*	0x00 ➔No data send
	ID_GEN_CTL_CLKE	0x01 ➔ Message timestamp internally generated
	TS_ID_ID_OUT	**_IDGEN_ID** ➔ Fixed value for the ID of the ID Gen module. ID can be configured in the Config file
	ID_GEN_CTL_MTRIG:	0x01 ➔ Set to 1 to send Data Message

3.3.6 STATe

This command returns the status of the trigger, it means mode, polarity, input and delay configuration

Usage:	TRIG? *or* TRIG:STAT? *or* TRIGger:STATe?
Type:	READ_ONLY

Returns:	<string> that contains a joined list with general information about the trigger. Trigger mode, delay, polarity and input
Ex.:	*TRIG?* [['MODE', 'SOFTWARE'], ['POLARITY', 'FAILING'], ['DELAY', 1000], ['INPUT', 'DIO_4']]

Technical description

This command is a combination of the trigger settings ***trigger_delay***, ***trigger_input***, ***trigger_mode*** and ***trigger_polarity*** and its value in RAM together in a list. This command doesn't checks any value in the FPGA.

3.4 ACQUISITION COMMANDS

These commands are used to configure or check the acquisition. They are grouped under the ***ACQUisition*** component. So to execute these commands you need to enter the component first, then follow by ':' and then the command.

By default, if no additional parameter or command is send, it executes the ***STATe*** command

3.4.1 FILTer

This command configures the acquisition filter, which sets current amplifier prefilter and postfilter, and it is applied to the four channel amplifiers at the same time.

It is measured in Hz and the parameter is integer value which corresponds to the filter value listed below (see "Set Value")

This can be read always, but can only be modified when the acquisition state is not ACQUIRING.

The filter value is only applied to the current amplifiers when the acquisition starts.

Usage:	ACQU:FILT <value> o*r* ACQUisition:FILTer <value> *Or* ACQU:FILT? o*r* ACQUisition:FILTer?
Type:	READ_WRITE
Set value:	0: 3200 Hz 1: 100 Hz 2: 10 Hz 3: 1 Hz 4: 0.5Hz
Returns:	<string> that contains the common value set to the four acquisition channels for the filter.
Ex.:	*ACQU:FILT?* 3200 *ACQU:FILT 1* 100

Technical description

Acquisition filters can be modified before starting an acquisition, but once it is started they can not be changed. It configures the different filters available in the current amplifier boards (CA), it means pre filter and post filter for the four CA boards available. These different filters can be controlled independently among them and among the different channels, but this command is just a common control for those filters, to apply the same setting for the four acquisition channels.

CA filters are controlled through SPI block in the FPGA, so basically to configure them the proper chip select has to be selected before sending the data to each filter and channel.

For more details see section *"1.3.1 Current Amplifiers configuration/monitoring through SPI devices"* of the specification document ALBA Em# Configuration SPI

FPGA blocks	SPI
Registers	*ctl_spi_addr:* 0x01 ➔ Chip Select for PostFilter the Port Expander 1 of CA #1 *ctl_spi_addr:* 0x02 ➔ Chip Select for PreFilter the Port Expander 2 of CA #1 *ctl_spi_addr:* 0x06 ➔ Chip Select for PostFilter the Port Expander 1 of CA #2 *ctl_spi_addr:* 0x07 ➔ Chip Select for PreFilter the Port Expander 2 of CA #2 *ctl_spi_addr:* *0x0b ➔ Chip Select for PostFilter the Port Expander 1 of CA #3* *ctl_spi_addr:* *0x0c ➔ Chip Select for PreFilter the Port Expander 2 of CA #3* *ctl_spi_addr:* *0x10 ➔ Chip Select for PostFilter the Port Expander 2 of CA #4* *ctl_spi_addr:* *0x11 ➔ Chip Select for PreFilter the Port Expander 2 of CA #4* Other register used to transmit/receive data though SPI: *ctl_tx, ctl_tx_data, ctl_tx_data_length, ctl_rx, ctl_rx_data_length, ctl_start, ctl_rx_data*

3.4.2 MEAS

This command returns a string that contains a joined list with the four channels current data acquired during the last acquisition. The data acquired is reset when a new acquisition is started.

It can be read always, independently of the state

Usage:	ACQU:MEAS? *Or* ACQUisition:MEAS?
Type:	READ_ONLY
Returns:	<string> that contains a joined list with the acquired data for the four channels
Ex.:	*ACQU:MEAS?* [['CHAN01', '[-1.453476, -1.453476]'], ['CHAN02', '[2.063828, 2.063828]'], ['CHAN03', '[0.000153, 0.000153]'], ['CHAN04', '[2.065964, 2.065887]']]

Technical description

Acquisition data is stored in the FPGA memory. The main application is checking continuously the FPGA memory in an independent thread and when new data is stored, it is read and located in its corresponding buffer in RAM. When this command is executed, it returns a list with the data sored in the acquisition buffer for each channel. No access to the FPGA is required, it just returns the local copy in RAM of the data acquired for the four channels.

3.4.3 NDATa

This command returns the number of triggers detected (hardware or software)

It can be read always, independently of the state

Usage:	ACQU:NDAT? *Or* ACQUisition:NDATa?
Type:	READ_ONLY
Returns:	<string> that contains the number of triggers
Ex.:	*ACQU:NDAT?* 2

Technical description

Returns the number of triggers, hardware or software, detected. When a new acquisition starts, we use

The SW trigger is generated using the ID_Generator FPGA block. When a SW trigger is requested by the user, an ID_GEN ID message is generated in the harmony bus (with no data and timestamp set by the FPGA))and this, triggers the FPGA memory to store the corresponding data, that can come form the AVERAGE module or the FIFO or the ADC, depending on the configuration set. To allow this, when the acquisition starts, the FPGA Memory has to be properly configured, to listen the ID GEN ID and store the selected data when this message is sent.

FPGA blocks	ID_GEN	
Registers	*DATA_GEN_DATA*	0x00 ➔No data send
	ID_GEN_CTL_CLKE	0x01 ➔ Message timestamp internally generated
	TS_ID_ID_OUT	**_IDGEN_ID** ➔ Fixed value for the ID of the ID Gen module. ID can be configured in the Config file
	ID_GEN_CTL_MTRIG:	0x01 ➔ Set to 1 to send Data Message

3.4.4 RANGe

This command sets the acquisition range, which results of combining the trans impedance and voltage gains, and it is applied to the four current amplifier channels at the same time. The maximum range that tis instrument can read is +/- 10V, so to convert from voltage to current the formula to apply is as follows:

current = (adc_voltage_read_value * range)/10V

It is measured in mA and the parameter is integer value which corresponds to the filter value listed below (see “Set Value”).

This can be read always, but can only be modified when the acquisition state is not ACQUIRING.

The range value is only applied to the current amplifiers when the acquisition starts.

Usage:	ACQU:RANG <value> *or* ACQUisition:RANGe <value> *Or* ACQU:RANG? *or* ACQUisition:RANGe?
Type:	READ_WRITE
Set value:	0: 1 1: 0.1

	2: 0.01 3: 0.001 4: 0.0001 5: 0.00001 6: 0.000001 7: 0.0000001 8: AUTO
Returns:	<string> that contains the common value set to the four acquisition channels for the filter.
Ex.:	*ACQU:RANG?* 1 *ACQU:RANG 5* 0.000001

Technical description

Acquisition range defines the maximum and minimum values in which the instrument works. The range can be modified before starting an acquisition and it configures the different gains to be applied in the current amplifiers (CA). These gains are the trans-impedance gain and the voltage gain of each amplifier. Adjustment of these amplifiers can be independent, but before starting an acquisition this command is used to set the same range settings to the 4-channels at the same time.

CA range are controlled through SPI block in the FPGA, so basically to configure them the proper chip select has to be selected before sending the data to each filter and channel.

For more details see section *"1.3.1 Current Amplifiers configuration/monitoring through SPI devices"* of the specification document ALBA Em# Configuration SPI

FPGA blocks	SPI
Registers	*ctl_spi_addr:* 0x01 ➔ Chip Select for TIGain the Port Expander 1 of CA #1 *ctl_spi_addr:* 0x02 ➔ Chip Select for VGain the Port Expander 2 of CA #1 *ctl_spi_addr:* 0x06 ➔ Chip Select for TIGain the Port Expander 1 of CA #2 *ctl_spi_addr:* 0x07 ➔ Chip Select for VGain the Port Expander 2 of CA #2 *ctl_spi_addr:* *0x0b ➔ Chip Select for* TIGain *the Port Expander 1 of CA #3* *ctl_spi_addr:* *0x0c ➔ Chip Select for* VGain *the Port Expander 2 of CA #3* *ctl_spi_addr:* *0x10 ➔ Chip Select for* TIGain *the Port Expander 2 of CA #4* *ctl_spi_addr:* *0x11 ➔ Chip Select for* VGain *the Port Expander 2 of CA #4* Other register used to transmit/receive data though SPI: *ctl_tx, ctl_tx_data, ctl_tx_data_length, ctl_rx, ctl_rx_data_length, ctl_start, ctl_rx_data*

3.4.5 STARt

This command is used to start an acquisition. It uses previous data set to configure the FPGA blocks and starts the thread that reads the FPGA memory.

This command changes the acquisition state from ON to ACQUIRING. So it state is not ON, a new acquisition cannot be started.

Usage:	ACQU:STAR <value> *Or* ACQUisition:STARt <value>

Type:	WRITE_ONLY
Set value:	Fixed to True (not used but required)
Returns:	None
Ex.:	*ACQU:STAR True* None

Technical description

All the logic and configuration of the FPGA is done when this command is sent. It uses previously defined trigger and acquisition settings to properly setup the FPGA blocks according the acquisition type set. The different settings to be used when acquisition starts are:

- ***trigger_delay, trigger_input, trigger_mode, trigger_polarity, acquisition_time***

Apart from these settings defined by the user, there are other settings calculated by the main software which result is used in , other values are also calculated like:

- **acquisition_type** that defines the configuration table to use in the FPGA
- **fifo_size** which is calculated using the acquisition time and the minimum time need to acquire one sample
- **mask_value** depending on the acquisition type, a different mask will be applied to save data in the FPGA memory
- **delay_time** the delay to apply has to be calculates in units of 1024*time nanoseconds

The **acquision_type** uses mainly the trigger mode and the acquisition time, to define the table of registers and values to apply in the FPGA registers. The trigger mode will define the Id's that will be sent by the harmony bus and will trigger the FPGA memory to store data. Meanwhile the acquisition time, defines the configuration of the FIFO and also, the way in which the main software acts when reading the data stored in the FPGA memory. Due to technical limitations, the FIFO can only store a maximum of 1024 samples per channel. These ALBA Em# is working with an acquisition of 200KS/s and an oversampling of 64, which means an acquisition time for one sample is 320microseconds. With this numbers, the maximum acquisition time we can get using the FPGA FIFO is 320 milliseconds. If the acquisition time set is bigger than this number, it has to be combined the result of several acquisitions with the FIFO to its maximum size.

With this, we have up to 4 different types of acquisition configurations:

- Trigger software with an acquisition time lower than FIFO limit (configuration data in Appendix B.1)
- Trigger software with an acquisition time bigger than FIFO limit (configuration data in Appendix B.2)
- Trigger hardware with an acquisition time lower than FIFO limit (configuration data in Appendix B.3)
- Trigger hardware with an acquisition time bigger than FIFO limit (configuration data in Appendix B.4)

The **fifo_size** is calculated in any case, depending on the acquisition mode. If acquisition time is lower than FIFO maximum size, then its value results form the division between the ***acquisition_time*** set and the sample acquisition time (which is 320microseconds). If the acquisition time is bigger then the ***acquisition_time*** is divided in equal **fifo_size** parts. The main software reads the data stored in the FPGA memory and also counts the number of parts acquired. When the number of partial counts it reaches the expected number, the acquisition is stopped and the resulting data is calculated as an average of the partial acquisitions acquired.

3.4.6 STATe

This command returns the status of the acquisition. The possible allowed states in this instrument are:

- INIT: The system is being initialized
- ON: System read to accept commands or acquisition finished
- ACQUIRING: Acquisition in progress
- FAULT: Problems detected in the system. Acquisition stopped.

Usage:	ACQU:STAT? *or* ACQUisition:STATe? *or* ACQU?
Type:	READ_ONLY
Returns:	<string> that contains the state of the instrument
Ex.:	*ACQU?* STATE_ON

Technical description

This command only return the internal state of the main software, regarding the acquisition. Thea are 4 different states that can be returned:

- INIT: The main software is still under initialization
- ON: System already initialized and ready to accept commands, to setup an acquisition, etc..
- ACQUIRING: An acquisition has been launched and it is in progress
- FAULT: Some miss function has happened and the normal operation cannot be carried out.

3.4.7 STOP

This command stops a previously started acquisition. It finishes the thread which is reading continuously the FPGA memory and configures the FPGA to stop sending data to the FPGA memory.

This command can be called from ACQUIRING state to finish an acquisition, or from any other state to reset the acquisition.

Usage:	ACQU:STOP <value> *Or* ACQUisition:STOP <value>
Type:	READ_WRITE
Set value:	Fixed to True (not used but required)
Returns:	None
Ex.:	*ACQU:STOP True* None

Technical description

When an acquisition is stopped, the saving of data samples in the FPGA memory is also stopped. Tis is done writing in the FPGA Memory register which disabled the data saving. Apart from this, the data sending through

the harmony bus is also stopped by resetting the output ID of the ADC channels in the ADCCORE. And finally, if the acquisition type is with partial acquisitions, the counter which is use to count those partial acquisitions is also disabled.

FPGA blocks	MEMORY, ADCCORE & DIGITALIO	
Registers	*MEMORY: CTL_SAVE*	0x00 ➔No save data
	ADCCORE: ID_CH1_ID	0x00 ➔ Rest Channel 1 ID. Stop to send data tho harmony bus.
	ADCCORE: ID_CH2_ID	0X00 ➔ Same as channel 1
	ADCCORE: ID_CH3_ID	0X00 ➔ Same as channel 1
	ADCCORE: ID_CH4_ID	0X00 ➔ Same as channel 1
	DIGITALIO:DIS_STP_DIS1_M	0x03 ➔ Disable mode to manual control
	DIGITALIO:DIS_STP_WD1	0x0 ➔ Disable harmoy counter

3.4.8 TIME

This command sets the acquisition time in milliseconds. The minimum acquisition time that can be sets corresponds to 0.320 milliseconds.

This can be read always, but can only be modified when the acquisition state is not ACQUIRING.

The acquisition time value is only applied to the current amplifiers when the acquisition starts.

Usage:	ACQU:TIME <value> o*r* ACQUisition:TIME <value> *Or* ACQU:TIME? o*r* ACQUisition:TIME?
Type:	READ_WRITE
Set value:	String that contains the number with the acquisition time to be set
Returns:	<string> that contains the acquisition time set.
Ex.:	*ACQU:TIME?* 1000 *ACQU:TIME 500* 500

Technical description

This command is used to set the internal variable ***acquistionr_time*** that is to configure the acquisition type, when it starts. This parameters is not allowed to be modified when an acquisition is already running.

3.5 CHANNEL COMMANDS

These commands are used to configure or check the status of the acquisition channels independently. They are grouped under the ***CHANnelXX*** component, where XX is the channel number from 01 to 04 (two digits). So to execute these commands you need to enter the component first, then follow by ':' and then the command.

By default, if no additional parameter or command is send, it executes the ***INSCurrent*** command

3.5.1 AVGCurrent

This command returns the average current value of all samples acquired in a particular channel. The data used to calculate this average value is the current data stored in the FPGA memory.

This command can always be read and it is reset when a new acquisition starts.

