Alibava GUI

Alibava Systems

Table of Contents

Introduction.

`alibava-gui` is a graphical user interface that controls the ALIBAVA card. It is able to configure the device, receive the data that the card sends via the USB bus and store it in a file for further analysis. `alibava-gui` also monitors the data while in acquisition mode so that the user can detect problems or just find the proper parameters to run the system in an optimal way.

What is alibava-gui?

The alibava firmware provides 5 run modes

- Pedestals: makes a pedestal run. Alibava generates an internal trigger that will allow to compute the baseline or pedestals and its variation (the noise)
- Calibration: makes a calibration run. Alibava programs the Beetle chips to inject calibration pulses to all the channels in order to charactirize the electrical behaviour of the ASICs.
- Laser synchronization: Alibava is able to send a pulse that can be used to trigger a laser system. This run mode scans the delay between the pulse sent by alibava and the acquisition so that the system will sample at the maximum of the signal produced by the laser.
- Laser: makes a laser run. One needs to run the in laser synchronization mode before in order to read back the optimal signal produced by the laser.
- Source: makes a run in which the acquisition is triggered by signals above the threshold in the input connectors.

Figure 1 shows the main window

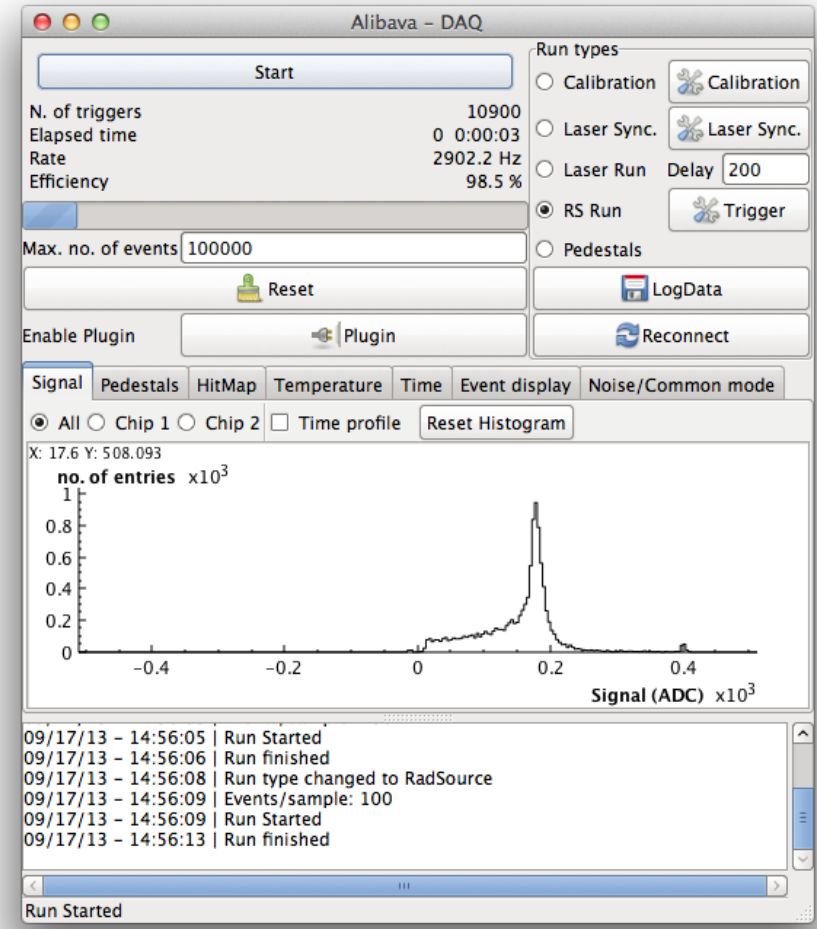

Figure 1. Alibava main window

As one can see, the run types are selected on the right hand side of the window. Right by the run type name there are buttons that, when clicked, will open a dialog window to configure the run parameters. All the settings can be stored in a configuration file by clicking save in the File menu. One can load different configurations clicking on Open in the File menu. There are more configuration settings that can be set by clicking on the different items of the Settings menu.

On the left hand side you can see the DAQ button. This button starts or stops the acquisition. You can monitor the number of acquired events, the elapsed time, the acquisition rate and the efficiency.The rate is integrated in 1 second time windows, so expect 0 values while the data is being read out from the USB port. The efficiency is the number of events that pass the criteria defined in the Analysis dialog window (the Section called *Analysis configuration*).

In the middle of the window there is a collection of tabs that will allow to monitor the data during the data acquisition and on some of the tabs one can find buttons that will refine the information displayed on the histograms.

`alibava-gui` also provides the possibility to load user defined plugins that will allow to perform non-standard actions at different stages of the acquisition process.

Those plugins can be written both in C++, as shared libraries, or in Python, as normal Python scripts. However, the plugin is not active by default. In order to activate or deactivate it one needs to toggle the state of the button named Plugin in the main window (See Figure 1).

Finally, note the Reconnect button. Sometimes you unplug the alibava system from the USB plug without quitting alibava-gui. When that happens you shoul dpress this button. It will close any open device and reopen it.

Starting alibava

Setting up the environment

Windows and Mac OSX operative systems are not very restrictive with permissions and one can usually access the serial ports with any particualr effort. Linux systems, however, are a bit mor restictive and it may happen that one does not have the access rights to read or write in the the serial port. In modern Linus distributions it is enough to belong to the dialout group in order to have access to /dev/ttyUSBn device file. If your user does not belong to that group then you will have to add it. To check if you are already in the dialout group type

```
id
```

on the terminal.

Older distributions of Linux used the udev package. When installing `alibava-gui` as a super user, a new group will be created named alibava. Also a new udev rule will be added granting read/write permissions to the members of that group. In order to grant any user with read/write permissions on the USB bus you will have to make him/her a member of the alibava group. This must be made as super user by typing the following

```
/usr/sbin/usermod -aG alibava {your user name}
```

For that to work the installation should be made as super user. If you installed `alibava-gui` withour root privilegues, then you will have to create the alibava group and install the udev rules manually as root by typing

```
/usr/sbin/groupadd -f alibava
```

followed by the execution the script **install-udev.sh**. The script can be found on the top folder of the distribution bundle.

To check that the installation has been done properly, plug in an Alibava card on your USB hub and check that there is a file called `/dev/alibava0`. If this is so the udev rules are properly installed and the members of the alibava group will be able to read from and write to the device. Also, alibava-gui will automatically detect in which port the Alibava card is connected. If alibava-gui is not able to figure out that, it will quit unless you force the program to try to open another device at the command line (see the Section called *How to launch the program*).

Warning

In some versions of Linux, the udev rules defined in install-udev.sh do not work and one can just add the user to the dialout group with the same command as for the alibava group.

Old `alibava-gui` versions

In alibava-gui version older than 0.1.6-3 the program was not able to detect the Alibava card and no udev rules were provided. In that case one was forced to do things manually. In order to allow alibava-gui to read from and write to the device there were a number of steps to follow which are explained below.

When the Alibava card is plugged in the computer, the driver decides which port to use and alibava-gui did not have any means of discovering which one it was in an automatic way.

After plugging in the card, one should type

```
dmesg
```

and look for the port that the driver has selected. The name is usually /dev/ttyUSBn where n is 0 most of the times. Another problem encountered quite often is that a normal user does not have read/write permissions. To solve that you should type

```
change_priv n
```

where n is the number you found for /dev/ttyUSBn. If it is 0, you do not need to specify it.

Warning

`change_priv` needs super user permissions. That means that you should install alibava as super user. This will make the program run with superuser attributes even if you launch it from within your account.

How to launch the program

Once this is done, one launches the alibava application by typing

```
alibava-gui [options] [config_file]
```

where config_file a file where all the settings have been saved. The options can be

Table 1. alibava-gui options

--gui	Shown the main GUI. This is the default
--no-gui	The program runs without a GUI
--emulator	Simulates (Emulates) the data. Useful to get familiar with the application
--nevts=n	Set n as the maximum number of events in the run
--sample=n	Number of events to acquire in the motherboard before transmitting the data to the PC

--dev=/dev/ttyUSBn	Set /dev/ttyUSBn as the port to communicate with the motherboard in case n is not the usual 0
--out=out_file_name	Sets the path and name of the output data file
--pedestal	Acquire a Pedestal run when in no-gui mode
--source	Acquire a RadSource run when in non-gui mode. Other required settings should be provided through a configuration file.
--calib	Acquire a Calibration run when in non-gui mode. Other required settings should be provided through a configuration file.
--laser	Acquire a Laser run when in non-gui mode. Other required settings should be provided through a configuration file.
--activate-plugin	Activates the plugin if defined in the configuration file.
--firmware=n	Forces a given version of the firmware.
--soap-port=nn	port number ofr the alibava-gui soap server

In general, when running in GUI mode, all the options listed above can be set in the various dilaogs. One could, for instance give some of the options at the start of alibava-gui to have those values by default at the beginning. When running in non GUI mode options need to be provided to change the default values. Note also that some of the options in the table above assume that there is some more information in a configuration file provided at the command line. Most of the times the default values of the program will not procuded the expected effect. This is particularly true for the scan parameters needed when making calibration runs.

Warning

Older versions of the `alibava-gui` program (before 0.4.0-1) used to assume that the firmware version was 0. If this is not the case, when running in non-gui mode one has to specify the firmware version. There are currently version 0 (the very first), version 1 and version 2, which is handled form `alibava-gui` version 0.4.0-1.

Taking data

Taking data is easy. Just select the run type, set it up properly and click on the DAQ button (the one named Start in Figure 1). Now, if you want to store the data for further analysis, you have to press the Log Data button. A dialog window will pop up where you can select the name of the output file. When starting the run by clicking start the data will be dumped into the data file.

The following sections describe the different run types

Calibration run

The calibration run has 2 main parameters: the strobe delay and the amplitude of the test pulse. At the moment one can only scan one of those parameters at a time.

The strobe delay sets the delay between the strobe signal that generates the calibration pulse and the clock rising edge of the beetle 128th slot of the pipeline. This allows to reconstruct the pulse shape. The amplitude controls the amplitude of the calibration pulse so that one can get the gain and offset of the characteristic curve of the preamplifier in all the channels. Note that to find the proper value of the gain one has to find the strobe delay where the signa reaches the maximum value.

Figure 2 shows the main parameters that can be set to define the calibration scan. As already mentioned, only the charge (with a fixed delay) or the delay (for a fixed charge) can be scanned at a time. One selects the type of scan with the radio buttons at the last row of the dialog. The value of the fixed variable is set with the first two rows (Delay and Charge), and then the definition of the scan: starting and ending points together with the number of points in the scan.

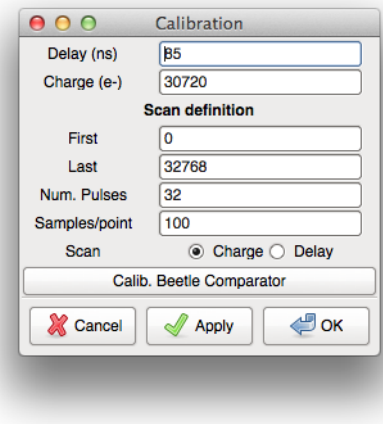

Figure 2. Setting properties of calibration scan

The number of samples per point specifies the number of events that will be acquired for each calibration pulse amplitude.

Note that there is button (quite hidden) with label **Calib. Beetle Comparator**. This is clicked when you want to align the threshold spread of the comparotor of the different channels. See section the Section called *Calibration of the individual channel corrections to the comparator threshold*.

Laser Synchronization

Alibava is able to send a pulse that can be used to trigger a laser system. This run mode scans the delay between the pulse sent by alibava and the acquisition so that the system will sample at the maximum of the signal produced by the laser. The parameters of the delay scan for the laser synchronization are set by clicking on the button on the right of the Laser Sync. radio button. A dialog box like the one in Figure 3 will pop-up.