Usage:	CHANXX:AVGC? *Or* CHANnelXX:AVGCurrent?
Type:	READ_ONLY
Returns:	<string> that contains value of the average current in millivolts.
Ex.:	*CHAN02:AVGC?* -1.453476

Technical description

This command returns the average value of all samples acquired by a particular channel (see technical descrition of CURRent). Each channel has an internal buffer where they store the data stored in the FPGA Memory. That data comes from the AVERAGE module (ID values `AVG_ID_X` where X is the channel number).

Each channel buffer is reset when a new acquisition starts, so this values are also reset.

3.5.2 CURRent

This command returns the current buffer that contains all samples acquired in a particular channel, since the acquisition started. The data returned by this command is data acquired and stored in the FPGA memory.

This command can always be read and it is reset when a new acquisition starts.

Usage:	CHANXX:CURR? *Or* CHANnelXX:CURRent?
Type:	READ_ONLY
Returns:	<string> that contains a joined list will all the current values in millivolts read in a channel.
Ex.:	*CHAN03:CURR?* [0.000153, 0.000153]

Technical description

This command returns the buffer of samples acquired by a particular channel. This buffer is filled with the data that comes from the FPGA Memory, which is the data that comes from the AVERAGE module of each channel. So basically, the data to be returned is that data with `AVG_ID_X` as id value in the FPGA Memory. Remember that each data stored contains three fields: the timestamp; the data; the ID of the module who has originated the data.

When the acquisition is using partial acquisitions, the data coming form the AVERAGE modules is stored in temporal buffers and when the number of partial acquisitions to count is reached, the average value of this temporal buffer is added to the channel buffer.

Each channel buffer is reset when a new acquisition starts.

3.5.3 INSCurrent

This command returns the instant current read by the ADC module in the FPGA. This is a direct read of the ADC channel.

This command can always be read and it is reset when a new acquisition starts.

Usage:	CHANXX:INSC? *Or* CHANnelXX:INSCurrent?
Type:	READ_ONLY
Returns:	<string> that contains the instant current value in millivolts read in a channel.
Ex.:	*CHAN04:INSC?* 2.063828

Technical description

This commands uses the directly the output of the ADCCORE module to show instant current values. The value returned is a voltage value, so it needs to be converted to current. For that conversion, the acquisition range is used with the following formula:

Current = (ADC_value read * acquisition_range) / maximum_voltage

Where the maximum voltage range that can be measured is 10Volts.

FPGA blocks	ADCCORE	
Registers	*ADC_CH1*	Voltage value channel 1
	ADC_CH2	Voltage value channel 2
	ADC_CH3	Voltage value channel 3
	ADC_CH4	Voltage value channel 4

3.5.4 CABO

This component is used to the get information and controls the current amplifier boards. This is a group of commands are grouped under the ***CHANnelXX:CABOard*** component, where XX is the channel number from 01 to 04 (two digits). So to execute these commands you need to enter the component first, then follow by ':' and then the command.

By default, if no additional parameter or command is send, it executes the ***INIT*** command

3.5.4.1 FILTer

This command configures the current amplifier filter, which sets the prefilter and postfilter, for a particular channel. It is measured in Hz and the parameter is integer value which corresponds to the filter value listed below (see "Set Value")

This can be read always, but can only be modified when the acquisition state is not ACQUIRING.

The filter value is only applied to the current amplifiers when the acquisition starts.

Usage:	CHANXX:CABO:FILT <value> o*r* CHANnelXX:CABOard:FILTer <value> *Or* CHANXX:CABO:FILT? o*r* CHANnelXX:CABOard:FILTer?
Type:	READ_WRITE
Set value:	0: 3200 Hz 1: 100 Hz 2: 10 Hz 3: 1 Hz 4: 0.5Hz
Returns:	<string> that contains the value set to the filter in a particular channel.
Ex.:	*CHAN01:CABO:FILT?* 3200 *CHAN01:CABO:FILT 1* 100

Technical description

All CABO (Current Amplifier Board) commands are set using the SPI module. Although the acquisition filter is also modifying this data, this command allows an individual modification of the filter set for a particular channel.

As it has been already explained, the Filter command means to change the PreFilter and the PostFilter values.

For more details see section "*1.3.1 Current Amplifiers configuration/monitoring through SPI devices"* of the specification document ALBA Em# Configuration SPI

(Apply only specific value for selected channel)

FPGA blocks	SPI	
Registers	*For Channel 1:*	
	ctl_spi_addr:	0x01 ➔ Chip Select for PostFilter the Port Expander 1 of CA #1
	ctl_spi_addr:	0x02 ➔ Chip Select for PreFilter the Port Expander 2 of CA #1
	For Channel 2:	
	ctl_spi_addr:	0x06 ➔ Chip Select for PostFilter the Port Expander 1 of CA #2
	ctl_spi_addr:	0x07 ➔ Chip Select for PreFilter the Port Expander 2 of CA #2
	For Channel 3	
	ctl_spi_addr:	*0x0b ➔ Chip Select for* PostFilter *the Port Expander 1 of CA #3*
	ctl_spi_addr:	*0x0c ➔ Chip Select for* PreFilter *the Port Expander 2 of CA #3*
	For Channel 4:	
	ctl_spi_addr:	*0x10 ➔ Chip Select for* PostFilter *the Port Expander 2 of CA #4*
	ctl_spi_addr:	*0x11 ➔ Chip Select for* PreFilter *the Port Expander 2 of CA #4*
	Other register used to transmit/receive data though SPI: *ctl_tx, ctl_tx_data, ctl_tx_data_length, ctl_rx, ctl_rx_data_length, ctl_start, ctl_rx_data*	

3.5.4.2 INIT

This command is used to initialize a particular current amplifier board. Control of the CA board is done though the SPI module in the FPGA that uses some port expanders to control it. During the initialization, the following values are applied:

- Set Inversion to True
- Sets Filter to 320 Hz, which means Postfilter to 3200 Hz and PreFilter to 3500 Hz
- Sets range to 1mA, which means trans impedance gain to 10k and V gain to 1

This can be read always, but can only be modified when the acquisition state is not ACQUIRING.

Usage:	CHANXX:CABO <value> *or* CHANXX:CABO:INIT <value> *or* CHANnelXX:CABOard:INIT <value> *Or* CHANXX:CABO? *or* CHANXX:CABO:INIT? *or* CHANnelXX:CABOard:INIT?
Type:	READ_WRITE
Set value:	Fixed to True (not used but required)
Returns:	<string> that contains information if the channel has been already initialized.
Ex.:	*CHAN01:CABO:INIT?* True *CHAN01:CABO:INIT True* True

Technical description

This command initializes a current amplifier module using the SPI module. The initialization is done in three steps. First doing dummy readings to all items in the CA module and setting de DAC load to ON. Then, doing a real initialization, by setting all port expanders bits as outputs. Finally, configuring the port expanders with default values, it means: Inversion set to On; trans-impedance gain to 10k; PostFilter set to 3200 Hz; PreFilter set to 3500Hz; V Gain set to 1.

For more details see section *"1.3.1 Current Amplifiers configuration/monitoring through SPI devices"* of the specification document ALBA Em# Configuration SPI

(Apply only specific value for selected channel)

FPGA blocks	SPI	
Registers	*For Channel 1:*	
	ctl_spi_addr:	0x00 ➔ Chip Select for for the EEPROM Memory ofCA #1
	ctl_spi_addr:	0x01 ➔ Chip Select for the Port Expander 1 of CA #1
	ctl_spi_addr:	0x02 ➔ Chip Select for the Port Expander 2 of CA #1
	ctl_spi_addr:	0x03 ➔ Chip Select for the DAC of CA #1
	For Channel 2:	
	ctl_spi_addr:	0x05 ➔ Chip Select for for the EEPROM Memory ofCA #2
	ctl_spi_addr:	0x06 ➔ Chip Select for the Port Expander 1 of CA #2
	ctl_spi_addr:	0x07 ➔ Chip Select for the Port Expander 2 of CA #2
	ctl_spi_addr:	0x08 ➔ Chip Select for the DAC of CA #2
	For Channel 3	
	ctl_spi_addr:	0x0a ➔ Chip Select for for the EEPROM Memory ofCA #3

	ctl_spi_addr:	0x0b ➔ Chip Select for the Port Expander 1 of CA #3
	ctl_spi_addr:	0x0c ➔ Chip Select for the Port Expander 2 of CA #3
	ctl_spi_addr:	0x0d ➔ Chip Select for the DAC of CA #3
	For Channel 4:	
	ctl_spi_addr:	0x0f ➔ Chip Select for for the EEPROM Memory ofCA #4
	ctl_spi_addr:	0x10 ➔ Chip Select for the Port Expander 1 of CA #4
	ctl_spi_addr:	0x11 ➔ Chip Select for the Port Expander 2 of CA #4
	ctl_spi_addr:	0x12 ➔ Chip Select for the DAC of CA #4
	Other register used to transmit/receive data though SPI: *ctl_tx, ctl_tx_data, ctl_tx_data_length, ctl_rx, ctl_rx_data_length, ctl_start, ctl_rx_data*	

3.5.4.3 INVErsion

This command is used to set the inverse gain for a certain channel.

This can be read always, but can only be modified when the acquisition state is not ACQUIRING.

Usage:	CHANXX:CABO:INVE <value> *or* CHANnelXX:CABOard:INVErsion <value> *Or* CHANXX:CABO:INVE? *or* CHANnelXX:CABOard:INVErsion?
Type:	READ_WRITE
Set value:	True: to enable inversion False: to disable inversion
Returns:	<string> with Boolean information about inversion enable.
Ex.:	*CHAN01:CABO:INVE?* True *CHAN01:CABO:INVE False* False

Technical description

The inversion of a channel can be controlled also independently using the SPI module.

For more details see section *"1.3.1 Current Amplifiers configuration/monitoring through SPI devices"* of the specification document ALBA Em# Configuration SPI

(Apply only specific value for selected channel)

FPGA blocks	SPI	
Registers	*For Channel 1:*	
	ctl_spi_addr:	0x01 ➔ Chip Select for PostFilter the Port Expander 1 of CA #1
	For Channel 2:	
	ctl_spi_addr:	0x06 ➔ Chip Select for PostFilter the Port Expander 1 of CA #2
	For Channel 3	
	ctl_spi_addr:	*0x0b ➔ Chip Select for* PostFilter *the Port Expander 1 of CA #3*
	For Channel 4:	
	ctl_spi_addr:	*0x10 ➔ Chip Select for* PostFilter *the Port Expander 2 of CA #4*

	Other register used to transmit/receive data though SPI: *ctl_tx, ctl_tx_data, ctl_tx_data_length, ctl_rx, ctl_rx_data_length, ctl_start, ctl_rx_data*

3.5.4.4 POSTfilter

The current amplifier post filter can be controlled of using the *FILTer* command or in particular, through this command. Same as the *FILTer* command, it is measured in Hz and the parameter is integer value which corresponds to the filter value listed below (see "Set Value")

This can be read always, but can only be modified when the acquisition state is not ACQUIRING.

Usage:	CHANXX:CABO:POST <value> *or* CHANnelXX:CABOard:POSTfilter <value> *Or* CHANXX:CABO:POST? *or* CHANnelXX:CABOard:POSTfilter?
Type:	READ_WRITE
Set value:	0: 3200 Hz 1: 100 Hz 2: 10 Hz 3: 1 Hz
Returns:	<string> that contains the value set to the post filter in a particular channel.
Ex.:	*CHAN01:CABO:POST?* 3200 *CHAN01:CABO:POST 1* 100

Technical description

Same as for the for the filter command, the Post filter command can be independently adjusted using te SPI module.

For more details see section "*1.3.1 Current Amplifiers configuration/monitoring through SPI devices"* of the specification document ALBA Em# Configuration SPI

(Apply only specific value for selected channel)

FPGA blocks	SPI	
Registers	*For Channel 1:*	
	ctl_spi_addr:	0x01 ➔ Chip Select for Inversion the Port Expander 1 of CA #1
	For Channel 2:	
	ctl_spi_addr:	0x06 ➔ Chip Select for Inversion the Port Expander 1 of CA #2
	For Channel 3	
	ctl_spi_addr:	*0x0b ➔ Chip Select for* Inversion *the Port Expander 1 of CA #3*
	For Channel 4:	
	ctl_spi_addr:	*0x10 ➔ Chip Select for* Inversion *the Port Expander 2 of CA #4*
	Other register used to transmit/receive data though SPI: *ctl_tx, ctl_tx_data, ctl_tx_data_length, ctl_rx, ctl_rx_data_length, ctl_start, ctl_rx_data*	

3.5.4.5 PREFilter

Same as post filter command, the current amplifier pre filter can be controlled of using the *FILTer* command or in particular, through this command. Same as the *FILTer* command, it is measured in Hz and the parameter is integer value which corresponds to the filter value listed below (see "Set Value")

This can be read always, but can only be modified when the acquisition state is not ACQUIRING.

Usage:	CHANXX:CABO:PREF <value> *or* CHANnelXX:CABOard:PREFilter <value> *Or* CHANXX:CABO:PREF? *or* CHANnelXX:CABOard:PREFilter?
Type:	READ_WRITE
Set value:	0: 3500 Hz 1: 100 Hz 2: 10 Hz 3: 1 Hz 4: 0.5 Hz
Returns:	<string> that contains the value set to the pre filter in a particular channel.
Ex.:	*CHAN01:CABO:PREF?* 3500 *CHAN01:CABO:PREF 1* 100

Technical description

Same as for the for the filter command, the Pre filter command can be independently adjusted using te SPI module.

For more details see section "*1.3.1 Current Amplifiers configuration/monitoring through SPI devices"* of the specification document ALBA Em# Configuration SPI

(Apply only specific value for selected channel)

FPGA blocks	SPI	
Registers	*For Channel 1:*	
	ctl_spi_addr:	0x02 ➔ Chip Select for PreFilter the Port Expander 1 of CA #1
	For Channel 2:	
	ctl_spi_addr:	0x07 ➔ Chip Select for PreFilter the Port Expander 1 of CA #2
	For Channel 3	
	ctl_spi_addr:	*0x0C ➔ Chip Select for* PreFilter *the Port Expander 1 of CA #3*
	For Channel 4:	
	ctl_spi_addr:	*0x11 ➔ Chip Select for* PreFilter *the Port Expander 2 of CA #4*
	Other register used to transmit/receive data though SPI: *ctl_tx, ctl_tx_data, ctl_tx_data_length, ctl_rx, ctl_rx_data_length, ctl_start, ctl_rx_data*	

3.5.4.6 RANGe

This command configures the current amplifier range, which results of combining the trans impedance and voltage gains, for a particular ca board. The parameter is an integer value which corresponds to factor to apply over the maximum range (see "Set Value") .

It is measured in mA and the parameter is integer value which corresponds to the range value listed below (see "Set Value").

Usage:	CHANXX:CABO:RANG <value> *or* CHANnelXX:CABOard:RANGe <value> *Or* CHANXX:CABO:RANG? *or* CHANnelXX:CABOard:RANGe?
Type:	READ_WRITE
Set value:	0: 1 1: 0.1 2: 0.01 3: 0.001 4: 0.0001 5: 0.00001 6: 0.000001 7: 0.0000001
Returns:	<string> that contains the common value set to the four acquisition channels for the filter.
Ex.:	*CHAN03:CABO:RANG?* 1 *CHAN03:CABO:RANG 5* 0.000001

Technical description

Same as all CABO (Current Amplifier Board) commands, it uses the SPI module and although the acquisition range command is also modifying this data, this command allows an individual modification of the range set for a particular channel.

As it has been already explained, the Filter command means to change the trans impedance and voltage gain values.