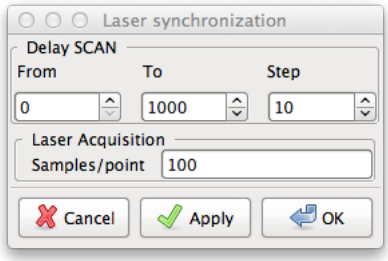

Figure 3. Laser Synchronization scan

The parameters from, to and step define the time interval that will be scanned and the time step with which the laser delay will be increased. The number of samples per point specifies the number of events that will be acquired for each value of the laser delay.

RS, Laser and Pedestal run modes

Laser, source and pedestal runs are very similar modes. However some of the parameters are very specific to the run mode.

For instance, RS mode needs the trigger to be properly configured. This can be done as described in the Section called *External Trigger configuration*.The laser run needs the right delay between the laser strobe and the Beetle trigger, which can be set as described in the Section called *Laser config*.

Configuring alibava

There are a number of ways in which you can configure `alibava-gui`. Once this is done, one can always save that configuration. To save the current configuration click on the **Save** or **Save As** items on the **File** menu of the main window. Saved configurations can be loaded afterwards either by given the configuration file path when starting `alibava-gui` or by choosing a configuration file through the **Open** item in the **File** menu. Most of the configuration parameters can be accessed through the **Settings** menu as shown in Figure 4. Each of the menu items will allow to configure different aspects of the `alibava-gui` behavior.

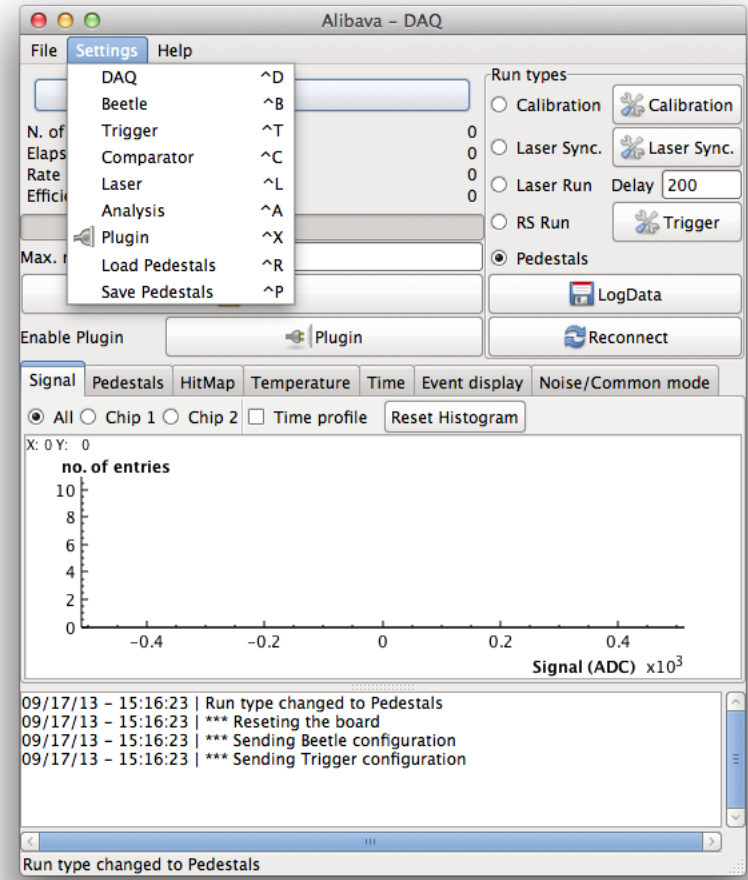

Figure 4. Settings accessible through the Settings item in the main window menu bar

DAQ configuration

The DAQ has a number of parameters:

- sample size: this is the number of events stored in the mother board memory before sending them to the PC. This is only used in Pedestal, Laser and Source modes
- number of events: maximum number of events. When then number of events acquired equals this value the run stops
- Delay: if no data arrives after this delay, alibava-gui will believe there is a communication problem
- Monitor channel: This is the channel whose characteristics curve will be shown in the monitor window

All those parameters can be set by clicking on the DAQ item of the Settings menu as show in Figure 5

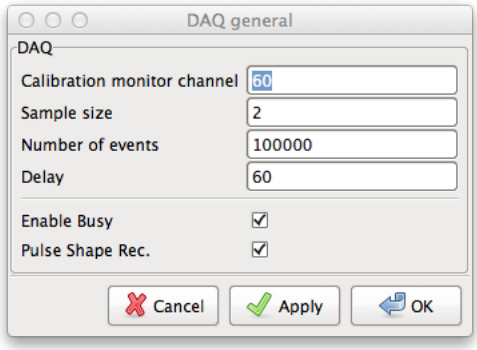

Figure 5. DAQ configuration window

There are two more options that control the behavior of the system.

Enable Busy

Checking this box will make the alibava motherboard send a busy signal through the LEMO connector used for the laser pulse. This will only happen during the Radioactive Source run mode. Remember that the signal output is an LVCMOS 3.3V signal that needs to be terminated to 50Ω. When the signa lis "on" it meands that alibava is BUSY, ie, it is not sensitive to new triggers.

> **Warning**
>
> Be careful with this option since you may send the strobe pulse to a laser system that may be in danger when receiving the strobe while switched off.

Pulse Shape Reconstruction

When running in RS mode, `Alibava` measures the time of the incoming trigger with an internal TDC. The way it works is that the trigger starts the TDC and a system clock raising edge stops it. Since the system clock has a period of 25ns, this will be the maximum value measured by the TDC.checks this box, the system will use a 100ns clock derived from the system clock. This allows to measure longer times and, therefore, to have a measurement of the full pulse shape by plotting the average of the signal as a function of the TDC measured. This is scketched in Figure 6.

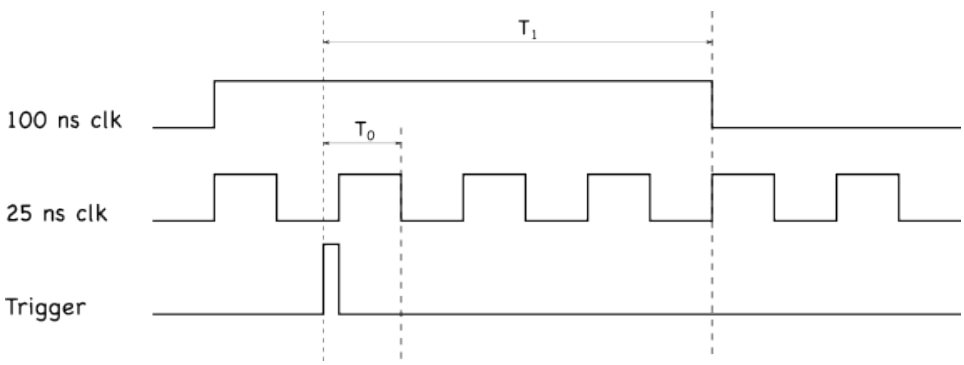

Figure 6. Time scheme of the TDC time

Beetle configuration

The Beetle parameters are set by clicking on the Beetle item of the Settings menu as show in Figure 7. At the top of the dialog you can select which chips will be active during the acquisition.

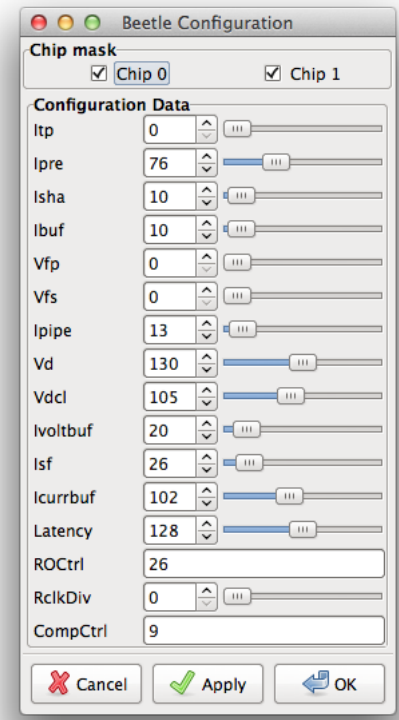

Figure 7. Beetle configuration

The BEETLE chip configuration parameters are described in Table A-1

Beetle Comparator configuration

One can also configure the behaviour of the Beetle comparator. This is only usefull for those daughter boards that process the output of the comparator to produce an autotrigger.

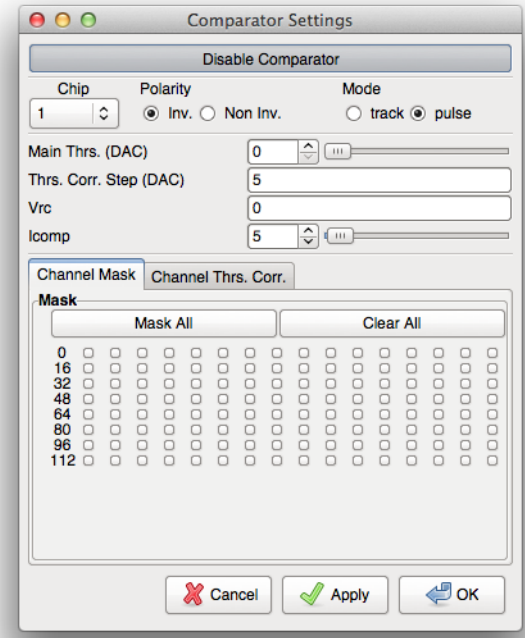

Figure 8. Beetle comparator configuration

The upper button is used to enable and disable this feature. Then one selects the chip and sets the appropriate parameters like the polarity, trigger mode, threshold, some advance parameters and, also, the channel mask and threshold corrections of the individual channels. The beetle chip documentation provides more information about all these parameters.

External Trigger configuration

The trigger for the Source run can be configured by clicking the Trigger item in the Settings menu or the Trigger button by the RS run radio button. The dialog is show in Figure 9. Units are in mV.

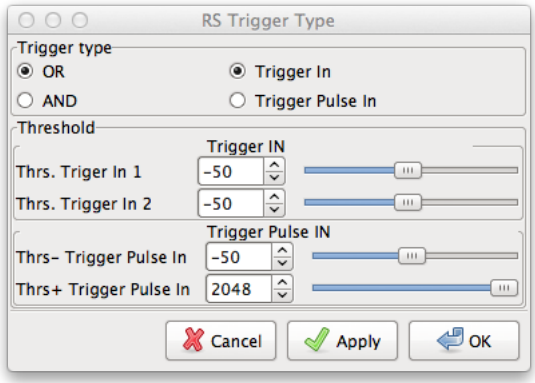

Figure 9. Trigger configuration

There are two types of trigger. One is the Trigger in, which uses two signals to produce the trigger. The input signals are supposed to be negative, as well as the corresponding thresholds. The trigger will be fired it either both (AND) or any of the two (OR) are below the threshold programmed.

The other trigger type is the Pulse in trigger. For this one there are two thresholds, one positive and one negative. The board will produce a trigger for any signal which is either above both values (a positive signal) or below both thresholds (negative). If the input pulse falls between the two values it will not produce a trigger.

Analysis configuration

The data is monitored while acquiring data and this is displayed in a number of histograms. The parameters defining how to find clusters, etc. are displayed in the analysis configuration window shown in Figure 10

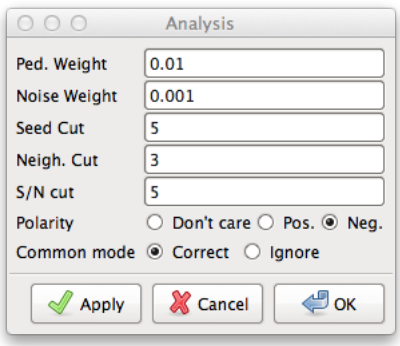

Figure 10. Analysis configuration

Laser config

The only parameter of the Laser run is the delay (in ns) that can be wet in the Laser item of the Settings menu or in the text entry by the Laser radio button.

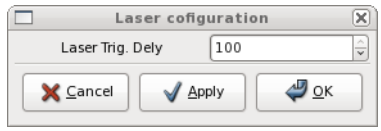

Figure 11. Laser config

Units are in nano seconds

Plugin configuration

The plugin configuration dialog box appears in Figure 12.

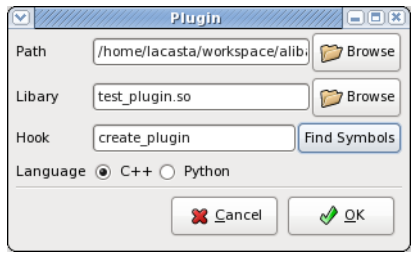

Figure 12. Plugin configuration dialog

There you can specify the plugin language, which can be either C++ or Python, a folder to add to the search path, the name of the library or Python module to load and the function to call. The **Find Symbols** button will open another window with a list of all the callable functions in the plugin. Select one and click OK. Otherwise you will have to type the function (or hook) name. Also note that when clicking on **Browse** for the Library, both the path and the library file name will be filled. `alibava-gui` will also select the language based on very simple assumptions.

Pedestals

`alibava-gui` can compute pedestals *on-line* either by making a pedestal run at the very beginning or estimating the pedestal and noise while taking data. However, for some run types pedestal calculation makes not sense. This is the case of the calibration, laser synchronization and laser since, *a priori*, some channels will always have the same amplitude. Nevertheless, we would like to see the real pedestals subtracted on th emonitoring histograms. To solve this we can either make a pedestal run or load a pedestal file via the **Load Pedestals** item in the **Settings** menu. Likewise, we can always save on a separate file a set of pedestals we are proud of via the **Save Pedestals** item in the **Settings** menu.

The format of that pedestal file is:

- each line corresponds to one channel with line zero for channel 0
- each line contains the pedestal value, followed by a white space-like character (space or tab) and then the noise value

Calibration of the individual channel corrections to the comparator threshold

The comparators on the Beetle chip have a non negligible spread and therefore the chip provides mechanisms to aling the thresholds of all the channels for a given working global threshold. This is a quite involved procedure but alibava-gui provides an authomatic, yet slightly slow, procedure to do it.

> **Warning**
>
> It is worth noting that this will only work for on the alibava daughter boards where the Beetle chip autotrigger is enabled.

The procedure is based on counting the number of triggers for a given configuration of the threshold and the channel corrections. This usually produces the socalled s-curves which, when properly normalized, yield a 1 when all the events produce a trigger, 0.5 when the threshold is equivalent to the mean of the input charge distributioin and 0 when no event produces a trigger. This is done with the Beetle calibration circuit. This system procudes alternate polarities for each channel, always with he same absolute value of the input charge. So when one programs a given number of events, only have of them will have the proper polarity. This is taken into account when normalizing so that when the threshold is so low that we trigger on the noise the value of the s-curve is 2, instead of 1, since we also trigger in the events with the oposite polarity. This defines a characteristic red band in the histograms produced. There is another "effect" that has to do with the case in which we trigger also on the undershoot of the pulse in the events with the oposite polarity. This explains why the s-curves have like to ending points.

The procedure is as follows:

1. Connect the output of the autotrigger in the DB to the TriggerPulse input in the MB.
2. Configure the trigger (see the Section called *External Trigger configuration*) to trigger on TriggerPulse and set the negative and positive values of the threshold to -500 and 50 respectively. Other values may work as well: check with your osciloscope.
3. Click on the Calibration settings button or item in the menu and there click on the button with label **Calib. Beetle Comparator** (see the Section called *Calibration run*). A dialog window lile the one shown in Figure 13 will pop up.
4. Choose the analysis you want to do and click on the Start button.
5. When the analysis are over, click OK if you want to save the settings, or Cancel otherwise.
6. Save the settings in a configuration file by clicking on the Save As.. item in the File menu.

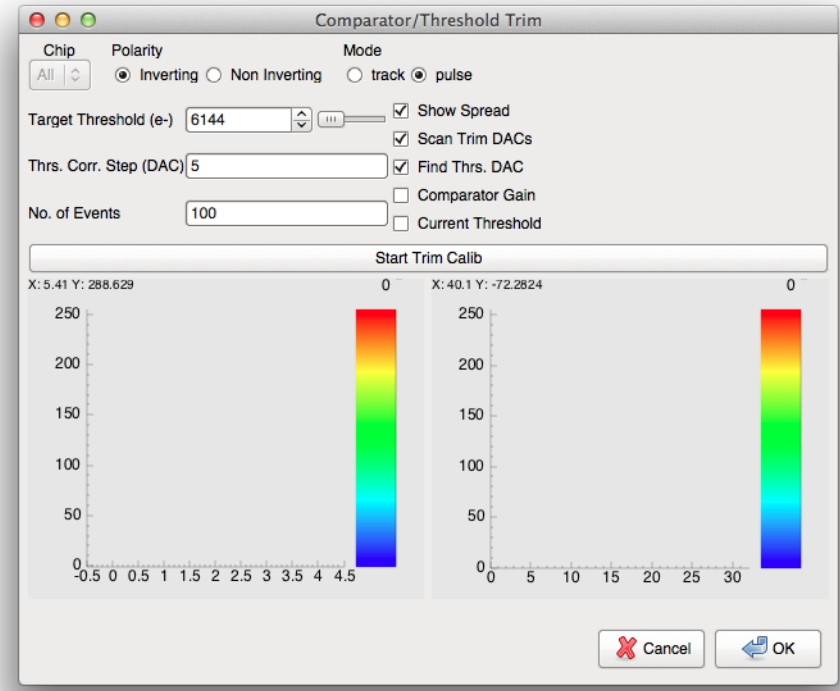

Figure 13. Calibration of the Threshold corrections for the channels

Once the window is there one chooses the target threshold (in units of electron charge: 1 fC or 6250 e- are approx. 22.5 keV. Then we decide if we need inversion at the input of the Beetle comparator or if we use pulse or track mode (see the Beetle documentaion to understand these parameters).

There is, on the left, a series of check buttons that will perform a specific task.

1. Show spread: This will make a scan of the threshold DAC values for an input calibration charge equivalent to the desired thresholds. The plot shows if the thresholds are aligned or not as shown in Figure 14. There one can clearly see the all the channels will have the thresholds alinged that that particular target.
2. Scan Trim DACs: This will make a scan of the threshold correction DAC for each channel and make a linear fit of the main DAC resulting as a function of the correction. This will be used to find the final threshold DAC with the smallest spread among the channels.
3. Find Threshold DACs: THis has to be done together with the previous. It will find the optimad threshold DAC that minimizes the threshold spread.
4. Comparator Gain: This will scan the threshold DAC for different input pulses so that one can see the characteristics curve of each channel's comparator.
5. Current threshold: This can be used to check what is the level of alignment for the current settings. The plot produced is as shown in Figure 15. It makes a charge scan for each channel so that we can see where the threshold is set for each of them. In the figure it is about 6000 e-.

Select the desired analysis and click the start button. You can click again the button to stop at any time. Then click cancel if you do not want to keep the settings or OK if you want to keep them.

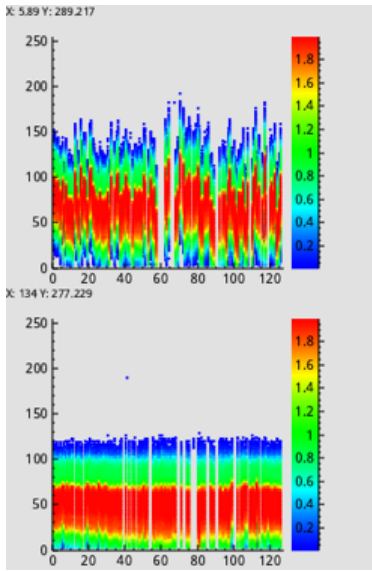

Figure 14. Channel comparator spread before and after alignment

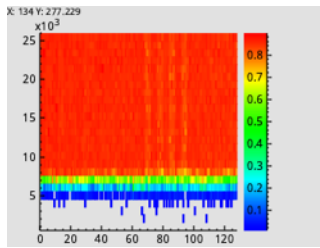

Figure 15. Current settings plot

Monitoring the data

As shown in Figure 1 there is a set of tabs in the main window that show a number of quantities relevant to the acquisition.

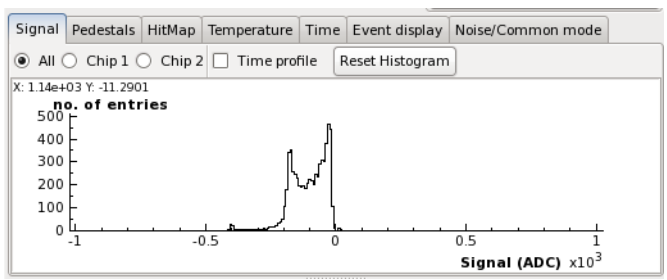

Figure 16. Signal histogram

Figure 16 shows the spectrum. However, do not expect to find here a landau when acquiring in Source mode, since we plot here the signals sampled all along the pulse

shape. You can choose to see all the spectrum from all the chips or from individual chips by clicking in the appropriate radio button on top of the histogram. If you want to see the average value of the signal as a function of the TDC time registered, click on the Time profile button and you will see the average wave form of the signal pulse as shown in Figure 17.

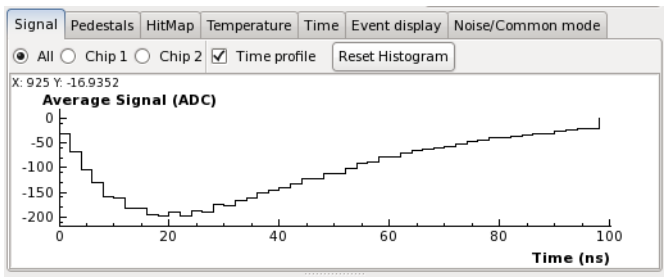

Figure 17. Signal histogram

If you are debugging the system and make changes you can click on Reset histogram and the histogram contents will be cleared so that you can see the effect of your tweaking.

Figure 18. Event display

Figure 18 shows the contents (in ADC units) of each channel for a given event. You can choose to see the raw data, without pedestal and common mode corrections or the *digested* data by clicking on the proper button on top of the histogram.

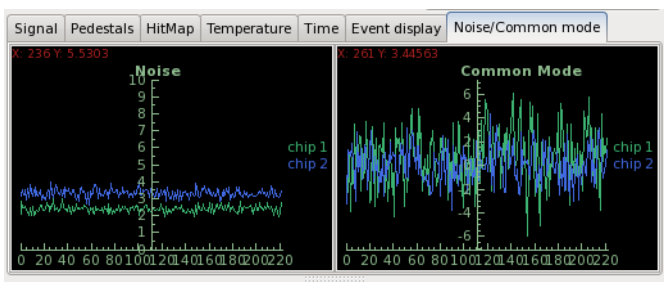

Figure 19. Noise and common mode tracer

Figure 19 shows the average noise and the common mode (in ADC units) of both chips. Note that the noise here and in the pedestals tab are not computed in the same way and the value shown here is slightly higher than there. In the pedestal tab the noise value is show per channel and is the RMS of the pedestal distribution. Here we show the RMS of the ADC values of the channels without signal.

Changing histogram attributes and histogram printing

One can change the histogram attributes by clicking on top of it with the right mouse button. A pop-up menu will appear as shown in Figure 20 .