For more details see section "*1.3.1 Current Amplifiers configuration/monitoring through SPI devices"* of the specification document ALBA Em# Configuration SPI

(Apply only specific value for selected channel)

FPGA blocks	SPI	
Registers	*For Channel 1:*	
	ctl_spi_addr:	0x01 ➔ Chip Select for TIGain the Port Expander 1 of CA #1
	ctl_spi_addr:	0x02 ➔ Chip Select for VGain the Port Expander 2 of CA #1
	For Channel 2:	
	ctl_spi_addr:	0x06 ➔ Chip Select for TIGain the Port Expander 1 of CA #2
	ctl_spi_addr:	0x07 ➔ Chip Select for VGain the Port Expander 2 of CA #2
	For Channel 3	
	ctl_spi_addr:	*0x0b ➔ Chip Select for* TIGain *the Port Expander 1 of CA #3*
	ctl_spi_addr:	*0x0c ➔ Chip Select for* VGain *the Port Expander 2 of CA #3*

	For Channel 4:	
	ctl_spi_addr:	*0x10* ➔ *Chip Select for* TIGain *the Port Expander 2 of CA #4*
	ctl_spi_addr:	*0x11* ➔ *Chip Select for* VGain *the Port Expander 2 of CA #4*
	Other register used to transmit/receive data though SPI: *ctl_tx, ctl_tx_data, ctl_tx_data_length, ctl_rx, ctl_rx_data_length, ctl_start, ctl_rx_data*	

3.5.4.7 TEMPerature

This command returns information about the current amplifier temperature.

Usage:	CHANXX:CABO:TEMP? *Or* CHANXX:CABOard:TEMPeratue?
Type:	READ_ONLY
Returns:	<string> that contains the ca board temperature in degrees
Ex.:	*CHAN03:CABO:TEMP?* 23.575

Technical description

Used in diagnostics, the carrier board temperature for each channel is read also using the SPI module.

For more details see section "*1.3.1 Current Amplifiers configuration/monitoring through SPI devices"* of the specification document ALBA Em# Configuration SPI

(Apply only specific value for selected channel)

FPGA blocks	SPI	
Registers	*For Channel 1:*	
	ctl_spi_addr:	0x04 ➔ Chip Select for the Temperature Sensor of CA #1
	For Channel 2:	
	ctl_spi_addr:	0x09 ➔ Chip Select for the Temperature Sensor of CA #2
	For Channel 3	
	ctl_spi_addr:	0x0E ➔ Chip Select for the Temperature Sensor of CA #3
	For Channel 4:	
	ctl_spi_addr:	0x13 ➔ Chip Select for the Temperature Sensor of CA #4
	Other register used to transmit/receive data though SPI: *ctl_tx, ctl_tx_data, ctl_tx_data_length, ctl_rx, ctl_rx_data_length, ctl_start, ctl_rx_data*	

3.5.4.8 TIGAin

The current amplifier trans-impedance gain is part of the RANGe command, but can also be controlled independently through this command.

It is measured in ohms and the parameter is integer value which corresponds to the ti gain value listed below (see "Set Value")

This can be read always, but can only be modified when the acquisition state is not ACQUIRING

Usage:	CHANXX:CABO:TIGA <value> *or* CHANnelXX:CABOard:TIGAin <value> *Or* CHANXX:CABO:TIGA? *or* CHANnelXX:CABOard:TIGAin?
Type:	READ_WRITE
Set value:	0: 10k 1: 1M 2: 100M 3: 1G 4: 10G
Returns:	<string> that contains the ti gain for the specified channel.
Ex.:	*CHAN03:CABO:TIGA?* 1M *CHAN03:CABO:TIGA 4* 10G

Technical description

Same as for the for the range command, the trans impedance gain command can be independently adjusted using te SPI module.

For more details see section "*1.3.1 Current Amplifiers configuration/monitoring through SPI devices"* of the specification document ALBA Em# Configuration SPI

(Apply only specific value for selected channel)

FPGA blocks	SPI	
Registers	*For Channel 1:*	
	ctl_spi_addr:	0x01 ➔ Chip Select for TIGain the Port Expander 1 of CA #1
	For Channel 2:	
	ctl_spi_addr:	0x06 ➔ Chip Select for TIGain the Port Expander 1 of CA #2
	For Channel 3	
	ctl_spi_addr:	*0x0b ➔ Chip Select for* TIGain *the Port Expander 1 of CA #3*
	For Channel 4:	
	ctl_spi_addr:	*0x10 ➔ Chip Select for* TIGain *the Port Expander 2 of CA #4*
	Other register used to transmit/receive data though SPI: *ctl_tx, ctl_tx_data, ctl_tx_data_length, ctl_rx, ctl_rx_data_length, ctl_start, ctl_rx_data*	

3.5.4.9 VGAIn

Same as the current amplifier Trans impedance gain, the Voltage gain is part of the RANGe command, but can also be controlled independently through this command.

The parameter is an integer value which corresponds to the v gain value listed below (see "Set Value")

This can be read always, but can only be modified when the acquisition state is not ACQUIRING

Usage:	CHANXX:CABO:VGAI <value> *or* CHANnelXX:CABOard:VGAIn <value> *Or* CHANXX:CABO:VGAI? *or* CHANnelXX:CABOard:VGAIn?
Type:	READ_WRITE

Set value:	0: 1 1: 10 2: 50 3:100 4: sat
Returns:	<string> that contains the v gain for the specified channel.
Ex.:	*CHAN03:CABO:VGAI?* 1 *CHAN03:CABO:VGAI 4* Sat

Technical description

Same as for the for the range command, the voltage gain command can be independently adjusted using te SPI module.

For more details see section *"1.3.1 Current Amplifiers configuration/monitoring through SPI devices"* of the specification document ALBA Em# Configuration SPI

(Apply only specific value for selected channel)

FPGA blocks	SPI	
Registers	*For Channel 1:*	
	ctl_spi_addr:	0x02 ➔ Chip Select for VGain the Port Expander 1 of CA #1
	For Channel 2:	
	ctl_spi_addr:	0x07 ➔ Chip Select for VGain the Port Expander 1 of CA #2
	For Channel 3	
	ctl_spi_addr:	*0x0C ➔ Chip Select for* VGain *the Port Expander 1 of CA #3*
	For Channel 4:	
	ctl_spi_addr:	*0x11 ➔ Chip Select for* VGain *the Port Expander 2 of CA #4*
	Other register used to transmit/receive data though SPI: *ctl_tx, ctl_tx_data, ctl_tx_data_length, ctl_rx, ctl_rx_data_length, ctl_start, ctl_rx_data*	

3.6 IOPORT COMMANDS

These commands are used to configure or check the status of the Digital Input Output ports (up to 4, LEMO connectors in the instrument rear side) and the General Purpose Input Output ports (up to 9 in the DB26 connector). They are grouped under the ***IOPOrtXX*** component, where XX is port number that goes form 01 to 13, where from 01 to 04 are the DIO ports and from 05 to 13 are the GPIO, it means:

- IOPOrt01 to IOPOrt04 are equivalent to DIO 1 to DIO 4
- IOPOrt 05 to IOPOrt13 are equivalent to GPIO 1 to GPIO 9

To execute these commands you need to enter the component first, then follow by ':' and then the command.

By default, if no additional parameter or command is send, it executes the ***NAME*** command

3.6.1 CONFig

This command is used to configure the port as Input (1) or Output (0)

This command can always be executed and ports can be modified, except that port used in the acquisition with trigger hardware, if he acquisition has already started.

Usage:	IOPOXX:CONF <value> *or* IOPOrt XX:CONFig <value> *or* IOPOXX:CONF? *or* IOPOrt XX:CONFig ?
Type:	READ_WRITE
Set value:	0: OUTPUT 1: INPUT
Returns:	<string> that port input or output type
Ex.:	*IOPO02:CONFIG 1* INPUT *IOPO02:CONFIG?* INPUT

Technical description

The configuration of the IO ports means to modify the setting in he digitalIO FPGA module which controls the IO ports and also to set the configuration in the front end board (FE), which is controlled by the SPI module. IO ports can be GPIO or High Speed ports, so depending of the port type the Port Expander address of the Front End board to select changes.

FPGA blocks	DIGITALIO, SPI	
Registers	`DIGITAL:DIGITALIO`	Sets if GPIO is an input(1) or output(0). The bit 0 refers to DIO1 and bit 4 to DIFF_IO_1
	`DIGITAL:DIO_STP_IOP`	If it is 1 the signal is polarity is changed. The bit 0 refers to DIO1 and bit 4 to DIFF_IO_1
	For GPIO ports:	
	ctl_spi_addr:	0x18 ➔ Chip Select for Port Expander 1 of the Front End Board
	ctl_spi_addr:	0x19 ➔ Chip Select for Port Expander2 of the Front End Board
	For High Speed IO ports	
	ctl_spi_addr:	0x1a ➔ Chip Select for Port Expander3 of the Front End Board
	Other register used to transmit/receive data though SPI: *ctl_tx, ctl_tx_data, ctl_tx_data_length, ctl_rx, ctl_rx_data_length, ctl_start, ctl_rx_data*	

3.6.2 NAME

This command is to get the port type name

Usage:	IOPOXX:NAME? *or* IOPOrtXX:NAME?
Type:	READ_ONLY
Returns:	<string> that contains the port name

Ex.:	*IOPO02:NAME?* DIO_2

3.6.3 VALUe

This command is used to read the port value in any case (input or output) or to set a value in case of the port will be configured as output port. To set a value in an input port is not allowed.

Usage:	IOPOXX:VALU <value> *or* IOPOrtXX:VALUe <value> *or* IOPOXX:VALE? *or* IOPOrtXX:VALUe?
Type:	READ_WRITE
Set value:	0: Set 0V to the port output 1: Set 5V to the port output
Returns:	<string> bool that informs about the status of the port
Ex.:	*IOPO02:VALU 1* True *IOPO02:VALU?* False

Technical description

To get the value of an IOPort it is just enough to read the value form the DigitalIO block in the FPGA:

FPGA blocks	DIGITALIO	
Registers	`DIO_V_DIO_V`	Digital input output Value, first the DIO1..4 and then DiffIO1..9.

3.7 Voltage Supply Commands

These commands are used to configure and control the 4 Differention I/O switch supply ports available in the FrontEnd board. Depending on the configuration of the DAC gain (explained in section 3.8.1), they provide voltage ranges tat go from 0 to 10V or from -10V to 10V. Voltage output control is independent for the 4 outputs, but DAC gain is common. So the voltage range output is configured for all the outputs. These ports are labeled as AOUT1 to AOUT4 in the back side of the equipment. They are grouped under the SUPPlyXX component, where XX is port number that goes form 01 to 04.

To execute these commands you need to enter the component first, then follow by : and then the command.

By default, if no additional parameter or command is send, it executes the VALU command

3.7.1 VALUe

This command is used to read the supply port value or to set a voltage a value in case of the port will be configured as output port. To set a value in an input port is not allowed.

Usage:	SUPPXX:VALU <value> or SUPPlyXX:VALUe <value> Or SUPPXX:VALU? or SUPPlyXX:VALUe?
Type:	READ_WRITE
Set value:	Voltage value to set in the output port that depends on the DAC Gain configuration. These values can go from 0 to 10 when DAC Gain set to 0, or from -10 to 10 when DAC Gain is set to 1
Returns:	<string> that contains the voltage output value
Ex.:	SUPP02:VALU -8 *-8* SUPP02:VALU? *-8*

Technical description

Switched supply ports are located in the FrontEnd board, so the control of these ports is done through the SPI module.

FPGA blocks	SPI	
Registers	*ctl_spi_addr:*	0x18 ➔ Chip Select for Port Expander 1 of the Front End Board
	Other register used to transmit/receive data though SPI: *ctl_tx, ctl_tx_data, ctl_tx_data_length, ctl_rx, ctl_rx_data_length, ctl_start, ctl_rx_data*	

3.8 DAC Commands

These commands are used to configure and control the high speed DAC Gain that is used for the 4 Differential switch supply ports available in the FrontEnd board. These commands are grouped under the ODAC component.

To execute these commands you need to enter the component first, then follow by : and then the command.

By default, if no additional parameter or command is send, it executes the GAIN command

3.8.1 GAIN

This command is used to configure the DAC Gain from 0 to 2.5 V or from 0 to 5V. These values correspond to analogue output voltage ranges that go from 0V to 10V or from -10V to 10V. Output voltage value levels are configured using the SUPPXX:VALU command explained in section 3.7.1

Usage:	ODAC:GAIN <value> Or ODAC:GAIN?
Type:	READ_WRITE
Set value:	0: DAC Gain output= 0 - 2.5V (AOUT: 0 - 10V) 1: DAC Gain output= 0 - 5V (AOUT: -10V - 10V)
Returns:	<string> DAC gain voltage value set

Ex.:	ODAC:GAIN 1 *1* ODAC:GAIN? *-1*

Technical description

High speed DAC that controls the 4x switched supply ports is located in the FrontEnd board, so the control of this DAC is done through the SPI module.

FPGA blocks	SPI	
Registers	*ctl_spi_addr:*	0x1a ➔ Chip Select for the Port Expander 3 of the Front End Board
	Other register used to transmit/receive data though SPI: *ctl_tx, ctl_tx_data, ctl_tx_data_length, ctl_rx, ctl_rx_data_length, ctl_start, ctl_rx_data*	

*3.9 High Voltage Commands**

***** Only for models including the HV IN.*

These commands are used to configure and control the flying voltage or high voltage in the Em# models that include the HV input (HV IN). Flying voltage control is done by the equipment, so these commands are basically used to monitor the status. Above a certain security voltage level 50V, the High voltage LED is ON to alert te user there is high voltage in the output and above a second voltage level the output relay is disabled. It is possible to configure a third protection level b the user, using the maximum and minimum voltage levels commands to define a user protection window. Apart from this, outside this protection levels, it is possible to manually control the status of the HV relay using the corresponding command. These commands are grouped under the FVCTrl component.

To execute these commands you need to enter the component first, then follow by : and then the command.

By default, if no additional parameter or command is send, it executes the STAT command

*3.9.1 AVERage**

** Only for models including the HV IN.*

This command is used to get the average voltage value read in the HV Input.

Usage:	FVCT:AVER? or FVCTrl:AVERage?
Type:	READ_ONLY
Returns:	<string> that contains the average voltage value read
Ex.:	FVCT:AVER? *10*

Technical description

FV control module is continuously reading the HV input and calculates the average voltage of the last 256 samples. This command is used to get this value of the average voltage calculated by the FV control module od the FPGA. To calculate the voltage from the FV_CTRL reading, it uses the following formula:

V=2.5*(1001*FV/1024 - 500)

FPGA blocks	FV_CTRL	
Registers	READ_MODE: 0x04 FV	Reading mode of Floating voltage set to the average value Floating ground Voltage read

*3.9.2 MAXLimit**

** Only for models including the HV IN.*

Apart from the predefined protection level included in the equipment, it is possible for the user to define it's own protection level. This is done specifying the window of allowed voltages, defining its maximum and minimum levels. These command is used to read or configure maximum voltage level permitted for the user protection level.

Usage:	FVCT:MAXL <value> or FVCTrl:MAXLimit <value> Or FVCT:MAXL? or FVCTrl:MAXLimit?
Type:	READ_WRITE
Set value:	Voltage level to set. Allowed values goes from -1200V to 1200V
Returns:	<string> that contains the maximum voltage value set
Ex.:	FVCT:MAXL 500 *500* FVCT:MAXL? *500*

Technical description

The maximum voltage before switching off the FV_Relay signal.The voltage follows the following formula

V=2.5*(1001*Lim_Max/1024 - 500)

FPGA blocks	FV_CTRL	
Registers	LIM_MAX	Maximum voltage level set

*3.9.3 MAXVoltage**

** Only for models including the HV IN.*

This command is used to get the maximum voltage level read in the HV Input.

Usage:	FVCT:MAXV? or FVCTrl:MAXVoltage?

Type:	READ_ONLY
Returns:	<string> that contains the maximum voltage value read
Ex.:	FVCT:MAXV? *12*

Technical description

FV control module is continuously reading the HV input and stores the maximum voltage read in it. This command is used to get this value of the maximum calculated by the FV control module of the FPGA. To calculate the voltage from the FV_CTRL reading, it uses the following formula:

V=2.5*(1001*FV/1024 - 500)

FPGA blocks	FV_CTRL	
Registers	READ_MODE: 0x01 FV	Reading mode of Floating voltage set to the maximum value Floating ground Voltage read

*3.9.4 MINLimit**

** Only for models including the HV IN.*

Same as the MAXLim explained in section 3.9.2, apart from the predefined protection level included in the equipment, it is possible for the user to define it's own protection level. This is done specifying the window of allowed voltages, defining its maximum and minimum levels. These command is used to read or configure minimum voltage level permitted for the user protection level.

Usage:	FVCT:MINL <value> or FVCTrl:MINLimit <value> Or FVCT:MINL? or FVCTrl:MINLimit?
Type:	READ_WRITE
Set value:	Voltage level to set. Allowed values goes from -1200V to 1200V
Returns:	<string> that contains the minimum voltage value set
Ex.:	FVCT:MINL -500 *-500* FVCT:MINL? *-500*

Technical description

The minimum voltage before switching off the FV_Relay signal.The voltage follows the following formula

V=2.5*(1001*Lim_Min/1024 - 500)

FPGA blocks	FV_CTRL	
Registers	LIM_MIN	Minimum voltage level set

3.9.5 MINVoltage*

** Only for models including the HV IN.*

Same as the MAXVoltage command explained in section 3.9.3, this command is used to get the minimum voltage level read in the HV Input.

Usage:	FVCT:MINV? or FVCTrl:MINVoltage?
Type:	READ_ONLY
Returns:	<string> that contains the minimum voltage value read
Ex.:	FVCT:MINV? *-12*

Technical description

FV control module is continuously reading the HV input and stores the minimum voltage read in it. This command is used to get this value of the minimum calculated by the FV control module of the FPGA. To calculate the voltage from the FV_CTRL reading, it uses the following formula:

$$V=2.5*(1001*FV/1024 - 500)$$

FPGA blocks	FV_CTRL	
Registers	READ_MODE: 0x02 FV	Reading mode of Floating voltage set to the minimum value Floating ground Voltage read

3.9.6 RELAy*

** Only for models including the HV IN.*

This command is used to control the status of the HV relay.

Usage:	FVCT:RELA <value> or FVCTrl:RELAy <value> Or FVCT:RELA? or FVCTrl:RELAy?
Type:	READ_WRITE
Set value:	0: Relay is OFF 1: Relay is ON
Returns:	<string> that contains the status of the relay
Ex.:	FVCT:RELA 1 *1* FVCT:RELA? *1*

Technical description

When reading this value, it returns the status of the FV relay (1 means FV is ON). when writing, it changes the state od the FV relay if the FV voltage is inside the correct rangeFV control module is

FPGA blocks	FV_CTRL	
Registers	RELAY	Floating ground Voltage read

3.9.7 RESEt*

** Only for models including the HV IN.*

This command is used to reset the internal values calculated like the maximum, minimum and average voltages read.

Usage:	FVCT:RESE <value> or FVCTrl:RESEt <value>
Type:	WRITE_ONLY
Set value:	Fixed to True (not used but required)
Returns:	None
Ex.:	FVCT:RESE True None

Technical description

Maximum, minimun and average values are continuously read and its values are calculated and stored in the main software. This command is used to reset these values to 0. As this control in hte main software, the FV_CTL is not affected.

3.9.8 STATe*

** Only for models including the HV IN.*

This command returns the status of the of the high voltage in HV IN input.

Usage:	FVCT:STAT? or FVCTrl:STATe?
Type:	READ_ONLY
Returns:	<string> that contains the HV status detected at the HV IN
Ex.:	FVCT:STAT? *0*

Technical description

This comand is used to get the status of the of FV LED. "0" means LED is off.

FPGA blocks	FV_CTRL	
Registers	LED	FV_Led_sta

3.9.9 VOLTage*

** Only for models including the HV IN.*

This command is used to get the instant voltage level read in the HV Input.

Usage:	FVCT:VOLT? or FVCTrl:VOLTage?
Type:	READ_ONLY
Returns:	<string> that contains the instant voltage value read
Ex.:	FVCT:VOLTV? *6*

Technical description

FV control module is continuously reading the HV input. This command is used to get the instant voltage read by the FV control module of the FPGA. To calculate the voltage from the FV_CTRL reading, it uses the following formula:

$$V=2.5*(1001*FV/1024 - 500)$$

FPGA blocks	FV_CTRL	
Registers	READ_MODE: 0x00	Reading mode of Floating voltage set to the instant value
	FV	Floating ground Voltage read

4 TABLE SUMMARY

This section just shows all available commands presented in table summary:

Component	Command	Type	Parameters	Description
---	***IDN**	READ_ONLY	?	Identification information
	***MAC**	READ_ONLY	?	Mac address
	***HLP**	READ_ONLY	?	shows short information about the list of available commands
	***RST**	WRITE_ONLY	True	General OS reset
	***SHT**	WRITE_ONLY	True	General OS shutdown
	RECOnfigure	WRITE_ONLY	True	Reconfigures this instrument according the data defined in the Config file
	FWVErsion	READ_ONLY	?	Version of the FPGA firmware
DIAGnostics**:**	**CBTE**mperature	READ_ONLY	?	Temperature of the carrier board
	FETEmperature	READ_ONLY	?	Temperature of the frontend board
	VSENse	READ_ONLY	?	***-default-*** Sensed voltages for the different voltages supplies
TRIGger**:**	**DELA**y	READ_WRITE	? <delay>	Delay in milliseconds of the acquisition after the trigger.
	INPUt	READ_WRITE	? 0: DIO_1 1: DIO_2 2: DIO_3 3: DIO_4 4: DIFF_IO_1 5: DIFF_IO_2 6: DIFF_IO_3 7: DIFF_IO_4 8: DIFF_IO_5 9: DIFF_IO_6 10: DIFF_IO_7 11: DIFF_IO_8 12: DIFF_IO_9	Input used for the trigger
	MODE	READ_WRITE	? 0: SOFTWARE 1: HARDWARE	Sets the trigger type internal (sw) or external (hw)
	POLArity	READ_WRITE	? 0: FALLING 1: RISING	Sets the polarity of the trigger at the input
	SWSEt	WRITE_ONLY	True	Sets a SW trigger when trigger mode is SOFTWARE
	STATe	READ_ONLY	?	***-default-*** Returns trigger configuration
ACQUisition**:**	**FILT**er	READ_WRITE	? 0: 3200 Hz 1: 100 Hz 2: 10 Hz 3: 1 Hz 4: 0.5Hz	configures the acquisition filter
	MEAS	READ_ONLY	?	All channels data acquired during the last acquisition
	NDATa	READ_ONLY	?	Number of triggers detected (hardware or software)
	RANGe	READ_WRITE	? 0: 1	Configures the acquisition range

			1: 0.1 2: 0.01 3: 0.001 4: 0.0001 5: 0.00001 6: 0.000001 7: 0.0000001	
	STARt	WRITE_ONLY	True	Starts an acquisition
	STATe	READ_ONLY	?	***-default-*** Acquisition status
	STOP	WRITE_ONLY	True	Stops an acquisition or resets a failed acquisition
	TIME	READ_WRITE	? <time>	Set acquisition time
CHANnel**XX:**	**AVGC**urrent	READ_ONLY	?	Average current value of all samples acquired in a particular channel
	CURRent	READ_ONLY	?	Buffer that contains all current samples acquired in a particular channel
	INSCurrent	READ_ONLY	?	***-default-*** Instant current read by the ADC module in the FPGA
CHANnel**XX:CABO**ard:	**FILT**er	READ_WRITE	? 0: 3200 Hz 1: 100 Hz 2: 10 Hz 3: 1 Hz 4: 0.5Hz	Configs current amplifier filter for a channel
	INIT	READ_WRITE	? True	***-default-*** Initializes a particular current amplifier board
	INVErsion	READ_WRITE	? True False	Sets the inverse gain for a certain channel
	POSTfilter	READ_WRITE	? 0: 3200 Hz 1: 100 Hz 2: 10 Hz 3: 1 Hz	Sets the post filter of a certain current amplifier board
	PREFilter	READ_WRITE	0: 3500 Hz 1: 100 Hz 2: 10 Hz 3: 1 Hz 4: 0.5 Hz	Sets the pre filter of a certain current amplifier board
	RANGe	READ_WRITE	? 0: 1 1: 0.1 2: 0.01 3: 0.001 4: 0.0001 5: 0.00001 6: 0.000001 7: 0.0000001 8: AUTO	Current amplifier range for a particular ca board
	TEMPerature	READ_ONLY	?	Current amplifier temperature
	TIGAin	READ_WRITE	? 0: 10k 1: 1M	Current amplifier trans impedance gain

			2: 100M 3: 1G 4: 10G	
	VGAIn	READ_WRITE	? 0: 1 1: 10 2: 50 3: 100 4: sat	Current amplifier voltage gain
IOPOrt**XX:**	**CONF**ig	READ_WRITE	? 0: OUTPUT 1: INPUT	Check or set the port type as input or output
	NAME	READ_ONLY	?	***default-*** Gets port name
	VALUe	READ_WRITE	? 0: Set to 0V 1: Set to 5V	Checks or sets the port value
SUPPXX:	**VALUe**	READ_WRITE	? <voltage>	***default-*** Check or set the voltage a supply port
ODAC:	**GAIN**	READ_WRITE	? 0: 0-2.5V 1: 0-5V	***default*** Configure the DAC Gain
FVCTrl:	**AVER**age	READ_ONLY	?	Command is used to get the average FV voltage
	MAXLimit	READ_WRITE	? <max_limit>	Command is used to set the user maximum voltage level
	MAXVoltage	READ_ONLY	?	Command is used to get the maximum FV voltage
	MINLimit	READ_WRITE	? <min_limit>	Command is used to set the user minimum voltage level
	MINVoltage	READ_ONLY	?	Command is used to get the minimum FV voltage
	RELAy	READ_WRITE	? 0: OFF 0: ON	Sets the FV relay
	RESEt	WRITE_ONLY	True	Resets the FV data
	STATe	READ_ONLY	True	***default-*** Returns the status of the FV detection
	VOLTage	READ_ONLY	?	Command is used to get the instant FV voltage

5 TANGO DEVICE SERVER

This ALBA Em# uses SCPI, as explained in section 2 SCPI Command Syntax as communication control protocol. An easy introduction of this instrument to any TANGO control system can be done, by installing the Skippy tango device which includes the ALBA EM command list.

Not all available commands in the ALBA Em# have been implement, but just the most useful ones to configure an acquisition, the trigger or to use the IOPorts.

The skippy has been developed at the ALBA Synchrotron and the source code can be found in: TANGO Skippy DS

5.1 PROPERTIES

The first thing to configure in this Skippy instrument is the property which defines the equipment to control. ThIs can be done using the device property **Instrument** which sets the equipment IP or hostname. But there are others also used to configure the communication with the instrument:

Name	Description	Type	Default Value
Instrument	The name of the instrument to use	String	None
Port	In case of socket interface the port value can be changed	short	5025
NumChannels	Number of channels available in the instrument, if it has	short	0
NumFunctions	Number of functions available in the instrument, if it has	short	0
MonitoredAttributes	When the device is in RUNNING state, the attributes listed here will be monitored (having events) with a period said in the attribute TimeStampsThreashold (or different if specified with a : separator after the attrName)	String[]	None
AutoOn	When device startup, try an on() to connect to the instrument automatically	boolean	True
AutoStart	When device startup, try an Start() to monitor attributes, if MonitoredAttributes is configuredc, authomatically	boolean	True
AutoStandby	When device startup, try an standby() to connect to the instrument automatically.	boolean	True

5.2 COMMANDS

All commands available in he Skippy device server are used to check/configure the status of the communication with the equipment. In any case they can be used to execute commands of the ALBA Em#.

This is the list of available commands:

Name	Input Type	Output type	Description
State	DEV_VOID	DEV_STATE	This command gets the device state (stored in its device state data member) and returns it to the caller.
Status	DEV_VOID	CONST_DEV_STRING	This command gets the device status (stored in its device_status data member) and returns it to the caller.
IDN	DEV_VOID	DEV_STRING	Request identification to the instrument.
Start	DEV_VOID	DEV_BOOLEAN	Start an active monitoring.
Stop	DEV_VOID	DEV_BOOLEAN	Stop the active monitoring.
On	DEV_VOID	DEV_BOOLEAN	Allow communication with the instrument.
Off	DEV_VOID	DEV_BOOLEAN	Release the communication with the instrument.
Exec	DEV_STRING	DEV_STRING	evaluate python code inside the device server. This command can be very helpful and dangerous.
AddMonitoring	DEV_STRING	DEV_BOOLEAN	Add an attribute to the list of monitored attributes
RemoveMonitoring	DEV_STRING	DEV_BOOLEAN	Remove an attribute from the list of monitored attributes
SetMonitoringPeriod	DEVVAR_STRINGARRAY	DEV_BOOLEAN	From the list of already monitored attributes, stablish (or change) the period that it is checked.
GetMonitoringPeriod	DEV_STRING	DEV_FLOAT	Get the period that is checked an attribute monitored.
CMD	DEV_STRING	DEV_STRING	Expert command for a direct send of a SCPI command and read the answer.
CMDfloat	DEV_STRING	DEVVAR_FLOATARRAY	Expert command for a direct send of a SCPI command and read the answer converted to a float list.
Standby	DEV_VOID	DEV_BOOLEAN	Stablish communication with the instrument.

5.3 ATTRIBUTES

These are really the commands of the ALBA Em#. Each command of the instrument is convert to an attribute in the Skippy device server.

This is the list of available commands:

Name	Type	Data type	Description
IDN	READ	DEV_STRING	Instrument identification
QueryWindow	READ_WRITE	DEV_STRING	When many attributes are requested at the same time, they are grouped in subqueries of this size
TimeStampsThreshold	READ_WRITE	DEV_STRING	This value sets the threshold time to use a cached value or hardware read it
MAC	READ	DEV_STRING	MAC address of the instrument
TriggerDelay	READ_WRITE	DEV_STRING	Delay in milliseconds of the acquisition after the trigger.