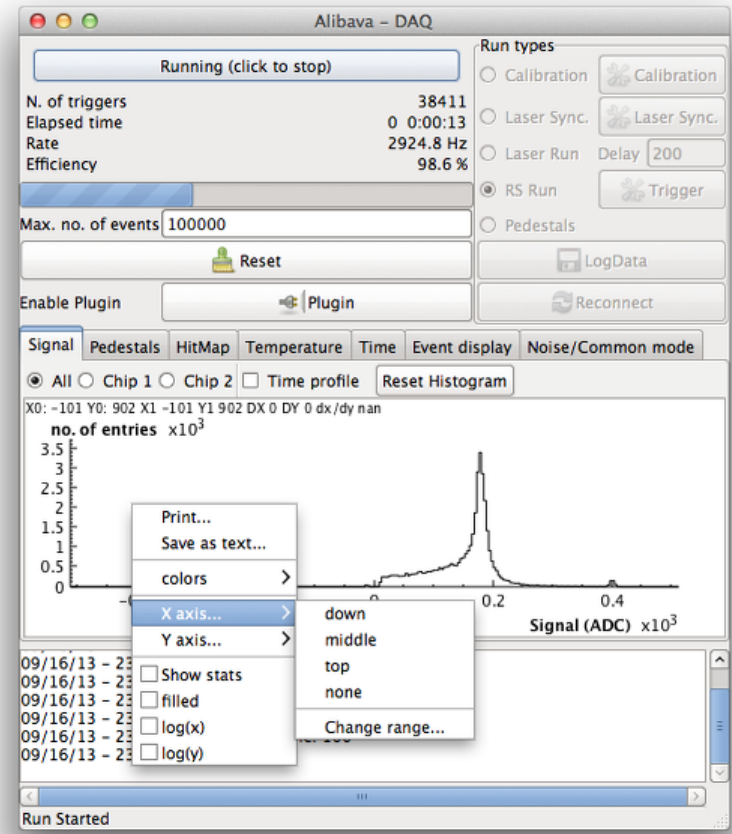

Figure 20. Histogram menu

Clicking on the print item will let you save the histogram as a picture. You can also save it as text so that you can recover the curve with the program of your choice. Clicking on colors will allow you to change the background and foreground colors of the axis, the canvas and the histogram itself. You can also change the position of the axis, its range, change to log scale, draw filled histograms or show the histogram statistics by activating the corresponding radio buttons.

You can also change the range of the histogram axis to zoom a given region. This is done by selecting it with the left button of the mouse while pressing:

- CTRL for the X axis or
- SHIFT for the Y axis

Plugins for alibava-gui

The Plugin object

As already mentioned, alibava-gui allows to load plugins that will enable the end-user to perform non-standard actions at very specific points of the data acquisition process. These stages are:

1. New file: every time we open a new file to store the data
2. Start of run: at the beginning of the run
3. New event: at the beggining of each event.
4. End of event: right before the event is going to be dumped to the output file. This gives the oportunity to filter the events or, even, change the data format (for instance filtering out unwanted channels)
5. End of run

Figure 23 shows the places, during the acquisition loop, in which the plugin methods are called.

The plugins can be written in C++, as shared libraries, or as Python scripts. Examples can be found in the `test` folder of the distribution.

In C++ the plugins are nothing but a class that derives from `Plugin` in Plugin.h, which is shown in Example 1. The `test` folder in the distribution bundle has an example of a C++ plugin together with a make file (`UserMakefile`). The example is described in Example 3. In the make files use the `pkg-config` program to get the compilation flags and the path to to the alibava include files.

Example 1. Plugin C++ class definition

```
class Plugin {
 Plugin();
 virtual ~Plugin();
 enum BlockType = {NewFile=0, StartOfRun, DataBlock, CheckPoint,
          EndOfRun};
 virtual void new_file(std::string &S);
 virtual int start_of_run(int run_type, int nevts, int sample_size);
 virtual bool new_event(int ievt);
 virtual void new_point(std::string &S);
 virtual void end_of_run(std::string &S);
 virtual int filter_event(const EventData & data, std::string &S);
 virtual void get_data_format(std::string &format);
}
```

The main methods in a Plugin class are:

```
void new_file(std::string &S);
```

This method is called at the beginning of a file. This will only happen when data logging is activated. The method returns a string that will be included in the NewFile data block of the data file. See the Section called *The Alibava Data Format* to understand the data blocks.

```
int start_of_run(int run_type, int nevts, int sample_size);
```

This method is called at the beginning of each run. The parameters are:

- run_type: this tells you the current run type (See AlibavaGUI::Runtype in AlibavaGUI.h for the possible values)
- nevts: the number of events for the current run as set by the user on the `alibava-gui` GUI.
- sample_size: this is the number or events that alibava will acquire in each acquisition

`start_of_run` returns an integer value that is the actual number of events that alibava-gui will consider. If the method is not superseeded by your plugin it will just return *`nevts`*. The idea behind this *bizarre* implementation is to allow the user to perform scans on different parameters and redefine this way the total number of events from the number of scan poitns and the number of events per point.

```
void end_of_run(std::string &S);
```

This is called at the end of each run. It can return a buffer with some user data that will be included in the EndOfRun data block of the data file. See the Section called *The Alibava Data Format* to understand the data blocks.

```
bool new_event(int evt);
```

This method is called at the beggining of each event. The argument evt is the current event number.

It returns a boolean which is

- *true*: when we want to right a CheckPoint block in the data before the current event. The actual content of the CheckPoint block will be given by the `new_point` method which will only be called when `new_event` returns true.
- *false*: `new_point` will not be called for this event.

The idea behind this is first, to have a handle right at the beginning of an event and, second, to decide whether we want to add extra information before this event on the data file. This extra information could be the time of the day or, more interesting, the values of the parameters of a scan when a new scan point is going to start.

```
void new_point(std::string &S);
```

This method is called only when `new_event` has returned *true* as explained above. It returns a string that will be included in a CheckPoint data block right before the current event DataBlock (See the Section called *The Alibava Data Format* to understand the data blocks). This is usefull to store in the file the parameters of a user-defined scan or some information that you would like to write periodically, like humidity (if you can measure it), detector current, etc.

Warning

If you use AsciiRoot (see the Section called *The DataFileRoot class*), to access the CheckPoint data you will have to write your own class deriving from AsciiRoot implementing the method `check_point`.

```
int filter_event(const EventData & data, std::string &S);
```

This is called at the end of an event. The idea is that the user can change the information and the format of a normal DataBlock (see the Section called *The Alibava Data Format*). A good example could be a laser scan in which you would only be interested in very few channels. The method returns a string which, if not

empty, will be writen in the DataBlock instead of the normal data. Alternatively, the user may only be interested in monitoring the data and wishes to keep the default format for the data file. In this case, the output string S should be empty, otherwise the program will write the contents on that string.

Warning

If you change the BlockData format then you will have to use a class which derives from AsciiRoot (see the Section called *The DataFileRoot class*) and implements the new_data_block method. Also the pedestal and noise values sotred int he data file will loose their meaning and will be unsusable.

```
void get_data_format(const std::string &data_format);
```

The plugin returns in *`data_format`* a description of the data format in the data chunck returned by `filter_event`. The format is specified as a comma separated list of items of the form

name/fmt

where name is the name to be given to the array and fmt a string describing the type.

It follows the convention of the struct module in Python.

Type description:

- Optional first char

  =: native order, std. size & alignment

  <: little-endian, std. size & alignment

  >: big-endian, std. size & alignment

Followed by a number to specify dimension.

Then comes the type:

c - char

b/B - signed/unsigned byte

h/H - signed/unsigned short

i/I - signed/unsigned int

l/L - signed/unsigned long

f - float

d - double

No other types are allowed

In Python the plugin class is as shown in Example 2.

Example 2. Definition of a Python plugin class

```
class Plugin (extendsobject) :
    def new_file(self) :
    def start_of_run(self, run_type, nevts, sample_size) :
    def new_event(self, ievt) :
    def new_point(self) :
    def end_of_run(self) :
    def filter_event(self, time, temperature, value, data) :
```

The parameters, return values and names of the Python methods are like in C++. The only different method is `filter_event` since it has a different signature.

```
string filter_event(self, time, temperature, value, data);
```

The parameters of this method are:

- time: an integer with the value of the Alibava TDC
- temperature: an integer with the value of the temperature measured by Alibava
- value: the value of the scan variable in the predefined scans (delay in laser synchronization and injected charge in calibration)
- data: an array of 256 integers with the ADC values

Note that the values of *time*, *temperature* and *value* are not decoded and therefore their meaning is as described in Table 4. See the description of the C++ method for more information and warnings.

Plugin Examples

Plugin examples can be found in the folder test on the distribution bundle.

C++ example

In order to make a useful plugin, you have to create your own class implementing some of the methods in Plugin. An example of such a class implementing a user defined scan is shown in Example 3.

Example 3. A C++ plugin to perform a scan

```
/*
 * test_plugin.cc
 *
 * This is an example of a plugin writen in C++.
 * Look at the documentation in PLugin.h
 *
 *  Created on: Jul 24, 2009
 *      Author: lacasta
 */
#include <iostream>
#include <sstream>
#include <Plugin.h>
#include "NewPoint.h"

/**
 * This is an implementation of the Plugin class.
 * It is a simple example that will make a scan.
 * We may use the new_file method to store the parameters
 * of the scan, new_event to determine when a new
 * point is the scan is needed and new_point to store
 * the actual values of
 * the scan variables.
 */
class MyPlugin : public Plugin
{
    private:
        int npoints;          // N. of pts we want for the scan
```

```
        int nevt_per_point;   // N. of pts acquired in each point
        int run_type;         // type of run
        int current_event;    // current event number
        EventCntr handler;    // The object that decides when
                              // to change to the next point

    public:
        // Constructor with default values
        MyPlugin() :
            npoints(50), nevt_per_point(1000), run_type(-1),
            handler(nevt_per_point), current_event(0) {}

        // destructor
        ~MyPlugin() {}

        /**
         * Declaration of Plugin methods to be implemented
         */
        void new_file(std::string &S );
        int start_of_run(int run_type, int nevts, int sample_size);
        void end_of_run(std::string &S);
        bool new_event(int evt);
        void new_point(std::string &S);
};

void MyPlugin::new_file(std::string &rc)
{
    rc = "New file";
    std::cout << "new_file" << std::endl;
}

int MyPlugin::start_of_run(int runtype, int nevts, int sample_size)
{
    run_type = runtype;
    std::cout << "start_of_run " << nevts << " events. "
              << "Runtype " << run_type
              << std::endl;

    if (sample_size > handler.value())
    {
        handler.value(sample_size);
        nevt_per_point = sample_size;
    }
    handler.reset();
    return npoints*nevt_per_point;
}

void MyPlugin::end_of_run(std::string &rc)
{
    rc = "end_of_run";
    std::cout << "end_of_run" << std::endl;
}

bool MyPlugin::new_event(int ievt)
{
    current_event = ievt;
    return handler(ievt);
}

void MyPlugin::new_point(std::string &rc)
```

```
{
    std::ostringstream ostr;
    ostr << "new point: " << current_event << std::endl;
    std::cout << ostr.str();
    rc = ostr.str();
}

/*
 * This is the factory function or "hook" in terms of the
 * Plugin dialog box where the instance of you Plugin
 * implementation is created.
 */
extern "C"
{
    Plugin *create_plugin()
    {
        MyPlugin *plugin = new MyPlugin();
        return plugin;
    }
}
```

In addition to that, we need a factory function that will create the class instance. The name of that function should be specified in the Plugin dialog box when the hook name is required. An example of such a factory function is shown at the very end of Example 3. Note that the function is declared as **extern "C"**. This is important since otherwise alibava-gui will not be able to find it when the shared library is loaded. See the complete example, together with the make file (UserMakefile) in the `test` folder of the distributed software. In your make files use the `pkg-config` program to get the compilation flags and the path to to the alibava include files.