TriggerInput	READ_WRITE	DEV_STRING	Input used for the trigger: 0: DIO_1 1: DIO_2 2: DIO_3 3: DIO_4 4: DIFF_IO_1 5: DIFF_IO_2 6: DIFF_IO_3 7: DIFF_IO_4 8: DIFF_IO_5 9: DIFF_IO_6 10: DIFF_IO_7 11: DIFF_IO_8 12: DIFF_IO_9
TriggerMode	READ_WRITE	DEV_STRING	Sets the trigger type internal (sw) or external (hw) 0: SOFTWARE 1: HARDWARE
TriggerPolarity	READ_WRITE	DEV_STRING	Sets the polarity of the trigger at the input 0: FALLING 1: RISING
SWTrigger	READ_WRITE	DEV_STRING	Sets a SW trigger when trigger mode is SOFTWARE
AcqFilter	READ_WRITE	DEV_STRING	configures the acquisition filter 0: 3200 Hz 1: 100 Hz 2: 10 Hz 3: 1 Hz 4: 0.5Hz
Meas	READ	DEV_STRING	All channels data acquired during the last acquisition
NData	READ	DEV_STRING	Number of triggers detected (hardware or software)
AcqRange	READ_WRITE	DEV_STRING	Configures the acquisition range 0: 1 1: 0.1 2: 0.01 3: 0.001 4: 0.0001 5: 0.00001 6: 0.000001 7: 0.0000001
AcqStart	READ_WRITE	DEV_STRING	Starts an acquisition
AcqState	READ	DEV_STRING	Acquisition status
AcqStop	READ_WRITE	DEV_STRING	Stops an acquisition or resets a failed acquisition
AcqTime	READ_WRITE	DEV_STRING	Set acquisition time
Channel commands where XX is the channel number and goes from 01 to 04			
CHANXX_InstantCurrent	READ	DEV_STRING	Average current value of all samples acquired in a particular channel.
CHANXX_Current	READ	DEV_STRING	Buffer that contains all current samples acquired in a particular channel
CHANXX_AverageCurrent	READ	DEV_STRING	Instant current read by the ADC module in the FPGA
CHANXX_CARange	READ_WRITE	DEV_STRING	Configs current amplifier range for a channel

			0: 1 1: 0.1 2: 0.01 3: 0.001 4: 0.0001 5: 0.00001 6: 0.000001 7: 0.0000001
CHANXX_CAFilter	READ_WRITE	DEV_STRING	Configs current amplifier filter for a channel 0: 3200 Hz 1: 100 Hz 2: 10 Hz 3: 1 Hz 4: 0.5Hz
CHANXX_CAPostFilter	READ_WRITE	DEV_STRING	Gets/Sets the post filter of a certain current amplifier board 0: 3200 Hz 1: 100 Hz 2: 10 Hz 3: 1 Hz
CHANXX_CAPreFilter	READ_WRITE	DEV_STRING	Gets/Sets the pre filter of a certain current amplifier board 0: 3500 Hz 1: 100 Hz 2: 10 Hz 3: 1 Hz 4: 0.5 Hz
CHANXX_CAInversion	READ_WRITE	DEV_STRING	Configs current amplifier inversion for a channel True False
CHANXX_CATIGain	READ_WRITE	DEV_STRING	Configs current amplifier TIGAin for a channel 0: 10k 1: 1M 2: 100M 3: 1G 4: 10G
CHANXX_CAVGain	READ_WRITE	DEV_STRING	Configs current amplifier VGainion for a channel 0: 1 1: 10 2: 50 3: 100 4: sat
High Speed ports only, where XX goes from 01 to 04.			
IOXX_CONFIG	READ_WRITE	DEV_STRING	Configure Hisgh Speed IOPort XX, 0 as Output 1 as Input
IOXX	READ_WRITE	DEV_STRING	Returns/Sets the value of a High Speed IO Port.

6 WEB SERVER

The ALBA Em# includes a graphical interface via web, based on WebSocket protocol, to display and monitor the status and configuration of the instrument. WebSocket is a computer communications protocol, providing full-duplex communication channels over a single TCP connection, which facilitates real-time data transfer between server and clients. The WebSocket Protocol is an independent TCP-based protocol and its only relationship to HTTP is that its handshake is interpreted by HTTP servers as an Upgrade request.

The WebSocket protocol is currently supported in most major browsers including Google Chrome, Microsoft Edge, Internet Explorer, Firefox, Safari and Opera. WebSocket also requires web applications on the server to support it. Following table shows from which version is available for the main web browser:

Protocol	Draft date	Internet Explorer	Firefox (PC)	Firefox (Android)	Chrome (PC, Mobile)	Safari (Mac, iOS)	Opera (PC, Mobile)	Android Browser
RFC6455	Dec.2011	10	11	11	16	6	12.10	4.4

The ALBA Em# web, is used only as a diagnostics or status report. So for the “normal” user, any parameter can be modified or changed from this page.

To access this page, just connect to the instrument ip or hostname, to the port 8888, from any internet browser. It means:

http:*Hostname_or_ipaddress*:8888

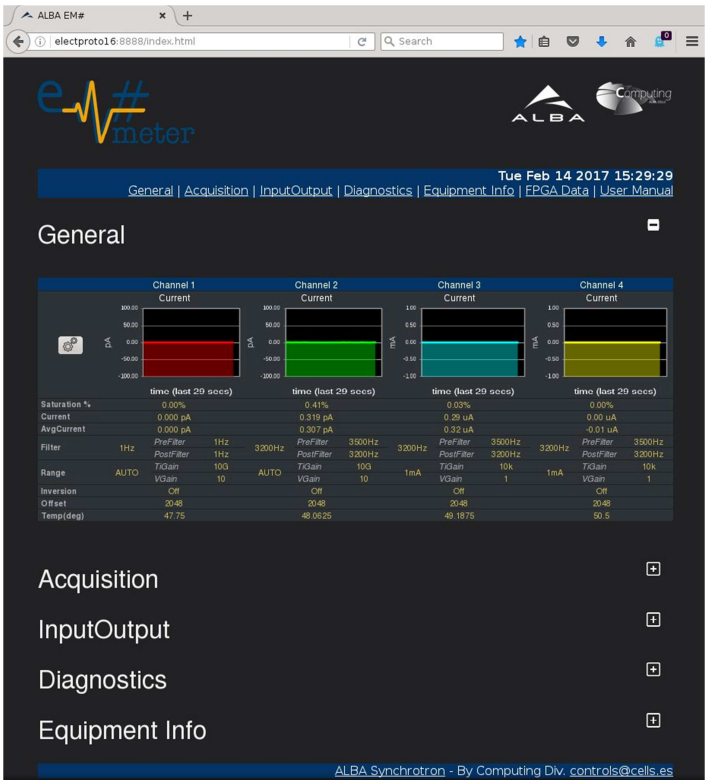

Figure 4. AlbaEm# web interface

Main screen in ***Figure 4***, shows the general status information of the 4 acquisition channels . For each channel, saturation, current, average current, filter, range, offset and the current amplifier temperature values are shown in this section.

Apart from this it is possible to check the acquisition configuration and data in the “acquisition” sections, as shown in ***Figure 5*** . It is also possible to save the acquisition data and its configuration to a file.

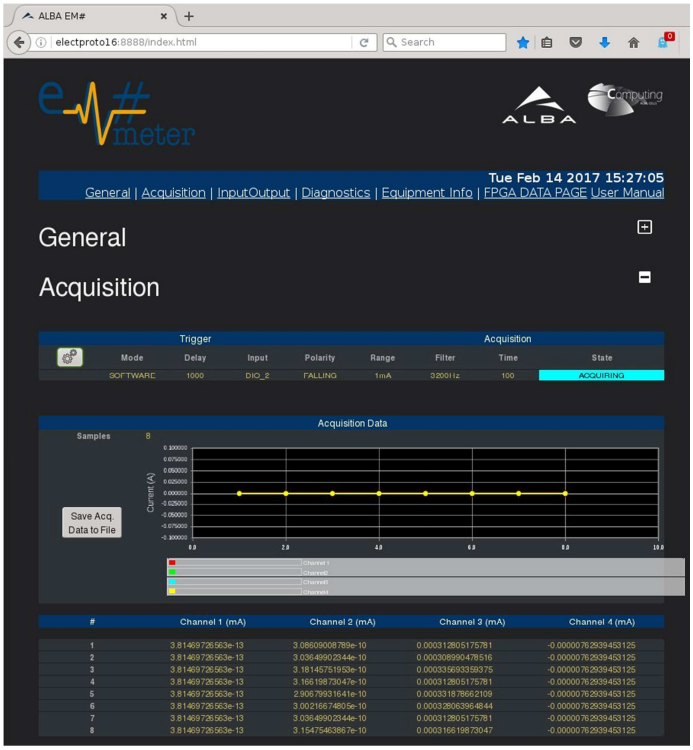

Figure 5. Data acquisition

Configuration and status of 13 Digital InputOutput ports and the 4 Analog is visible in the “InputOutput” section and the equipment diagnostics status or general equipment information are available by displaying in the “Diagnostics” and “Equipment Info” sections.

FPGA data is available for developers, showing information about FPGA blocks and its registers values. They are shown in tables where their registers are listed with its current values. Detailed information about a FPGA block and its register information can be showed just clicking on the FPGA block name.

Finally, it is also to check an html version of the “user manual” or directly download tis manual from the web, by accessing to the User Manual link on the web.

7 DISPLAY

Local equipment control is possible through to the touch-screen display. The Display includes navigation through menus as shown in ***Figure 6***, used to let the user check the overall status of the current inputs, the status of the I/O ports or just simply get general equipment information, like the mac, Sw version or hostname.

Overall control of the equipment is not accessible using the local display; it has to be done using the remote control. Through the local display it is only possible to launch the auto calibration sequence or change the range or filter settings for each current channel input

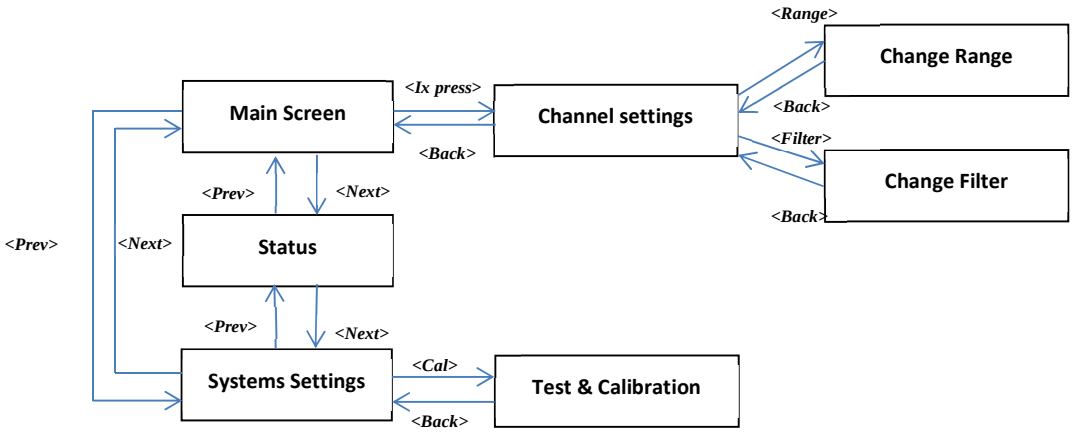

Figure 6. *Display menus block diagram.*

After 2 minutes without keyboard screen touch action, the display backlight is decreased and the screensaver picture is displayed.

7.1 Main Screen

The main screen (***Figure 7***) shows the current reading of the instant values of the four values simultaneously. One value per second refresh rate.

The current reading for each channel (x=1,2,3,4) "Ix=" uses three digits and ½ values, as detailed in the following table.

Range	Format (8 char)
100 pA	1XX.X pA
1 nA	1XXX pA
10 nA	1X.XX pA
100 nA	1XX.X nA
1 µA	1XXX nA
10 µA	1X.XX µA
100 µA	1XX.X µA
1 mA	1XXX µA

On the top of the screen, there is the status line, which is kept for all screens and shows different useful information for the user appear in the status bar.

- **AUTO** together with the channel number **1, 2, 3** or **4** appears when the range is selected to be AUTO for a channel.
- **ACQ** (Acquiring) appears if an acquisition is running (via trigger hardware of software). The message appears flashing.
- **HV** (High voltage) appears when HV bias is detected. This message also appears flashing.

7.2 Channel Settings

From the main screen, simply touch a particular channel to enter in the channel settings screen (***Figure 8***). It shows a few more detail information about the selected channel and lets the user to modify some parameters, like the selected range or the selected filter.

The current reading includes a little more detail as in the main screen. In this screen it shows five digits and ½ values as detailed in the following table:

Range	Format (8 char)
100 pA	1XX.XXX pA
1 nA	1XXX.XX pA
10 nA	1X.XXXX pA
100 nA	1XX.XXX nA
1 µA	1XXX.XX nA

10 µA	1X.XXXX µA
100 µA	1XX.XXX µA
1 mA	1XXX.XX µA

Simple touch on the “range” or “filter” enters in the change range (***Figure 9***) or change filter screens (***Figure 10***) letting the user to modify any of these parameters for the selected channel. The information is refresh periodically so if any of this parameters is changed remotely, the change is reflected in this screens..

7.3 Status Screen

From the main screen, it is possible to access to the status screen (***Figure 11***) to check the status of the 4 I/O ports, the 4 analogue outputs, or the High Voltage bias detected. This is just an information screen, so any parameter can be modified from this screen.

As explained in 7.1.***Main Screen*** on top of this screen is the status line, which shows important information about the equipment status. Below this line, there is the 4 high speed I/O ports status line.

- **TR** shows the status of the trigger input. In normal font state it reflects a low level state at the input and inverted font is high level state present at the input.
- **DIx** or **DOx** shows the status of the 4 I/O ports, where x goes from 1 to 4. When the port is set as an input DI is shown and when it is set as an output, DO. Same as before, if a low level state is detected or being output, the port appears with normal letters. When a high level state is detected or it is output, the port appears with inverted letters.

The next line shows the **HV_Bias** voltage (high voltage bias) detected in the “HV_IN” input, whihc appears in the format “1XXX V”.

Finally, the 4 DAC analogue outputs (A_OUTx) are shown from “**A1**” to “**A4**” using the “±1X.X V” format

7.4 System Settings

Also the main screen, it is possible to access to the system settings screen (***Figure 12Figure 11***). This is screen shows general equipment information like the Mac address, IP and hostnames, FW and SW versions or the Model name.

From this screen, the **Test/Cal** enters in the autotesting and auto calibration procedures, that let the user to calibrate the complete equipment together an independent calibration of the four current amplifiers. The autocalibration procedure requires an external calibration equipment, specially designed for this new AlbaEm# to be connected at any of the USB inputs.

APPENDIX

A. CONFIGURATION FILE

The new ALBA Em# has been designed to work not only as an electrometer, but also as a multipurpose instrument. This is possible by applying different configurations to the available registers inside the FPGA blocks. The idea for the main software running in the NUC SBC, is to work as a library of functions or functionalities that can be enabled or disabled using a configuration file, without the need to change the software running in the instrument.

In this section, it is shown the current configuration file used for the new ALBA Em#.

```
#   GENERAL CONFIGURATION FILE FOR ALBA INSTRUMENTATION     #
#
#       AlIn Project - ALBA SYNCHROTRON LIGHT SOURCE
#                  Copyright 2016, ALBA
#                Author: mbroseta@cells.es
#
#


#-------------------------------------------------------------------------
# Project related configuration options
#-------------------------------------------------------------------------

# The MANUFACTURER_ is the name of the AlIn manufacturer

MAIN_MANUFACTURER_ = ALBA SYNCHROTRON

# The INSTRUMENT_ contains teh name of the equipment

MAIN_INSTRUMENT_ = Electrometer 2

# The SERIAL_ contains teh serial number assigned to thsi particular equipment

MAIN_SERIAL_ = 000000000

# If the DEBUG_ flag is set to 1, all scpi log information will be added to
# the log file and to the screen output. Setting this flag to 0, will
# just disable the log information through screen

MAIN_DEBUG_ = 0

# The DEBUGLEVEL_ is an integer value which indicates the expected debug
# level for the corresponding device. Expected values are:
#       10 = DEBUG
#       20 = INFO
#       30 = WARNING
#       40 = ERROR
#       50 = CRITICAL

MAIN_DEBUGLEVEL_ = 20


#-------------------------------------------------------------------------
#
# Applications
#
#-------------------------------------------------------------------------

#-------------------------------------------------------------------------
# SCPI driver configuration
#-------------------------------------------------------------------------
```

```
# if the SCPI_AUTOOPEN_ flag is set, then the scpi module wil ltry to open the
# defined communication PORT_ by itself. Otherwhise the port will have to be
# opened manually

SCPI_AUTOOPEN_ = 1

# If the SCPI_LOCAL_ flag is set, the scpi application will open the local
# addess as the default communication address, it means 0.0.0.0

SCPI_LOCAL_ = 0

# The SCPI_PORT_ is an integer value that it is used to specify the port number
# that will be used in the communications through telnet service. By default it
# is set to localhost, port 5025

SCPI_PORT_ = 5025

# If the SCPI_DEBUG_ flag is set to 1, all scpi log information will be added to
# the log file and to the screen output. Setting this flag to 0, will
# just disable the log information through screen


SCPI_DEBUG_ = 0


#-------------------------------------------------------------------------
#
# Middleware
#
#-------------------------------------------------------------------------

#-------------------------------------------------------------------------
# HARMONY_ device configuration
#-------------------------------------------------------------------------

# If the DEBUG_ flag is set to 1, all log information will be added to
# the log file and to the screen output. Setting this flag to 0, will
# just disable the log information through screen

HARMONY_DEBUG_ = 1

# The DEBUGLEVEL_ is an integer value which indicates the expected debug
# level for the corresponding device. Expected values are:
#       10 = DEBUG
#       20 = INFO
#       30 = WARNING
#       40 = ERRORACQU:TIME 3000;TRIG:MODE 1;TRIG:INPU 0;IOPO03:CONFIG 0;ACQU:START 1
#       50 = CRITICAL

HARMONY_DEBUGLEVEL_ = 20

#-------------------------------------------------------------------------
# DIAGNOSTICS device configuration
#-------------------------------------------------------------------------

# If the DEBUG_ flag is set to 1, all log information will be added to
# the log file and to the screen output. Setting this flag to 0, will
# just disable the log information through screen

DIAGS_DEBUG_ = 1

# The DEBUGLEVEL_ is an integer value which indicates the expected debug
# level for the corresponding device. Expected values are:
#       10 = DEBUG
#       20 = INFO
#       30 = WARNING
#       40 = ERROR
#       50 = CRITICAL

DIAGS_DEBUGLEVEL_ = 40
```

```
#------------------------------------------------------------------------------
# Display device configuration
#------------------------------------------------------------------------------

# If the DEBUG_ flag is set to 1, all log information will be added to
# the log file and to the screen output. Setting this flag to 0, will
# just disable the log information through screen

DISPLAY_DEBUG_ = 1

# The DEBUGLEVEL_ is an integer value which indicates the expected debug
# level for the corresponding device. Expected values are:
#       10 = DEBUG
#       20 = INFO
#       30 = WARNING
#       40 = ERROR
#       50 = CRITICAL

DISPLAY_DEBUGLEVEL_ = 10

#------------------------------------------------------------------------------
# Power Supply Board device configuration
#------------------------------------------------------------------------------

# If the DEBUG_ flag is set to 1, all log information will be added to
# the log file and to the screen output. Setting this flag to 0, will
# just disable the log information through screen

PSB_DEBUG_ = 1

# The DEBUGLEVEL_ is an integer value which indicates the expected debug
# level for the corresponding device. Expected values are:
#       10 = DEBUG
#       20 = INFO
#       30 = WARNING
#       40 = ERROR
#       50 = CRITICAL

PSB_DEBUGLEVEL_ = 40


#------------------------------------------------------------------------------
#
# Drivers
#
#------------------------------------------------------------------------------

#------------------------------------------------------------------------------
# ADC FPGA device module configuration
#------------------------------------------------------------------------------

# The DEVICE_ parameter refers to the spec module name that will take care of the
# control for this device. It should match with the name of the devices defined
# in the alin device list, included in the /alin/deviceslib folder

ADCCORE_DEVICE_ = WB-FMC-ADC-CORE

# If the DEBUG_ flag is set to 1, all log information will be added to
# the log file and to the screen output. Setting this flag to 0, will
# just disable the log information through screen

ADCCORE_DEBUG_ = 1

# The DEBUGLEVEL_ is an integer value which indicates the expected debug
# level for the corresponding device. Expected values are:
#       10 = DEBUG
#       20 = INFO
#       30 = WARNING
#       40 = ERROR
#       50 = CRITICAL
```

```
ADCCORE_DEBUGLEVEL_ = 40


#-------------------------------------------------------------------------
# ADC driver to monitor status ofl PSB board
#-------------------------------------------------------------------------

# The I2C_ADDRESS_ contains the 7 MSB shifted 1 bit to the right address
# of the device to control through I2C protocol.
# I.e.: Display I2C address is 0x90 for reading and 0x91 for writing. In this
# case the 7 MSB corresponds to 0x90 >> 1 = 0x4A

ADS7828_I2C_ADDRESS_ = 0x48

# The I2C_BUS_ contains the bus number to be used in the I2C communications.
# Alin system can use one of the two different i2c-designware-pci buses for
# the I2C protocol (I2C bus 0 or I2C bus 1). You can check the corresponding
# numbers detected for these buses by executing the command i2cdetect -l
# With this, display panel is using the I2C bus 0, that corresponds to I2C
# bus number 8

ADS7828_I2C_BUS_ = 9

# If the DEBUG_ flag is set to 1, all log information will be added to
# the log file and to the screen output. Setting this flag to 0, will
# just disable the log information through screen

ADS7828_DEBUG_ = 1

# The DEBUGLEVEL_ is an integer value which indicates the expected debug
# level for the corresponding device. Expected values are:
#       10 = DEBUG
#       20 = INFO
#       30 = WARNING
#       40 = ERROR
#       50 = CRITICAL

ADS7828_DEBUGLEVEL_ = 40

#-------------------------------------------------------------------------
# Average FPGA device module configuration
#-------------------------------------------------------------------------

# The DEVICE_ parameter refers to the spec module name that will take care of the
# control for this device. It should match with the name of the devices defined
# in the alin device list, included in the /alin/deviceslib folder

AVERAGE_DEVICE_ = WB-HRMY-AVERAGE

# If the DEBUG_ flag is set to 1, all log information will be added to
# the log file and to the screen output. Setting this flag to 0, will
# just disable the log information through screen

AVERAGE_DEBUG_ = 1

# The DEBUGLEVEL_ is an integer value which indicates the expected debug
# level for the corresponding device. Expected values are:
#       10 = DEBUG
#       20 = INFO
#       30 = WARNING
#       40 = ERROR
#       50 = CRITICAL

AVERAGE_DEBUGLEVEL_ = 40

#-------------------------------------------------------------------------
# Crossbar FPGA device module configuration
#-------------------------------------------------------------------------

# The DEVICE_ parameter refers to the spec module name that will take care of the
# control for this device. It should match with the name of the devices defined
```

```
# in the alin device list, included in the /alin/deviceslib folder

CROSSBAR_DEVICE_ = WB-HRMY-CROSSBAR

# If the DEBUG_ flag is set to 1, all log information will be added to
# the log file and to the screen output. Setting this flag to 0, will
# just disable the log information through screen

CROSSBAR_DEBUG_ = 1

# The DEBUGLEVEL_ is an integer value which indicates the expected debug
# level for the corresponding device. Expected values are:
#       10 = DEBUG
#       20 = INFO
#       30 = WARNING
#       40 = ERROR
#       50 = CRITICAL

CROSSBAR_DEBUGLEVEL_ = 40

#-------------------------------------------------------------------------
# Digitial IO FPGA device module configuration
#-------------------------------------------------------------------------

# The DEVICE_ parameter refers to the spec module name that will take care of the
# control for this device. It should match with the name of the devices defined
# in the alin device list, included in the /alin/deviceslib folder

DIGITALIO_DEVICE_ = WB-EM2-DIGITAL_IO

# If the DEBUG_ flag is set to 1, all log information will be added to
# the log file and to the screen output. Setting this flag to 0, will
# just disable the log information through screen

DIGITALIO_DEBUG_ = 1

# The DEBUGLEVEL_ is an integer value which indicates the expected debug
# level for the corresponding device. Expected values are:
#       10 = DEBUG
#       20 = INFO
#       30 = WARNING
#       40 = ERROR
#       50 = CRITICAL

DIGITALIO_DEBUGLEVEL_ = 40

#-------------------------------------------------------------------------
# eDIP128 display driver configuration
#-------------------------------------------------------------------------

# The I2C_ADDRESS_ contains the 7 MSB shifted 1 bit to the right address
# of the device to control through I2C protocol.
# I.e.: Display I2C address is 0x94 for reading and 0x95 for writing. In this
# case the 7 MSB corresponds to 0x94 >> 1 = 0x4A

EDIP128_I2C_ADDRESS_ = 0x4A

# The I2C_BUS_ contains the bus number to be used in the I2C communications.
# Alin system can use one of the two different i2c-designware-pci buses for
# the I2C protocol (I2C bus 0 or I2C bus 1). You can check the corresponding
# numbers detected for these buses by executing the command i2cdetect -r

EDIP128_I2C_BUS_ = 8


# If the DEBUG_ flag is set to 1, all log information will be added to
# the log file and to the screen output. Setting this flag to 0, will
# just disable the log information through screen

EDIP128_DEBUG_ = 1
```

```
# The DEBUGLEVEL_ is an integer value which indicates the expected debug
# level for the corresponding device. Expected values are:
#       10 = DEBUG
#       20 = INFO
#       30 = WARNING
#       40 = ERROR
#       50 = CRITICAL

EDIP128_DEBUGLEVEL_ = 40

#-------------------------------------------------------------------------
# FIFO FPGA device module configuration
#-------------------------------------------------------------------------

# The DEVICE_ parameter refers to the spec module name that will take care of the
# control for this device. It should match with the name of the devices defined
# in the alin device list, included in the /alin/deviceslib folder

FIFO_DEVICE_ = WB-HRMY-FIFO

# The MAXSIZE_ parameter defines the maximum defined size for the FIFO for
# each of the channels. This should be returned by the FPGA. This value is set
# just in case

FIFO_MAXSIZE_ = 1000

# If the DEBUG_ flag is set to 1, all log information will be added to
# the log file and to the screen output. Setting this flag to 0, will
# just disable the log information through screen

FIFO_DEBUG_ = 1

# The DEBUGLEVEL_ is an integer value which indicates the expected debug
# level for the corresponding device. Expected values are:
#       10 = DEBUG
#       20 = INFO
#       30 = WARNING
#       40 = ERROR
#       50 = CRITICAL

FIFO_DEBUGLEVEL_ = 40

#-------------------------------------------------------------------------
# IDGen FPGA device module configuration
#-------------------------------------------------------------------------

# The DEVICE_ parameter refers to the spec module name that will take care of the
# control for this device. It should match with the name of the devices defined
# in the alin device list, included in the /alin/deviceslib folder

IDGEN_DEVICE_ = WB-HRMY-ID-GEN

# If the DEBUG_ flag is set to 1, all log information will be added to
# the log file and to the screen output. Setting this flag to 0, will 1
# just disable the log information through screen

IDGEN_DEBUG_ = 1

# The DEBUGLEVEL_ is an integer value which indicates the expected debug
# level for the corre rango de adquisicionsponding device. Expected values are:
#       10 = DEBUG
#       20 = INFO
#       30 = WARNING
#       40 = ERROR
#       50 = CRITICAL

IDGEN_DEBUGLEVEL_ = 40

#-------------------------------------------------------------------------
# MCP23008 Port Expander driver to configure PSB board
#-------------------------------------------------------------------------
```

```
# The I2C_ADDRESS_ contains the 7 MSB shifted 1 bit to the right address
# of the device to control through I2C protocol.
# I.e.: Display I2C address is 0x40 for reading and 0x41 for writing. In this
# case the 7 MSB corresponds to 0x40 >> 1 = 0x20

PEXPDRV_I2C_ADDRESS_ = 0x20

# The I2C_BUS_ contains the bus number to be used in the I2C communications.
# Alin system can use one of the two different i2c-designware-pci buses for
# the I2C protocol (I2C bus 0 or I2C bus 1). You can check the corresponding
# numbers detected for these buses by executing the command i2cdetect -r

PEXPDRV_I2C_BUS_ = 9

# If the DEBUG_ flag is set to 1, all log information will be added to
# the log file and to the screen output. Setting this flag to 0, will
# just disable the log information through screen

PEXPDRV_DEBUG_ = 1

# The DEBUGLEVEL_ is an integer value which indicates the expected debug
# level for the corresponding device. Expected values are:
#       10 = DEBUG
#       20 = INFO
#       30 = WARNING
#       40 = ERROR
#       50 = CRITICAL

PEXPDRV_DEBUGLEVEL_ = 40

#-------------------------------------------------------------------------
# Harmony memory configuration
#-------------------------------------------------------------------------

# If the DEBUG_ flag is set to 1, all log information will be added to
# the log file and to the screen output. Setting this flag to 0, will
# just disable the log information through screen

MEMORY_DEBUG_ = 1

# The DEBUGLEVEL_ is an integer value which indicates the expected debug
# level for the corresponding device. Expected values are:
#       10 = DEBUG
#       20 = INFO
#       30 = WARNING
#       40 = ERROR
#       50 = CRITICAL

MEMORY_DEBUGLEVEL_ = 40

# The BUFFERSIZE_ is an integer that specifies the maximum buffer lenght for
# of the channel id buffers that the harmony bus can have (up to 256 Id)

MEMORY_BUFFERSIZE_ = 20000


#-------------------------------------------------------------------------
# SPI device configuration
#-------------------------------------------------------------------------

# The DEVICE_ parameter refers to the spec module name that will take care of the
# control for this device. It should match with the name of the devices defined
# in the alin device list, included in the /alin/deviceslib folder

SPI_DEVICE_ = SPI

# If the DEBUG_ flag is set to 1, all log information will be added to
# the log file and to the screen output. Setting this flag to 0, will
# just disable the log information through screen
```

```
SPI_DEBUG_ = 1

# The DEBUGLEVEL_ is an integer value which indicates the expected debug
# level for the corresponding device. Expected values are:
#       10 = DEBUG
#       20 = INFO
#       30 = WARNING
#       40 = ERROR
#       50 = CRITICAL

SPI_DEBUGLEVEL_ = 20


#-------------------------------------------------------------------------
#
# Base Drivers
#
#-------------------------------------------------------------------------

#-------------------------------------------------------------------------
# AlIn driver configuration
#-------------------------------------------------------------------------

# If the SIMULATED_ flag is set to 1, there will not be communication
# with the spec_pci card and all values returned will be simulated random
# values. Default expected setting should be 0

ALIN_SIMULATED = 0

# If the DEBUG_ flag is set to 1, all log information will be added to
# the log file and to the screen output. Setting this flag to 0, will
# just disable the log information through screen

ALIN_DEBUG_ = 0

# The DEBUGLEVEL_ is an integer value which indicates the expected debug
# level for the corresponding device. Expected values are:
#       10 = DEBUG
#       20 = INFO
#       30 = WARNING
#       40 = ERROR
#       50 = CRITICAL

ALIN_DEBUGLEVEL_ = 40


#-------------------------------------------------------------------------
# AlInDevice driver configuration
#-------------------------------------------------------------------------

# If the DEBUG_ flag is set to 1, all log information will be added to
# the log file and to the screen output. Setting this flag to 0, will
# just disable the log information through screen

ALINDEV_DEBUG_ = 0

# The DEBUGLEVEL_ is an integer value which indicates the expected debug
# level for the corresponding device. Expected values are:
#       10 = DEBUG
#       20 = INFO
#       30 = WARNING
#       40 = ERROR
#       50 = CRITICAL

ALINDEV_DEBUGLEVEL_ = 40
```

B. ACQUISITION FPGA CONFIGURATION

When an acquisition starts, there are some checks and operations to be done by the software, depending on the trigger and acquisition settings set by the user, before apply the result s to the corresponding registers in the FPGA. Things like the FIFO size, the delay value to apply, the trigger to store ID in the FPGA memory or the mask to store ID in it are some of those operations that need to be carried out before starting an acquisition.