Python example

As already mentioned, plugins can also be written as Python scripts. An example similar to the previous C++ plugin is shown in

Example 4. An example of a Python plugin

```
""" An example of an alibava plugin
    This example implements a user defined scan
"""

import time
import inspect

#
# Define some usefull constants
#
# Block types
NewFile,StartOfRun, DataBlock, CheckPoint, EndOfRun = range(0,5)

# Run types
Unknown,Calibration,LaserSync,Laser,RadSource,Pedestal,LastRType = range(0,7)


class MyPlugin(object):
    """ This is an object that can be loaded by alibava to
        be called at certain stages of the DAQ process.

    """
```

```
def __init__(self):
    """ Initialization
    """
    self.current_point = 0
    self.current_event = 0
    self.npoints = 50
    self.nevt_per_point = 1000
    self.handler = EventCounter(self.nevt_per_point)
    self.run_type = -1

def new_file(self):
    """ This is called at the beginning of each file
        It should return a string with information
        that will be stored in the file header
    """
    print "new_file"
    return "Hola !!!"

def start_of_run(self, run_type, nevts, sample_size):
    """ This is called at the beginning of each run.
        It should return the total number of events
        that we want to acquire. As an extra input we
        have the size of the data chunk that Alibava
        acquires each time we activate the acquisition.

        Run types are predefined in the variables:
        Unknown,Calibration,LaserSync,Laser,RadSource,Pedestal
    """
    info_msg("Starting a new run")
    self.handler.start()
    self.run_type = run_type
    write_msg( "start_of_run %d events. Run type: %d"
               %
               (nevts, run_type ) )
    write_msg( "...sample size %d" % (sample_size) )

    self.handler.reset()
    if sample_size > self.handler.nevts:
        print "Changing handler.nevts"
        self.handler.nevts = sample_size
        print "...new value", self.handler.nevts
        self.nevt_per_point = sample_size

    if run_type!= RadSource and run_type!=Laser:
        return nevts
    else:
        # Here we return the number of events we really
        # want to acquire given that we need to scan npoints
        # with nevt_per_point events per point.
        return self.npoints * self.nevt_per_point

def end_of_run(self):
    """ Called at the end of a run
    """
    write_msg("end_of_run")
    return "end_of_run"

def new_event(self, ievt):
    """ This is called at the beginning of each event.
        Should return True if we want alibava to call
```

```
                the method new_point.

                The input parameter is the current event number
            """
            self.current_event = ievt
            if self.handler.check(ievt):
                return True
            else:
                return False

        def move_axis(self):
            """ A dummy function where we could, for instance,
                move the axis that hole the laser or the source
            """
            pass

        def new_point(self):
            """ Called every time that new_event returns True
            """
            self.current_point += 1
            self.move_axis()
            print "new_point %d event %d" % (self.current_point,
                                             self.current_event)
            return "Current axis position is %d" % self.current_point

    def create_plugin():
        """ This is the 'hook'. This is the method called to
            create an instance of the MyPlugin class
        """

        write_msg("Loading %s" % __name__)
        plugin = MyPlugin()
        return plugin
```

Note that as in the case of C++ we also need here a factory function or hook to create the instance of the Plugin and pass it to Alibava.

Hacking the `alibava-gui` code

`alibava-gui` is written in `C++`. The amount of classes and source files can be a little bit confusing for a beginner that wants to know where and how should a change, improvement or patch be applied. In order to facilitate that, a short description of the code organization will be given here.

There three main groups of objects in `alibava-gui`. In the first group we have the objects in charge of *talking* to the USB port of Alibava, we then have the objects in charge of handling the configuration and, finally the objects in charge of the data acquisition. The main object arbitrating all the interactions with Alibava is `AlibavaGUI`, which also controls the graphical user interface (GUI) of alibava-gui

USB communication objects

The Alibava module can be read and configured via an USB port. `alibava-gui` has decoupled the raw USB communication from the Alibava command generation and readout. This can be seen on Figure 21. The main class, `AlibavaGUI`, has an object, `Alibava`, which is the responsible of generating the commands for the hardware an of reading out the data. It does so with the help of yet another object interface, `USBport`,

which defines the protocol to interact with the USB port. The different implementations of an USBport object have to do with the kernel driver one uses to access the USB data. There are, currently, four of those incarnations of `USBport` which are `USBd2xx`, `USBserial`, `USBFifo` and `USBemulator`.

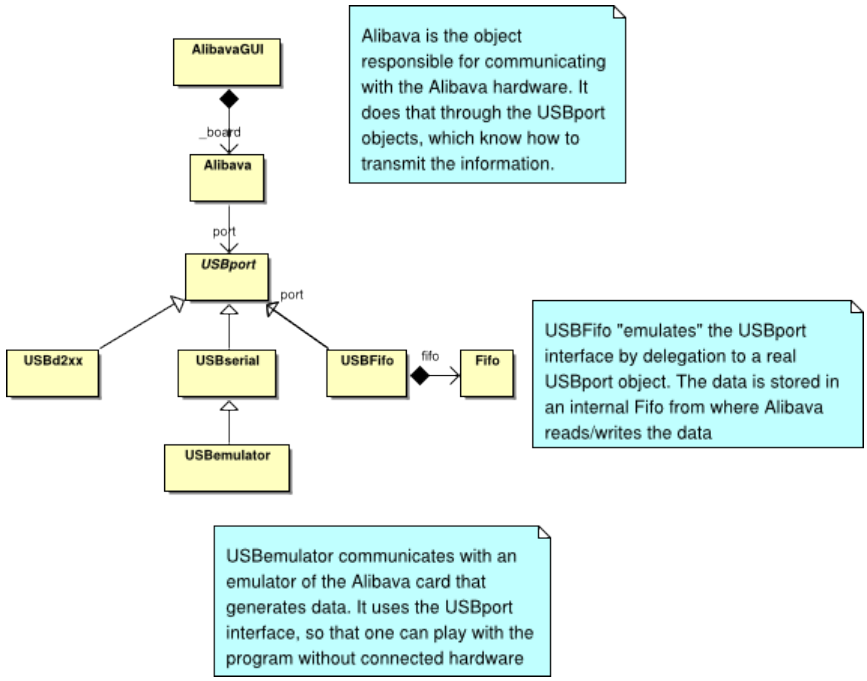

Figure 21. USB communications

`USBserial` makes used of the usbserial driver in Linux. `USBd2xx` uses the FTDI library provided by the USB chip vendor. Finally, `USBFifo` creates a memory Fifo for the USB input from which the user reads. All the raw operations with the USB device are delegated in a `USBport` object given at the instantiation. The default for alibavagui is `USBFifo` using `USBserial`. There is still a fourth one, USBemulator, that contacts a software daemon that emulates the behavior of the Alibava board. With USBemulator one can exercise the program without the need of having any hardware connected to the computer.

The DAQ objects

`AlibavaGUI` handles the acquisition process by communicating with a `RunManager`. Such an object has a number of methods that are called a very precise stages of the acquisition. The role of the RunManager is to impliment the differences of the different among the different run modes and make them *invisible* to the DAQ manager which is `AlibavaGUI`.

Figure Figure 22 shows the different RunManager objects defined in the program. Each of them will handle one of the different run modes.

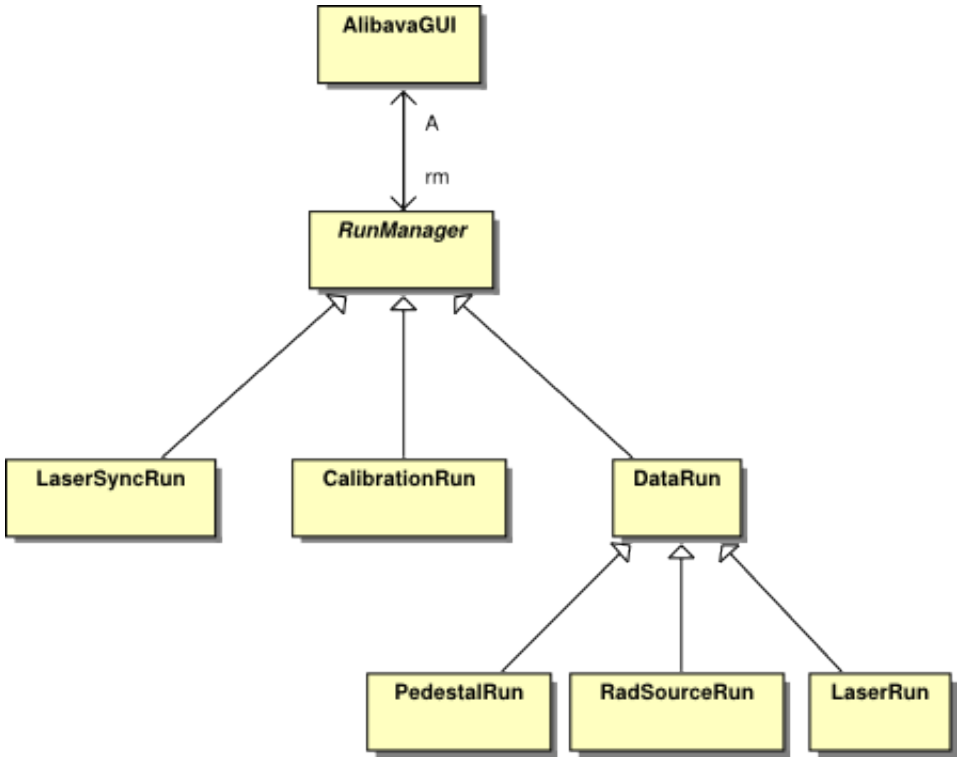

Figure 22. The RunManager

The DAQ loop

The DAQ loop is sketched in Figure 23. There we can see the main players of the acquisition loop. `AlibavaGUI` calls the `open` method which opens the device, sends a reset command and configures the beetle and the trigger. Then, `new_file` and `start_of_run` methods of the `Plugin` are called. At this stage the initialization is over and the program enters the acquisition loop by calling the `acquire` method in `AlibavaGUI`. In Figure 23 we have illustrated a calibration run, but the behaviour is the same for other run modes of `alibava-gui`.

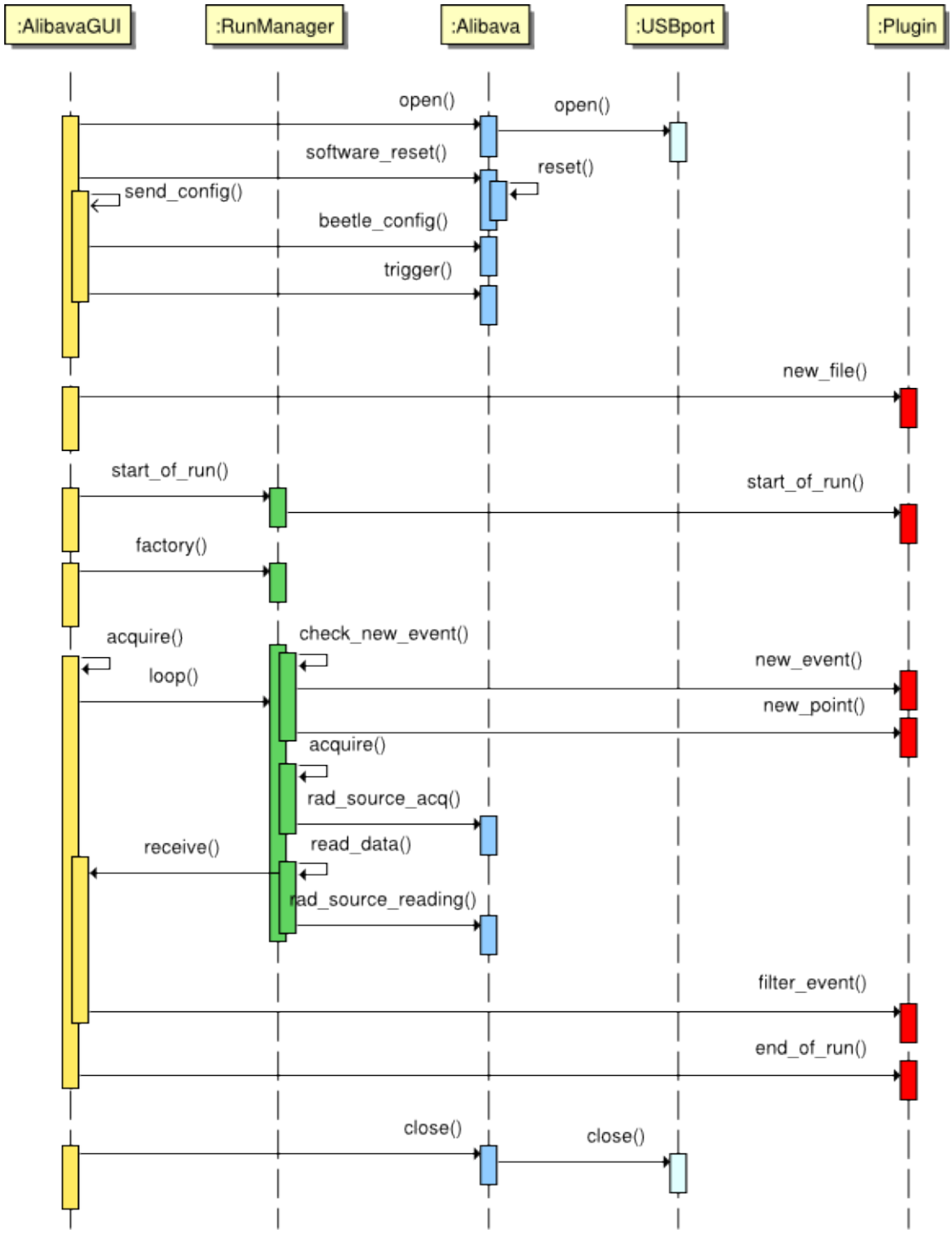

Figure 23. DAQ loop

The configuration objects

Most of the alibava-gui parameters can be configured. The values can be saved to a configuration file or restored from a previously saved configuration file. Each of the parts that can be configured store the data in a class that derives from ConfigFile.

Figure 24 shows all those classes. For each of them, there is a class (same name with a GUI suffix) that allows to see, set and modifiy on dialog windows the current values of the parameters.

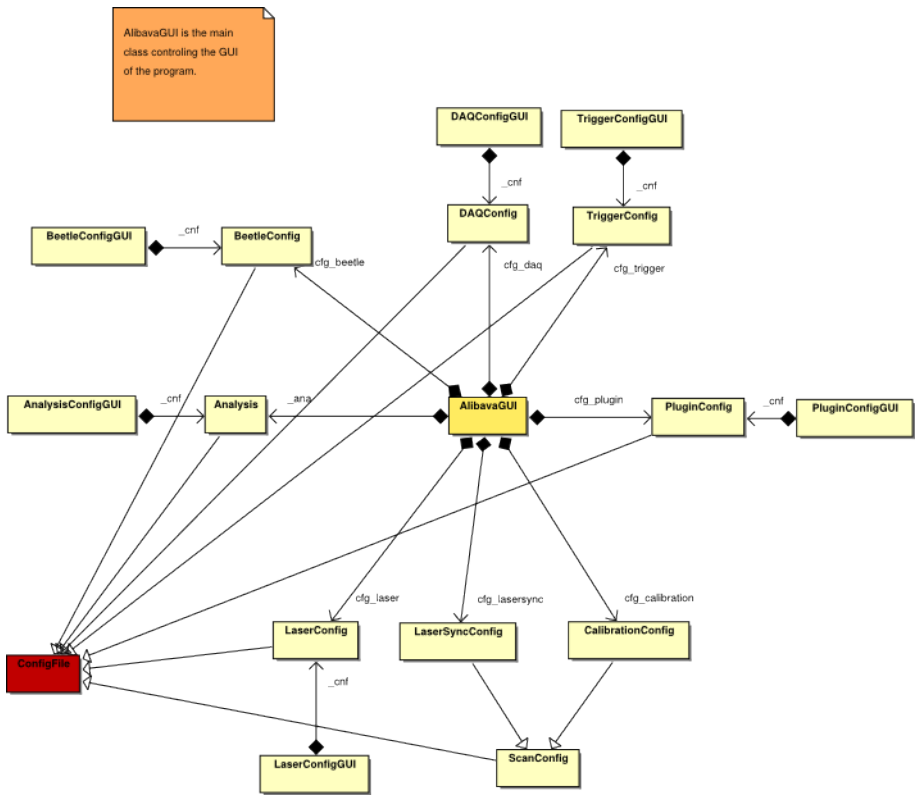

Figure 24. Configuration objects

Data analysis.

`Alibava GUI` can store the data with 2 different formats. The first one is a binary file with a proprietary format which is there for historical reasons. The second data format uses HDF5 which can easily be read from python, Matlab or Octave. The following sections describe the two different formats.

The Alibava Data Format

Binary Data format

The data is stored in binary form. However, the format of the data files is quite simple and it is shown in Table 2. For the sizes used in the tables we follow the convention:

uint32

An unsigned 32 bit integer

uint16

An unsigned 16 bit integer

int16

A signed 16 bit integer

int32

A signed 32 bit

char

An 8bit character (1 byte)

Table 2. Data Format

Data size and type	Meaning
uint32	Time of start of run
int32	Run type. The run type can have various values: 1. Calibration run 2. Laser Sync. 3. Laser 4. Rad. source 5. Pedestal
uint32	Header length (header_length)
header_length * char	Header data. The header data contains some information that is useful when analyzing the data. The header is stored as an ASCII string and the format is:• In the case of calibration of laser sync: • Vn.n | npts;from;to;step • In the case of laser or rad. source: • Vn.n | num_events;sample_size
256 * double (32 bit)	Pedestals (ADC units)
256 * double (32 bit)	Noise (ADC units)

Data size and type	Meaning
Datablock	Following the overall header of the file describing the parameters of the alibava run there are a number of DataBlocks each containing specific information. All the data blocks have the same structure, which is described in Table 4. The possible DataBlocks are:• NewFile • StartOfRun • DataBlock • CheckPoint • EndOfRun

The file data has an overall header, containing the running parameters of Alibava and then a series of data blocks. The data blocks have all the same format, which is described in Table 3. The data itself is one of those data blocks and is the only one which is always written by alibava-gui. The rest are only written when the user activates a plugin and any of the methods returns a data buffer.

Table 3. Format of a data Block

Data size and type	Meaning
uint32: 0xcafennnn	Header of the data block. nnnn is the data block type. The different types can be: 1. NewFile. 2. Start of Run 3. Data 4. Check Point 5. End of Run
uint32	The size in bytes of the block data
size * char	The block data.

Only the Data block has a fixed format, given by Alibava. The format of the other blocks depends on the plugin activated by the user. The format of the Data block in show in Table 4

Table 4. Format of the Data block

Data size and type	Meaning
0xcafe0002	The block data
522	The size of the block data
uint32	Clock counter since the last MB reset. The clock is around 40 MHz but for an accurate value it should be calibrated with a pulse generator used as trigger.

Data size and type	Meaning
uint32	Time as read in the TDC. T = 100.0*(ipart + (fpart/65535.)) whereipart (X & 0xFFFF0000)>>16 fpart sign(ipart)*(X & 0xFFFF)
uint16	Coded Temperature (T = 0.12*X-39.8)
256 * uint16	The ADC values of the 256 channels
double (32 bit)	An extra value that corresponds to the scanned variable in the predefined scans: Calibration (charge) and Laser synchronization (delay)

An example on how to deal with the data can be found in AsciiRoot.cc in the root_macros folder.

The HDF5 data format

In HDF5 the data is structured in groups each having different information. There are 2 main groups. The header group contains general information about the run. It has the setup attribute that specifies the type of run and some other useful information, like the time of the acquisition. It also contains the pedestals and noise of the active channels. The events group has four tables with contain the data collected for each event: the value on each channel in signal, the time given by the TDC (see Figure 6), the temperature measured and a sort of timestamp as a 40MHz clock counter since the last reset of the mother board. See Figure 25.

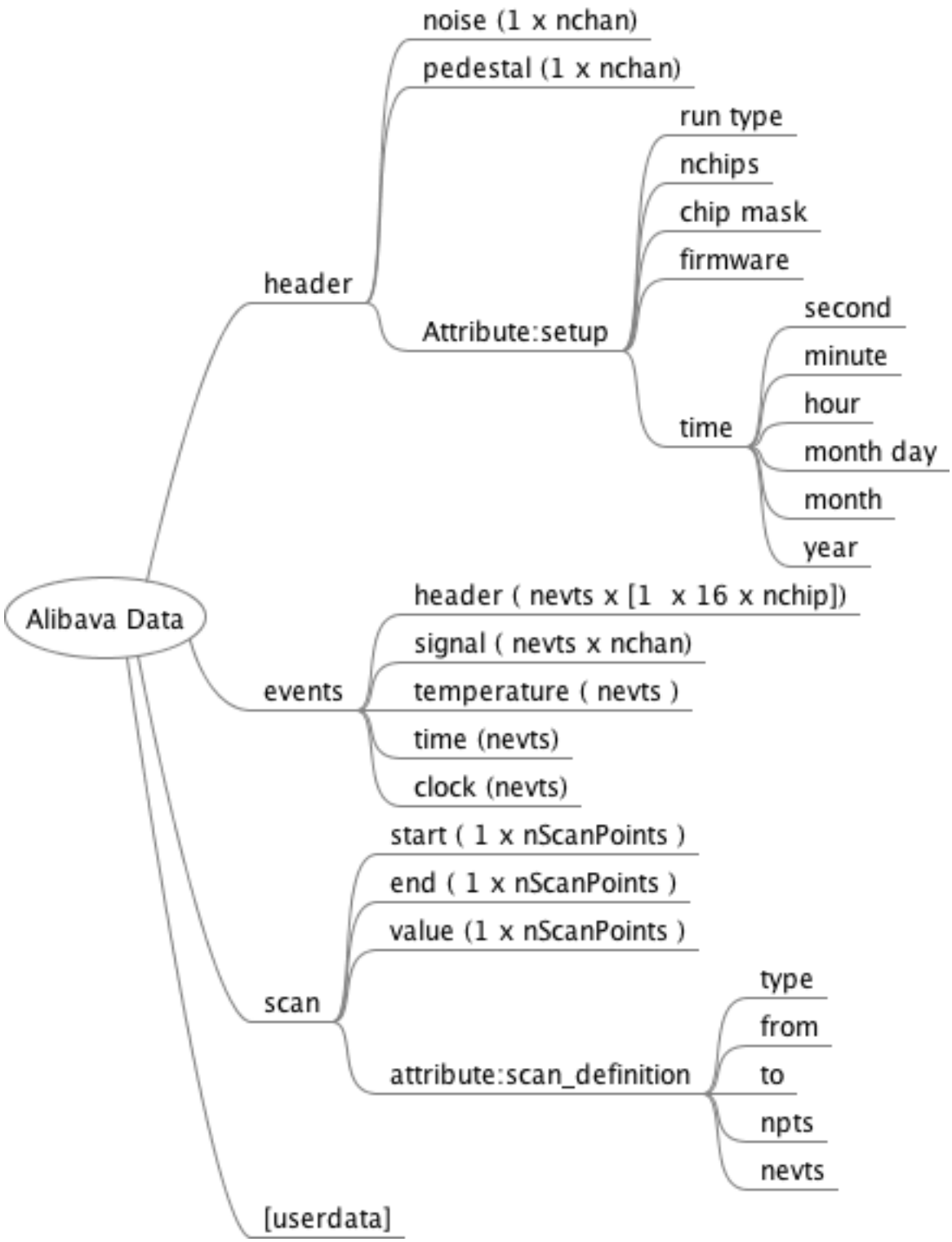

Figure 25. HDF5 file data format

In the Calibration or Laser Scan runs the scan group contains the points at which the scanned values change as well as the description of the scan. Have a look at HDFRoot.cc which provides the data class that handles the hdf5 data.