Apart from that, each data stored in the FPGA memory contains three fields: timestamp, data and ID of the module which has generated the data. These ID's are not fixed and is the main software who dynamically assigns and ID value to each module. So one of the task to do before starting an acquisition, is to set the corresponding ID's to be used by each module. If there is no ID value assigned to a module output, then that module will not send messages through the harmony bus and it will be not possible to store its data to the FPGA memory.

For this new ALBA Em# we are using fixed ID's value in code, but for future developments, these ID values could be defined in the config file.

B.1 Settings for software trigger and full acquisition in FIFO FPGA

This section includes the different settings to apply when the acquisition trigger mode is set to software and when the acquisition time is smaller than the maximum acquisition time allowed by the FIFO FPGA size, which is 320ms.

FPGA Block	Register ➔ Value	Description
ADCCORE	ID_CH1_ID ➔ 0 ID_CH2_ID ➔ 0 ID_CH3_ID ➔ 0 ID_CH4_ID ➔ 0	Stop sending ADC data through the Harmony bus ADC output.
MEMORY	CTL_SAVE ➔ 0x00 CTL_ONTRIGGER ➔ 0x00	Data saving in FPGA memory, disabled.
AVERAGE_1	AVG_CTL_ENA ➔ 0x03 AVG_CTL_PRE_DIV ➔ 0x00 ID_CFG_ID_OUT ➔ **AVG_ID_1** ID_CFG_ID_IN ➔ **FIFO_ID_1** ID_CFG_ID_IP ➔ **ADC_CH_ID_1**	Configure average modules for the four channels in the same way. Average output for each channel is generated after receiving both negative and positive inputs, without shifting the output. Configure positive input as the data coming from the ADC and the negative input is the data coming form the FIFO. The Average output ID is fixed and is assigned from a table inside the NUC software, but can be defined in the config file.
AVERAGE_2	AVG_CTL_ENA ➔ 0x03 AVG_CTL_PRE_DIV ➔ 0x00 ID_CFG_ID_OUT ➔ **AVG_ID_2** ID_CFG_ID_IN ➔ **FIFO_ID_2** ID_CFG_ID_IP ➔ **ADC_CH_ID_2**	
AVERAGE_3	AVG_CTL_ENA ➔ 0x03 AVG_CTL_PRE_DIV ➔ 0x00 ID_CFG_ID_OUT ➔ **AVG_ID_3** ID_CFG_ID_IN ➔ **FIFO_ID_3** ID_CFG_ID_IP ➔ **ADC_CH_ID_3**	
AVERAGE_4	AVG_CTL_ENA ➔ 0x03 AVG_CTL_PRE_DIV ➔ 0x00 ID_CFG_ID_OUT ➔ **AVG_ID_4** ID_CFG_ID_IN ➔ **FIFO_ID_4** ID_CFG_ID_IP ➔ **ADC_CH_ID_4**	
FIFO_1	MEM_CTL_ID_IN ➔ **ADC_CH_ID_1** MEM_CTL_ID_OUT ➔ **FIFO_ID_1** FIFO_SIZE_SIZE ➔ **fifosize**	Configure And Initialize FIFO modules for the 4 acquisition channels. The FIFO needs to be cleared first. To do that, FIFO_MAX_SIZE data, with value 0x00 is generated using
FIFO_2	MEM_CTL_ID_IN ➔ **ADC_CH_ID_2** MEM_CTL_ID_OUT ➔ **FIFO_ID_2**	

	FIFO_SIZE_SIZE ➔ **fifosize**	the IDGEN module . The FIFO IN is configured with the same ID than is being generated with the ID GEN. After this initialization, the FIFO input ID is the one coming from the ADC. The FIFO output ID, same as the average ID is fixed by software, but can be externally defined in the config file The FIFO size is calculated by dividing the acquisition time set between the acquisition need for one sample. The result is the number of samples to sore in the FIFO. In this configuration mode, the number of samples doesn't exceeds the FIFO MAXIMUM SIZE.
FIFO_3	MEM_CTL_ID_IN ➔ **ADC_CH_ID_3** MEM_CTL_ID_OUT ➔ **FIFO_ID_3** FIFO_SIZE_SIZE ➔ **fifosize**	
FIFO_4	MEM_CTL_ID_IN ➔ **ADC_CH_ID_4** MEM_CTL_ID_OUT ➔ **FIFO_ID_4** FIFO_SIZE_SIZE ➔ **fifosize**	
DIGITALIO	CNT0_STP_OUT_ID ➔ 0x00 CNT0_STP_SRC_ID ➔ **IDGEN_ID** CNT0_STP_SIG ➔ 0x00 CNT0_STP_DIS ➔ 0x07 CNT0_V_CNT_V ➔ 0x00 CNT0_STP_SRC ➔ 0x0E TRG0_V_CNT_V ➔ 0x01	Configure Counter 0 to count the number of IDGEN ID received. Counter 0 is enabled and set to count Harmony ID's which value is IDGEN ID
IDGEN	ID_GEN_CTL_ETI ➔ 0x00 ID_GEN_CTL_CLKE ➔ 0x01 DATA_GEN_DATA ➔ 0x00 TS_ID_ID_OUT ➔ **IDGEN_ID** WAIT_TIME ➔ **delay_value**	IDGEN module is configured to send a data with value 0x00 and ID the IDGEN_ID each time the MTRIG register is set to 1. The calculated trigger delay value is also applied and generated with the IDGEN module, by delaying the ID out which triggers the Memory data save.
MEMORY	CTL_TRIGID ➔ **IDGEN_ID** CTL_IDMASK ➔ **maskval** CTL_ID ➔ **AVG_ID_1**	Memory trigger and mask are configured. The resulting number is combination of the four AVERAGE channels Id, which contains the ADC data we want to store. The trigger to store data corresponds to the IDGEN ID module. The **maskval** is the result of an or operation between **AVG_ID_1**, **AVG_ID_2**, **AVG_ID_3** and **AVG_ID_4**
AVERAGE_1	AVG_CTL_WB_RST ➔ 0x01	Before enabling the ADC module, the AVERAGE modules need to be reset.
AVERAGE_2	AVG_CTL_WB_RST ➔ 0x01	
AVERAGE_3	AVG_CTL_WB_RST ➔ 0x01	
AVERAGE_4	AVG_CTL_WB_RST ➔ 0x01	
MEMORY	CTL_ONTRIGGER ➔ 0x01 CTL_RST ➔ 0x01 CTL_SAVE ➔ 0x01	Enable data saving in FPGA memory
ADCCORE	CTL_RST ➔ 0x01 CTL_ADQ_M ➔ 0x00 CTL_RANGE ➔ 0x01 CTL_ADC_OS ➔ 0x06 CTL_DBL_RATE ➔ 0x01 ID_CH1_ID ➔ **ADC_CH_ID_1** ID_CH2_ID ➔ **ADC_CH_ID_2** ID_CH3_ID ➔ **ADC_CH_ID_3** ID_CH4_ID ➔ **ADC_CH_ID_4**	Configure de ADC module to use oversampling of 64 values, working at 200KS/s and in a continuous acquisition mode. ADC channels output to the harmony bus are enabled by setting the corresponding value to each channel ID. These Id are defined in software, but same as the rest of ID's, can be defined in the config file.
DIGITALIO	DIO_STP_RST ➔ 0x1	Just in case. Reset the digitalIO module.

B.2 Settings for software trigger and partial acquisition in FIFO FPGA

This section includes the different settings to apply when the acquisition trigger mode is set to software and when the acquisition time is bigger than the maximum acquisition time allowed by the FIFO FPGA size, which is 320ms. In this case, we have to split the acquisition in several equal parts in the FPGA, then store those partial acquisitions in the FPGA memory, read them in the NUC software and calculate the average of those acquisitions to get the final acquisition data. In example, for an acquisition of 1000ms, the software calculates that it needs 4 partial acquisitions with a FIFO size of 781. So the FIFO is configured to get those 781 samples and calculate the average, the resulting value is stored in the FPGA memory. The main software gets up to 4 of these partial acquisitions and calculates the average to complete the requested acquisition.

FPGA Block	Register ➔ Value	Description
ADCCORE	ID_CH1_ID ➔ 0 ID_CH2_ID ➔ 0 ID_CH3_ID ➔ 0 ID_CH4_ID ➔ 0	Stop sending ADC data through the Harmony bus ADC output.
MEMORY	CTL_SAVE ➔ 0x00 CTL_ONTRIGGER ➔ 0x00	Data saving in FPGA memory, disabled.
AVERAGE_1	AVG_CTL_ENA ➔ 0x03 AVG_CTL_PRE_DIV ➔ 0x00 ID_CFG_ID_OUT ➔ **AVG_ID_1** ID_CFG_ID_IN ➔ **FIFO_ID_1** ID_CFG_ID_IP ➔ **ADC_CH_ID_1**	Configure average modules for the four channels in the same way. Average output for each channel is generated after receiving both negative and positive inputs, without shifting the output. Configure positive input as the data coming from the ADC and the negative input is the data coming form the FIFO. The Average output ID is fixed and is assigned from a table inside the NUC software, but can be defined in the config file.
AVERAGE_2	AVG_CTL_ENA ➔ 0x03 AVG_CTL_PRE_DIV ➔ 0x00 ID_CFG_ID_OUT ➔ **AVG_ID_2** ID_CFG_ID_IN ➔ **FIFO_ID_2** ID_CFG_ID_IP ➔ **ADC_CH_ID_2**	
AVERAGE_3	AVG_CTL_ENA ➔ 0x03 AVG_CTL_PRE_DIV ➔ 0x00 ID_CFG_ID_OUT ➔ **AVG_ID_3** ID_CFG_ID_IN ➔ **FIFO_ID_3** ID_CFG_ID_IP ➔ **ADC_CH_ID_3**	
AVERAGE_4	AVG_CTL_ENA ➔ 0x03 AVG_CTL_PRE_DIV ➔ 0x00 ID_CFG_ID_OUT ➔ **AVG_ID_4** ID_CFG_ID_IN ➔ **FIFO_ID_4** ID_CFG_ID_IP ➔ **ADC_CH_ID_4**	
FIFO_1	MEM_CTL_ID_IN ➔ **ADC_CH_ID_1** MEM_CTL_ID_OUT ➔ **FIFO_ID_1** FIFO_SIZE_SIZE ➔ **fifosize**	Configure And Initialize FIFO modules for the 4 acquisition channels. The FIFO needs to be cleared first. To do that, FIFO_MAX_SIZE data, with value 0x00 is generated using the IDGEN module . The FIFO IN is configured with the same ID than is being generated with the ID GEN. After this initialization, the FIFO input ID is the one coming from the ADC. The FIFO output ID, same as the average ID is fixed by software, but can be externally defined in the config file The FIFO size is calculated by dividing the acquisition time set between the acquisition need for one sample. The result is the number of samples to sore in the FIFO. In this configuration mode, the number of samples doesn't exceeds the FIFO MAXIMUM SIZE.
FIFO_2	MEM_CTL_ID_IN ➔ **ADC_CH_ID_2** MEM_CTL_ID_OUT ➔ **FIFO_ID_2** FIFO_SIZE_SIZE ➔ **fifosize**	
FIFO_3	MEM_CTL_ID_IN ➔ **ADC_CH_ID_3** MEM_CTL_ID_OUT ➔ **FIFO_ID_3** FIFO_SIZE_SIZE ➔ **fifosize**	
FIFO_4	MEM_CTL_ID_IN ➔ **ADC_CH_ID_4** MEM_CTL_ID_OUT ➔ **FIFO_ID_4** FIFO_SIZE_SIZE ➔ **fifosize**	

DIGITALIO	CNT0_STP_OUT_ID → 0x00 CNT0_STP_SRC_ID → **IDGEN_ID** CNT0_STP_SIG → 0x00 CNT0_STP_DIS → 0x07 CNT0_V_CNT_V → 0x00 CNT0_STP_SRC → 0x0E TRG0_V_CNT_V → 0x01	Configure Counter 0 to count the number of IDGEN ID received. Counter 0 is enabled and set to count Harmony ID's which value is IDGEN ID
IDGEN	ID_GEN_CTL_ETI → 0x00 ID_GEN_CTL_CLKE → 0x01 DATA_GEN_DATA → 0x00 TS_ID_ID_OUT → **IDGEN_ID** WAIT_TIME → **delay_value**	IDGEN module is configured to send a data with value 0x00 and ID the IDGEN_ID each time the MTRIG register is set to 1. The calculated trigger delay value is also applied and generated with the IDGEN module, by delaying the ID out which triggers the Memory data save.
DIGITALIO	CNT1_STP_OUT_ → **CNT1_ID** CNT1_STP_SRC_ID → **ADC_CH_ID_1** CNT1_STP_SIG → 0x00 CNT1_STP_TRIG → 0x01 CNT1_STP_SRC → 0x0E CNT1_V_CNT_V → 0x00 TRG1_V_CNT_V → **fifosize** DIS_STP_DIS1_ID → **IDGEN_ID** CNT1_STP_DIS → 0x01 DIS_STP_DIS1_M → 0x03 DIS_STP_WD1 → 0x01 DIS_STP_DIS1_M → 0x02	Configure Counter 1 to count ADC1 samples generated in the harmony bus, when n the number of samples matches the value of the **fifosize** set, then it generates the corresponding out ID set, which is in this case the **CNT1_ID.** This counter is disabled at the beginning of the acquisition and starts to count when the **IDGEN_ID** is received.
MEMORY	CTL_TRIGID → **CNT1_ID** CTL_IDMASK → **maskval** CTL_ID → **AVG_ID_1**	Memory trigger and mask are configured. The resulting number is combination of the four AVERAGE channels Id, which contains the ADC data we want to store. The trigger to store data corresponds to the output of the Counter 1. The **maskval** is the result of an or operation between **AVG_ID_1**, **AVG_ID_2**, **AVG_ID_3**, **AVG_ID_4**, **IDGEN_ID** and **CNT1_ID**
AVERAGE_1	AVG_CTL_WB_RST → 0x01	Before enabling the ADC module, the AVERAGE modules need to be reset.
AVERAGE_2	AVG_CTL_WB_RST → 0x01	
AVERAGE_3	AVG_CTL_WB_RST → 0x01	
AVERAGE_4	AVG_CTL_WB_RST → 0x01	
MEMORY	CTL_ONTRIGGER → 0x01 CTL_RST → 0x01 CTL_SAVE → 0x01	Enable data saving in FPGA memory
ADCCORE	CTL_RST → 0x01 CTL_ADQ_M → 0x00 CTL_RANGE → 0x01 CTL_ADC_OS → 0x06 CTL_DBL_RATE → 0x01 ID_CH1_ID → **ADC_CH_ID_1** ID_CH2_ID → **ADC_CH_ID_2** ID_CH3_ID → **ADC_CH_ID_3** ID_CH4_ID → **ADC_CH_ID_4**	Configure de ADC module to use oversampling of 64 values, working at 200KS/s and in a continuous acquisition mode. ADC channels output to the harmony bus are enabled by setting the corresponding value to each channel ID. These Id are defined in software, but same as the rest of ID's, can be defined in the config file.
DIGITALIO	DIO_STP_RST → 0x1	Just in case. Reset the digitalIO module.

B.3 Settings for hardware trigger and full acquisition in FIFO FPGA

This section includes the different settings to apply when the acquisition trigger mode is set to hardware and when the acquisition time is smaller than the maximum acquisition time allowed by the FIFO FPGA size, which is 320ms. Basically configuration is the same as explained in section B.1, but with the different here that instead of manual generate the trigger with the IDGEN module, is the trigger input selected which triggers the IDGEN module, through the DigitalIO counter 0, to generate the ID that triggers the FPGA memory data saving.