Analysing the data

Knowing the data format you can write your own program to analyze the data in your preferred language. However, `Alibava` provides a collection of root macros (still evolving) to read the data files and produce histograms. The root macros are in the root_macros folder of the alibava distribution. If you have ROOT already installed during the alibava installation, you will find, at the end of the installation process the ROOT libraries in `INSTALL_DIR/lib/alibava/root`. INSTALL_DIR is usually /usr/local unless you specify it differently as explained in Appendix C.

If you are not planning to modify the source code of the root macros you can use those libraries. To do so, you will need in your working directory a rootlogon.C file that loads them when root is initialized from within that directory. It could look like the one showed in Example 5

Example 5. rootlogon.C for using precompiled ROOT libraries

```
#define DYNPATH "INSTALL_DIR/lib"
#define INCPATH "INSTALL_DIR/include/alibava/root"

void SLload(char *lnam)
{
   if ( gSystem->Load(lnam) )
      cout << ":> " << lnam << " NOT loaded " << endl;
   else
      cout << ":> " << lnam << " loaded " << endl;

}

void rootlogon()
{
    // Add the library folder in the dynamic path so that ROOT finds
    // the library
    TString ss = gSystem->GetDynamicPath();
    gSystem->SetDynamicPath(ss+":"+DYNPATH);

    // Add the Alibava include path in the ROOT include path so that
    // you can include Alibava header files in your own macros
    gInterpreter->AddIncludePath(INCPATH);

    // Load the library
    std::cout << "============================================================
    SLload("libAlibavaRoot.so");
    std::cout << "============================================================

    // This is cosmetics
    gROOT->SetStyle("Plain");
    gStyle->SetPalette(1);
    init_landau();
}
```

If you want to make modifications to the source of the ROOT macros you will need to run make on the `root_macros` folder and, eventually, make install to install the "modified libraries". You can just copy the libAlibavaRoot.so (libAlibavaRoot.dylib in MAC OSX) in a place where ROOT can find it.

In any of the two cases, the best is to start executing a the `sin_preguntas` function that will do almost everything for you.

Example 6. The make-all-for-you function prototype

```
void sin_preguntas(DataFileRoot *A, const char *data_file0, const char*
cal_file0, const char *ped_file0, int polarity0, bool dofittrue, int
tcd05, int tdc115);
```

where the arguments have the following meaning:

A

a pointer to a user supplied DataFileRoot (or descendant) object. Usual implementations are AsciiRoot, to interpret the data with the binary data as described in the Section called *Binary Data format*, or an HDFRoot object to interpret the HDF5 data described in the Section called *The HDF5 data format*. One can also inherit from any of these two to interpret the data produced by a user defined plugin. See the Section called *The DataFileRoot class*. The easiest way to get the pointer is with the static DataFileRoot method OpenFile with is able to determine the file type and creates the proper class pointer.

```
DataFile *DataFileRoot::OpenFile(const char *file_path, const char
    *pedfile0, const char *gainfile0);
```

data_file

The path of the data file to be analyzed. If NULL, the current file in A will be used

cal_file

The path of a calibration file. It can be an Alibava data file produced during a calibration run or an ASCII text file with as many lines as channels with gain and offset in each line. If you do not have this file, set 0 here. The only difference is that if the calibration file the histogram units will be in electrons. Otherwise they will be in ADC units.

ped_file

compute pedestals or an ascii text file with as many lines as channels and pedestal and noise for each channel. If no file is given, sin_preguntas will use the data file to compute pedestals.

polarity

this is the expected polarity of the signal (or the bias voltage): -1 for negative signals and +1 for positive signals.

dofit

this is a boolean that specifies whether the program should try to fit a landau to the signal histogram. If true is given it will do the fit.

tdc0, tdc1

Define a time window around the peak of the pulse shape to produce the signal histogram

In any case you can have a look there to see how the data is handled in the *usual* cases. Have a look at `analysis.cc` to see how the DataFileRoot class is used and how data is analyzed in the examples provided.

The DataFileRoot class

In the root_macros folder you will find a number of example files to analyze the data. They do not intend to be a standard but just examples. At least this is how they were born, though they have been evolving and, as of today, they are too complicated an example. However the DaraFileRoot class can still serve as a good tool to read the files and to access the current data to make your own analysis.

Most of the methods in `DataFileRoot` are applied indistinctible to all the channels in a chip of the DB. However, some of them can be applied just to a set of channels. These sets or regions are defined with the `ChanList` class. This class is described in

Example 7. The ChanList class definition

```
 class ChanList {
  public ChanList(const char * list_def = 0);
  public void Set(const char * list_def);
}
```

The DataFileRoot class definition is shown in Example 8. Only a few methods are show here. For the complete definition of the class, please look in DataFileRoot.h.

Example 8. The AsciiRoot class definition

```
 class DataFileRoot {
  public AsciiRoot(const char * data_file);
  public ~AsciiRoot();
  public enum BlockType = {NewFile=0, StartOfRun, DataBlock, CheckPoint,
            EndOfRun};
  public bool valid();
  public void open(const char * data_file);
  public void close();
  public void rewind();
  public int read_event();
// Plugin extra data Blocks
  public virtual void new_file(int size, const char * data);
  public virtual void start_of_run(int size, const char * data);
  public virtual void check_point(int size, const char * data);
  public virtual void new_data_block(int size, const char * data);
  public virtual void end_of_run(int size, const char * data);
  public void set_data(int size, const unsigned short * data);
// Analysis methods
  public TH2 * compute_pedestals(int mxevts = -1, bool do_cmmd = true);
  public void compute_pedestals_fast(int mxevts = -1, double ped_weight = 0.01, double noise_wei
  public void load_pedestals(const char * file_name);
  public void save_pedestals(const char * file_name);
  public void load_gain(const char * file_name);
  public void load_masking(const char * file_name);
// Anaylsis in strip regions
  public int n_channel_list();
  public void add_channel_list(const ChanList & C);
  public void clear_channel_lists();
  public ChanList get_channel_list(int i);
  public void find_clusters(const ChanList & C);
  public void common_mode(const ChanList & C, bool correct = false);
// Debugging methods
  public void spy_data(bool with_signal = false, int nevt = 1);
  public TH1 * show_pedestals();
```

```
  public TH1 * show_noise();
}
```

By default, DataFileRoot only reads the DataBlock which is the only that has a more or less defined format. If the user has created other data blocks with a user-defined plugin, then he/she will have to define a class which derives from AsciiRoot and implements the methods that receive the data from those extra blocks. Those methods are explained below

`AsciiRoot`constchar *`data_file`

The constructor. data_file is the path of the data file.

```
public void new_file(int size, const char * data);
```

This method is called whenever a NewFile block is found on the file. The arguments are the size of the block data and the data itself (see Table 3).

```
public void start_of_run(int size, const char * data);
```

This method is called when a StartOfRun block is found on the data file. The arguments are the size of the block data and the data itself (see).

```
public void check_point(int size, const char * data);
```

This method is called when a CheckPoint block is found in the data file. The arguments are the size of the block data and the data itself (see Table 3).

```
void new_data_block(int size, const char * data);
```

This method is called when a DataBlock is found in the data file. The main use of this method is to decode the event data when a Plugin::filter_event method (see Example 1) has modified the default data format during the acquisition. The arguments are the size of the block data and the data itself (see Table 4). This method should call set_data in order to set the active channels and their ADC values.

Warning

Note that when you change the default format in the DataBlock, the pedestal and noise values stored in the file loose their meaning and you will have to recompute them with `compute_pedestals` or `compute_pedestals_fast`

```
void end_of_run(int size, const char * data);
```

This method is called when an EndOfRun block is found in the data file. The arguments are the size of the block data and the data itself (see Table 3).

```
void set_data(int size, const unsigned short * data);
```

This method should be used when the user has modified the DataBlock format. You should provide the number of channels (*`size`*) and an array with the ADC values (*`data`*)

```
void load_pedestals(const char * file_name);
void save_pedestals(const char * file_name);
```

load/save pedestals from/to a file. The file is a simple ASCII file, each line containing the pedestal and noise values of a channel. Line *i* corresponds to channel *i*.

```
void load_gain(const char * file_name);
```

Load the gain factors (ADC counts to electrons) of the channels. The input file is an ASCII file, each line containing the channel number followed by the gain value.

```
TH2 * compute_pedestals(int mxevts = -1, bool do_cmmd = true);
```

This method computes the pedestals in the usual way. What it does is to produce, for each channel, a histogram with all the ADC values and fit a gaussian to the peak with the lowest mean. The pedestal and noise of that channel will be the mean and the sigma of the gaussian fit. It returns a 2D histogram showing the distribution of all the channels. The method parameters are:

- mxevts: number of events to use in the pedestal calculation. If negative, then all the events in the file will be used.
- do_cmmd: if set to true, the algorithm will make common mode subtraction on an event by event basis.

```
void compute_pedestals_fast(int mxevts = -1, double ped_weight = 0.01, double noise_w
```

This method computes the pedestals with a somewhat different algorithm than `compute_pedestals`. It tries to follow any change of the pedestal and the noise of the channels and updates their values. It is the method that `alibava-gui` uses to monitor the data during the acquisition. For analysis one should use `compute_pedestals`.

```
void spy_data(bool with_signal = false, int nevt = -1);
```

This method is very useful to debug the data. It shows a pannel of histograms for a single event, like the raw data, processed data, common mode noise, found clusters, etc. If the first argument is true it will only show events with signal, skipping the events where no clusters have been found. The second argument is the number of events you want to see. The default is to show only one event at a time, but you could see as many as the number indicated.

For more information take a look at DataFileRoot.h and the source code in DataFileRoot.cc. In the test folder of the distribution bundle you will also find some examples.

SOAP: Communicating with alibava-gui

`alibava-gui` can also work as a SOAP server. This means that we can send a number (very limited) of commands to the `alibava-gui` process to start or stop a run and, also, to retrieve some information about the status of the acquisition. The SOAP server and client libraries are implemented using the gSOAP toolkit for Web Services[11]

The SOAP commands are summarized below:

```
void getStatus(in int request, out Status status);
```

Returns the status of the acquisition in a structure of type Status, described in Example 9. The `request` parameter can be ignored and one can send any value, usually 0.

```
void Reset(in value);
```

Resets the board

`void startRun(in DAQParam daqParam, out base64binary data);`

Starts a run with the parameters specified in a structure of type DAQParam described in Example 11. It returns a data block which contains various histograms. This is a synchronous methods and will not return until the run is over.

`void startRunAsync(in daqParam);`

Starts a run with the parameters specified in a structure of type DAQParam described in Example 11. It returns immediately. You should use the `getStatus` method to check the status of the run.

`void Reset(in value);`

Resets the board

`void stopRun(in int value, out int response);`

Stops the current run. The `value` parameter has no meaning and any value is accepted. The output parameter, `response`, with return 0 if the operation was succesful and an error code otherwise.

`void setDataFile(in string fileName, out int response);`

Sets the name of the data file and forces `alibava-gui` to log data into that file. The output parameter, `response`, with return 0 if the operation was succesful and an error code otherwise.

`void setParameter(in ParValue value, out int response);`

Set the values of some running parameters like some registers of the Beetle chips.