FPGA Block	Register ➔ Value	Description
ADCCORE	ID_CH1_ID ➔ 0 ID_CH2_ID ➔ 0 ID_CH3_ID ➔ 0 ID_CH4_ID ➔ 0	Stop sending ADC data through the Harmony bus ADC output.
MEMORY	CTL_SAVE ➔ 0x00 CTL_ONTRIGGER ➔ 0x00	Data saving in FPGA memory, disabled.
AVERAGE_1	AVG_CTL_ENA ➔ 0x03 AVG_CTL_PRE_DIV ➔ 0x00 ID_CFG_ID_OUT ➔ **AVG_ID_1** ID_CFG_ID_IN ➔ **FIFO_ID_1** ID_CFG_ID_IP ➔ **ADC_CH_ID_1**	Configure average modules for the four channels in the same way. Average output for each channel is generated after receiving both negative and positive inputs, without shifting the output. Configure positive input as the data coming from the ADC and the negative input is the data coming form the FIFO. The Average output ID is fixed and is assigned from a table inside the NUC software, but can be defined in the config file.
AVERAGE_2	AVG_CTL_ENA ➔ 0x03 AVG_CTL_PRE_DIV ➔ 0x00 ID_CFG_ID_OUT ➔ **AVG_ID_2** ID_CFG_ID_IN ➔ **FIFO_ID_2** ID_CFG_ID_IP ➔ **ADC_CH_ID_2**	
AVERAGE_3	AVG_CTL_ENA ➔ 0x03 AVG_CTL_PRE_DIV ➔ 0x00 ID_CFG_ID_OUT ➔ **AVG_ID_3** ID_CFG_ID_IN ➔ **FIFO_ID_3** ID_CFG_ID_IP ➔ **ADC_CH_ID_3**	
AVERAGE_4	AVG_CTL_ENA ➔ 0x03 AVG_CTL_PRE_DIV ➔ 0x00 ID_CFG_ID_OUT ➔ **AVG_ID_4** ID_CFG_ID_IN ➔ **FIFO_ID_4** ID_CFG_ID_IP ➔ **ADC_CH_ID_4**	
FIFO_1	MEM_CTL_ID_IN ➔ **ADC_CH_ID_1** MEM_CTL_ID_OUT ➔ **FIFO_ID_1** FIFO_SIZE_SIZE ➔ **fifosize**	Configure And Initialize FIFO modules for the 4 acquisition channels. The FIFO needs to be cleared first. To do that, FIFO_MAX_SIZE data, with value 0x00 is generated using the IDGEN module . The FIFO IN is configured with the same ID than is being generated with the ID GEN. After this initialization, the FIFO input ID is the one coming from the ADC. The FIFO output ID, same as the average ID is fixed by software, but can be externally defined in the config file The FIFO size is calculated by dividing the acquisition time set between the acquisition need for one sample. The result is the number of samples to sore in the FIFO. In this configuration mode, the number of samples doesn't exceeds the FIFO MAXIMUM SIZE.
FIFO_2	MEM_CTL_ID_IN ➔ **ADC_CH_ID_2** MEM_CTL_ID_OUT ➔ **FIFO_ID_2** FIFO_SIZE_SIZE ➔ **fifosize**	
FIFO_3	MEM_CTL_ID_IN ➔ **ADC_CH_ID_3** MEM_CTL_ID_OUT ➔ **FIFO_ID_3** FIFO_SIZE_SIZE ➔ **fifosize**	
FIFO_4	MEM_CTL_ID_IN ➔ **ADC_CH_ID_4** MEM_CTL_ID_OUT ➔ **FIFO_ID_4** FIFO_SIZE_SIZE ➔ **fifosize**	
DIGITALIO	DIGITALIO ➔ **trigger_input** DIO_STP_IOP ➔ **trigger_polarity**	Configures the selected trigger input and its polarity
SPI	For GPIO: ctl_spi_addr ➔ 0x18	Configures the front end board port expanders to set the

	ctl_spi_addr ➔ 0x19 For High Speed I/O ports: ctl_spi_addr ➔ 0x1a Other register used to transmit/receive data though SPI: ctl_tx, ctl_tx_data, ctl_tx_data_length, ctl_rx, ctl_rx_data_length, ctl_start, ctl_rx_data	corresponding trigger input as input.
DIGITALIO	CNT0_STP_OUT_ID ➔ **CNT0_ID** CNT0_STP_SIG ➔ 0x00 CNT0_STP_DIS ➔ 0x07 CNT0_STP_TRIG ➔ 0x00 CNT0_STP_SRC ➔ **trigger_input** TRG0_V_CNT_V ➔ 0x01 CNT0_V_CNT_V ➔ 0x00	Confiure Counter 0 to count voltage changes in the selected trigger input. The trigger counter is set to 1, so once a trigger input is received the corresponding **CNT0_ID** is generated.
IDGEN	ID_GEN_CTL_ETI ➔ 0x01 ID_GEN_CTL_CLKE ➔ 0x01 DATA_GEN_DATA ➔ 0x00 ID_GEN_CTL_TRIGGERID ➔ **CNT0_ID** TS_ID_ID_OUT ➔ **IDGEN_ID** WAIT_TIME ➔ **delay_value**	The configuration of the ID GEN module here is differente to the one when using software trigger. The IDGEN module is configured to send a data with value 0x00 and ID the **IDGEN_ID** each time the **CNT0_ID** is received The calculated trigger delay value is also applied and generated with the IDGEN module, by delaying the ID out which triggers the Memory data save.
MEMORY	CTL_TRIGID ➔ **IDGEN_ID** CTL_IDMASK ➔ **maskval** CTL_ID ➔ **AVG_ID_1**	Memory trigger and mask are configured. The resulting number is combination of the four AVERAGE channels Id, which contains the ADC data we want to store. The trigger to store data corresponds to the IDGEN ID module. The **maskval** is the result of an or operation between **AVG_ID_1**, **AVG_ID_2**, **AVG_ID_3** and **AVG_ID_4**
AVERAGE_1	AVG_CTL_WB_RST ➔ 0x01	Before enabling the ADC module, the AVERAGE modules need to be reset.
AVERAGE_2	AVG_CTL_WB_RST ➔ 0x01	
AVERAGE_3	AVG_CTL_WB_RST ➔ 0x01	
AVERAGE_4	AVG_CTL_WB_RST ➔ 0x01	
MEMORY	CTL_ONTRIGGER ➔ 0x01 CTL_RST ➔ 0x01 CTL_SAVE ➔ 0x01	Enable data saving in FPGA memory
ADCCORE	CTL_RST ➔ 0x01 CTL_ADQ_M ➔ 0x00 CTL_RANGE ➔ 0x01 CTL_ADC_OS ➔ 0x06 CTL_DBL_RATE ➔ 0x01 ID_CH1_ID ➔ **ADC_CH_ID_1** ID_CH2_ID ➔ **ADC_CH_ID_2** ID_CH3_ID ➔ **ADC_CH_ID_3** ID_CH4_ID ➔ **ADC_CH_ID_4**	Configure the ADC module to use oversampling of 64 values, working at 200KS/s and in a continuous acquisition mode. ADC channels output to the harmony bus are enabled by setting the corresponding value to each channel ID. These Id are defined in software, but same as the rest of ID's, can be defined in the config file.
DIGITALIO	DIO_STP_RST ➔ 0x1	Just in case. Reset the digitalIO module.

B.4 Settings for hardware trigger and partial acquisition in FIFO FPGA

Again this section is same as explained in section B.2 but with the different here that instead of manual generate the trigger with the IDGEN module, is the trigger input selected which triggers the IDGEN module, through the DigitalIO counter 0, to generate the ID that triggers the FPGA memory data saving.

FPGA Block	Register ➔ Value	Description
ADCCORE	ID_CH1_ID ➔ 0 ID_CH2_ID ➔ 0 ID_CH3_ID ➔ 0 ID_CH4_ID ➔ 0	Stop sending ADC data through the Harmony bus ADC output.
MEMORY	CTL_SAVE ➔ 0x00 CTL_ONTRIGGER ➔ 0x00	Data saving in FPGA memory, disabled.
AVERAGE_1	AVG_CTL_ENA ➔ 0x03 AVG_CTL_PRE_DIV ➔ 0x00 ID_CFG_ID_OUT ➔ **AVG_ID_1** ID_CFG_ID_IN ➔ **FIFO_ID_1** ID_CFG_ID_IP ➔ **ADC_CH_ID_1**	Configure average modules for the four channels in the same way. Average output for each channel is generated after receiving both negative and positive inputs, without shifting the output. Configure positive input as the data coming from the ADC and the negative input is the data coming form the FIFO. The Average output ID is fixed and is assigned from a table inside the NUC software, but can be defined in the config file.
AVERAGE_2	AVG_CTL_ENA ➔ 0x03 AVG_CTL_PRE_DIV ➔ 0x00 ID_CFG_ID_OUT ➔ **AVG_ID_2** ID_CFG_ID_IN ➔ **FIFO_ID_2** ID_CFG_ID_IP ➔ **ADC_CH_ID_2**	
AVERAGE_3	AVG_CTL_ENA ➔ 0x03 AVG_CTL_PRE_DIV ➔ 0x00 ID_CFG_ID_OUT ➔ **AVG_ID_3** ID_CFG_ID_IN ➔ **FIFO_ID_3** ID_CFG_ID_IP ➔ **ADC_CH_ID_3**	
AVERAGE_4	AVG_CTL_ENA ➔ 0x03 AVG_CTL_PRE_DIV ➔ 0x00 ID_CFG_ID_OUT ➔ **AVG_ID_4** ID_CFG_ID_IN ➔ **FIFO_ID_4** ID_CFG_ID_IP ➔ **ADC_CH_ID_4**	
FIFO_1	MEM_CTL_ID_IN ➔ **ADC_CH_ID_1** MEM_CTL_ID_OUT ➔ **FIFO_ID_1** FIFO_SIZE_SIZE ➔ **fifosize**	Configure And Initialize FIFO modules for the 4 acquisition channels. The FIFO needs to be cleared first. To do that, FIFO_MAX_SIZE data, with value 0x00 is generated using the IDGEN module . The FIFO IN is configured with the same ID than is being generated with the ID GEN. After this initialization, the FIFO input ID is the one coming from the ADC. The FIFO output ID, same as the average ID is fixed by software, but can be externally defined in the config file The FIFO size is calculated by dividing the acquisition time set between the acquisition need for one sample. The result is the number of samples to sore in the FIFO. In this configuration mode, the number of samples doesn't exceeds the FIFO MAXIMUM SIZE.
FIFO_2	MEM_CTL_ID_IN ➔ **ADC_CH_ID_2** MEM_CTL_ID_OUT ➔ **FIFO_ID_2** FIFO_SIZE_SIZE ➔ **fifosize**	
FIFO_3	MEM_CTL_ID_IN ➔ **ADC_CH_ID_3** MEM_CTL_ID_OUT ➔ **FIFO_ID_3** FIFO_SIZE_SIZE ➔ **fifosize**	
FIFO_4	MEM_CTL_ID_IN ➔ **ADC_CH_ID_4** MEM_CTL_ID_OUT ➔ **FIFO_ID_4** FIFO_SIZE_SIZE ➔ **fifosize**	
DIGITALIO	DIGITALIO ➔ **trigger_input** DIO_STP_IOP ➔ **trigger_polarity**	Configures the selected trigger input and its polarity
SPI	For GPIO: ctl_spi_addr ➔ 0x18 ctl_spi_addr ➔ 0x19	Configures the front end board port expanders to set the corresponding trigger input as input.

	For High Speed I/O ports: ctl_spi_addr ➔ 0x1a Other register used to transmit/receive data though SPI: ctl_tx, ctl_tx_data, ctl_tx_data_length, ctl_rx, ctl_rx_data_length, ctl_start, ctl_rx_data	
DIGITALIO	CNT0_STP_OUT_ID ➔ **CNT0_ID** CNT0_STP_SIG ➔ 0x00 CNT0_STP_DIS ➔ 0x07 CNT0_STP_TRIG ➔ 0x00 CNT0_STP_SRC ➔ **trigger_input** TRG0_V_CNT_V ➔ 0x01 CNT0_V_CNT_V ➔ 0x00	Confiure Counter 0 to count voltage changes in the selected trigger input. The trigger counter is set to 1, so once a trigger input is received the corresponding **CNT0_ID** is generated.
IDGEN	ID_GEN_CTL_ETI ➔ 0x01 ID_GEN_CTL_CLKE ➔ 0x01 DATA_GEN_DATA ➔ 0x00 ID_GEN_CTL_TRIGGERID ➔ **CNT0_ID** TS_ID_ID_OUT ➔ **IDGEN_ID** WAIT_TIME ➔ **delay_value**	The configuration of the ID GEN module here is differente to the one when using software trigger. The IDGEN module is configured to send a data with value 0x00 and ID the **IDGEN_ID** each time the **CNT0_ID** is received The calculated trigger delay value is also applied and generated with the IDGEN module, by delaying the ID out which triggers the Memory data save.
DIGITALIO	CNT1_STP_OUT_ ➔ **CNT1_ID** CNT1_STP_SRC_ID ➔ **ADC_CH_ID_1** CNT1_STP_SIG ➔ 0x00 CNT1_STP_TRIG ➔ 0x01 CNT1_STP_SRC ➔ 0x0E CNT1_V_CNT_V ➔ 0x00 TRG1_V_CNT_V ➔ **fifosize** DIS_STP_DIS1_ID ➔ **IDGEN_ID** CNT1_STP_DIS ➔ 0x01 DIS_STP_DIS1_M ➔ 0x03 DIS_STP_WD1 ➔ 0x01 DIS_STP_DIS1_M ➔ 0x02	Configure Counter 1 to count ADC1 samples generated in the harmony bus, when n the number of samples matches the value of the **fifosize** set, then it generates the corresponding out ID set, which is in this case the **CNT1_ID.** This counter is disabled at the beginning of the acquisition and starts to count when the **IDGEN_ID** is received.
MEMORY	CTL_TRIGID ➔ **CNT1_ID** CTL_IDMASK ➔ **maskval** CTL_ID ➔ **AVG_ID_1**	Memory trigger and mask are configured. The resulting number is combination of the four AVERAGE channels Id, which contains the ADC data we want to store. The trigger to store data corresponds to the output of the Counter 1. The **maskval** is the result of an or operation between **AVG_ID_1**, **AVG_ID_2**, **AVG_ID_3**, **AVG_ID_4**, **IDGEN_ID** and **CNT1_ID**
AVERAGE_1	AVG_CTL_WB_RST ➔ 0x01	Before enabling the ADC module, the AVERAGE modules need to be reset.
AVERAGE_2	AVG_CTL_WB_RST ➔ 0x01	
AVERAGE_3	AVG_CTL_WB_RST ➔ 0x01	
AVERAGE_4	AVG_CTL_WB_RST ➔ 0x01	
MEMORY	CTL_ONTRIGGER ➔ 0x01 CTL_RST ➔ 0x01 CTL_SAVE ➔ 0x01	Enable data saving in FPGA memory
ADCCORE	CTL_RST ➔ 0x01 CTL_ADQ_M ➔ 0x00 CTL_RANGE ➔ 0x01	Configure de ADC module to use oversampling of 64 values, working at 200KS/s and in a continuous acquisition mode.

	CTL_ADC_OS ➔ 0x06 CTL_DBL_RATE ➔ 0x01 ID_CH1_ID ➔ ADC_CH_ID_1 ID_CH2_ID ➔ ADC_CH_ID_2 ID_CH3_ID ➔ ADC_CH_ID_3 ID_CH4_ID ➔ ADC_CH_ID_4	ADC channels output to the harmony bus are enabled by setting the corresponding value to each channel ID. These Id are defined in software, but same as the rest of ID's, can be defined in the config file.
DIGITALIO	DIO_STP_RST ➔ 0x1	Just in case. Reset the digitalIO module.