ParValue, as shown in Example 10 is a pair of a name and a value. The name identifies the parameter. Parameter names are shown in Table 5.

`void getHistogram(in string hstName, in string hstType, out int response);`

Gets the picture of the historam hstName with type hstType, which can be any of: png, jpg, svg, eps and pdf. The names of the available histograms are listed in Table 6. Note, however, that the histograms are produced only when the server is run in GUI mode.

Example 9. The Status structure

```
class Status {
 public string status;
 public time time;
 public int nexpected;
 public int ntrigger;
 public double rate;
 public string run_type;
 public double value;
}
```

Example 10. The ParValue struc

```
class ParValue {
 public string name;
 public string value;
}
```

Example 11. The DAQParam structure

```
 class DAQParam {
  public int runType;
  public int numEvents;
  public int sampleSize;
  public int nbin;
  public double xmin;
  public double xmax;
}
```

Table 5. Names of the parameters that can be changed with setParameter. The table does not show the parameters tht refer to a beetle register. Those names are the same as in Table A-1

Parameter Names	Description
trgIn	Switches ON/OFF Trigger In
thrsIn1	Threshold for the first TrigIn comparator
thrsIn2	Threshodl for the second TrigIn comparator
trgAND	Set AND as the operation between In1 and In2
trgOR	Set OR as the operation between In1 and In2
trgPulse	Switches ON/OFF Pulse
thrsPulse-	Nevative threshold for TrigPulse
thrsPulse+	Positive threshold for TrigPuls
enableComp	Switches ON/OFF the Beetle comparator
trackMode	Set the comparator working mode to track
pulseMode	Set the comparator working mode to pulse
compPolarity	Sets the polarity for the comparator
mainTh	It sets the main comparator settings. The parameter name can be followed by a number specifying a particular chip. If no number is given, the valu will be applied to to all chips.
deltaTh	The parameter name can be followed by a number specifying a particular chip. If no number is given, the valu will be applied to to all chips.
trimCh<n>	Sets the vector of corrections for trimming the threshold. <n> is the chip number and should be specified.

Parameter Names	Description
maskCh<n>	Sets the vector of corrections for trimming the threshold. <n> is the chip number and should be specified.
fileFormat	Set the data format in the data file. Valid values are *hdf* and *alibava*

Table 6. Names of the histograms that can be retrieved as pictures.

Histogram Name	Description
hstSignal	The spectrum
hstPedestal	The pedestals of all the channels
hstNoise	The noise of all the channels
hstHitmap	The hitmap
hstTemp	The tracer of the measured temperature
hstTime	The tracer of the TDC time
hstEvent	The event viewer. It show, for a given event, the content of each channel.
hstCmmdNoise	The noise from the common mode
hstCmmd	The common mode

The data returned by the startRun consists of 3 histograms in most cases. The first contains the spectrum, the second the mean value of the signal seen by each channel and the third the rms of the signal seen by each channel. In a Pedestal run the last two histograms would correspond to the pedestals and noise respectively. The format of the data is described in Table 7.

Table 7. Format of the data from startRun

Size and type	Description
int32	number of histograms
For each histogram	
int32	number of bins
double	xmin
double	xmax
nbin * double	data chunk with nbin doubles

There are available a Python and a C++ wrap classes to hide the SOAP complexity to the user. These are described in the Section called *SOAP examples*.

SOAP examples

The test folder in the distribution contains some examples that may help understanding the procedure to communicate with `alibava-gui` via SOAP. The examples are written in two languages, python and C++. There are Python and C++ classes that hide the complexity of the SOAP implementation and privide a much easier interface. The interface is very similar in both programming languages.

Example 12. SOAP interface

```
 class Alibava {
  Alibava(const char * uri = 0);
  void connect(in const char * uri);
  getStatus(out Status &status);
  Reset(in int val);
  setDataFile(in string &file_name);
  public int setParameter(in string &name, in string &value);
  int getMask();
  void stopRun();
  startPedestalRun(int nevt, bool async, int nbin = 512, int xmin = -512, int xmax = 512, int nsa
  startSourceRun(int nevt, bool async, int nbin = 512, int xmin = -512, int xmax = 512, int nsamp
  startLaserRun(int nevt, bool async, int nbin = 512, int xmin = -512, int xmax = 512, int nsampl
  startCalibrationRun(bool async, int nevt_per_point = 50, int npts = 10, double vfrom = 0.0, dou
  startLaserSync(bool async, int nevt_per_point = 50, int npts = 10, double vfrom = 0.0, double v
  startChargeScan(bool async, int nevt_per_point = 50, int npts = 10, double vfrom = 0.0, double
}
```

The histograms returned by the methods that start a synchronous run are retrieved differently in python and C++. The histograms are returned directly by the method in python, while in C++ one need to retrieve them by calling the get_histogram method. See the examples below.

Python example

In the case of python, we recomend to have `SOAPpy` installed in the system. It can be downloaded from http://sourceforge.net/projects/pywebsvcs/files/SOAP.py. It is also in most of the Linux distributions so the best is to use the one provided by your particular distribution if you are using Linux. The example below uses this python package. There are two parts. One that hides all the complexity of the SOAP data types intrinsic to Albava and the other the one that contains your actual commands.

The first one is shown in Example 13 and the second in Example 14.

Example 13. alibavaSOAP: internals to alibava SOAP structures in Python

```
#!/usr/bin/env python
"""
Example of a soap client for alibava
"""
import sys
from alibavaSOAP import Alibava, Status
import SOAPpy


def main(host="localhost", port=10000):
    server = Alibava(host, port)

    S = Status( server.getStatus() )
    print S
    R = server.stopRun()

    R = server.startPedestalRun(1000)
    S = Status( server.getStatus() )
    print S
```

```
        R = server.startLaserRun(1000, nbin=32, xmin=-512, xmax=512)
        S = Status( server.getStatus() )
        print S
        for hst in R:
            print hst


        server.setDataFile( "alibava_data.dat" )
        R= server.startSourceRun(1000, nbin=32)
        for hst in R:
            print hst


        R = server.startCalibrationRun(100, 20, 0, 30000)
        for hst in R:
            print hst


        #SOAPpy.Config.debug=1
        server.setParameter("Isha", 32)
        server.setParameter("Vfs", 19)
        server.setParameter("trgPulse",1)

        mask=”
        for i in range(0,128):
            if i%4 : mask+=’1’
            else: mask+=’0’
        server.setParameter("maskCh1", mask)

        trim=”
        for i in range(0,128):
            if i%4 : trim+=’0,’
            else: trim+=’1,’
        server.setParameter("trimCh1", trim)


    if __name__ == "__main__":
        try:
            host = sys.argv[1]
        except IndexError:
            host = "localhost"

        try:
            port = int(sys.argv[2])
        except IndexError:
            port = 10000

        main(host, port)
```

Example 14. testSoap.py: an example of use

```
#!/usr/bin/env python
"""
Example of a soap client for alibava
"""
import sys
from alibavaSOAP import soapClient, Status


def main(host="localhost", port=10000):
    # connect to the server
    server = soapClient(host, port)

    # Get the current status and print it
    S = server.getStatus()
    print S

    # Stop the run
    R = server.stopRun()

    # Start a pedestal run with 1000 events
    R = server.startPedestalRun(1000)

    # Start a Laser Run. Set the parameters of the
    # histogram axis
    R = server.startLaserRun(1000, nbin=32, xmin=-512, xmax=512)
    for hst in R:
        print hst

    # Set the name of the data file and start a Source Run
    server.setDataFile( "alibava_data.dat" )
    R= server.startSourceRun(1000, nbin=32)
    for hst in R:
        print hst


if __name__ == "__main__":
    try:
        host = sys.argv[1]
    except IndexError:
        host = "localhost"

    try:
        port = int(sys.argv[2])
    except IndexError:
        port = 10000

    main(host, port)
```

C++ example

The alibava package provides libraries for communicating with `alibava-gui`. The test folder of the distribution contains an example, which also shown in Example 15. Look at the given Makefile to see which are the requered includes and libraries. The example shown here also uses the Histogram class whose implementation can also be found in the test folder of the distribution.

Example 15. C++ soap client

```
#include <iostream>
#include <alibavaClient.h>


int main(int argc, char **argv)
{
    const char *uri = "http://localhost:10000";
    Alibava client;

    if (argv[1])
        uri = argv[1];

    client.connect(uri);


    ns1__Status status;
    client.getStatus(status);
    std::cout << "Status:" << std::endl;
    std::cout << status << std::endl;

    client.startPedestalRun(5000, false);
    client.getStatus(status);
    std::cout << "Status:" << std::endl;
    std::cout << status << std::endl;

    client.startLaserRun(5000, false);
    client.getStatus(status);
    std::cout << "Status:" << std::endl;
    std::cout << status << std::endl;

    client.setParameter("fileFormat", "hdf5");
    client.setDataFile("/tmp/data_file.dat");
    client.startSourceRun(10000, false);
    client.getStatus(status);
    std::cout << "Status:" << std::endl;
    std::cout << status << std::endl;
    client.get_histogram(0)->Print(std::cout);

    return 0;
}
```

Monitoring Alibava runs from *abroad*

The SOAP server also provides a web service where you can monitor the current status of your run. The interface is really simple. The addres you have to give to the browser is

```
http://address_of_your_computer:nnnn
```

where nnn is the port you have given to the SOAP server (10000 by default). You may need to enable that port in order to do that from another computer. Figure 26 shows an exmple of what the server will provide. It will, essentially, be the histograms that are shown at the main window of the alibava-gui application.

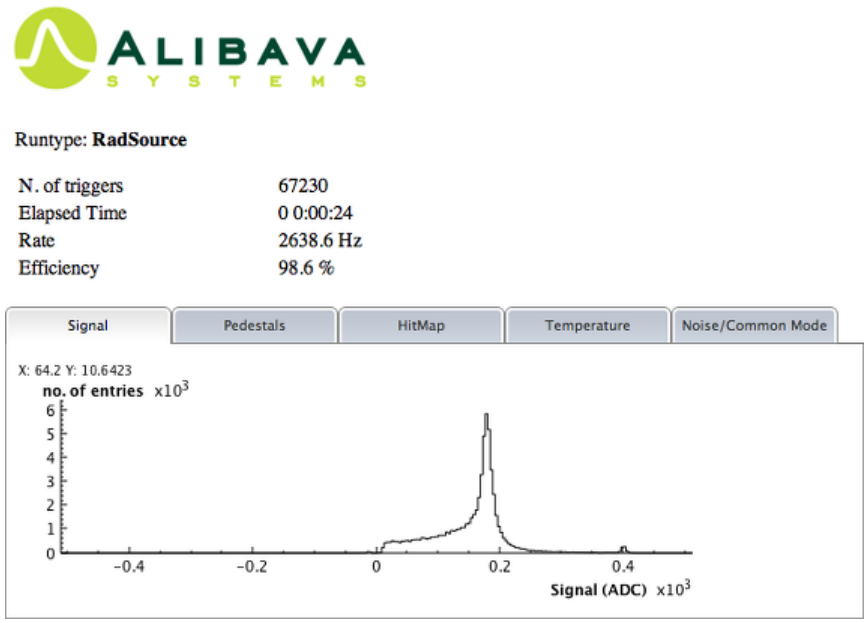

Figure 26. Web server

A. Parameters of the Beetle chip

This appendix lists the parameters of the Beetle chip as described in the original work by S. Löchner and M. Schmelling (The Beetle Reference Manual, LHCb-2005-105).

Table A-1. Beetle parameters and default values

Name	Range	Step	Nominal	Reg. content	Description
Itp	0-2 mA	7.8 μA	0 μA	0x00	test pulse bias current
Ipre	0-2 mA	7.8 μA	600 μA	0x4C	preamplifier bias current

Name	Range	Step	Nominal	Reg. content	Description
Isha	0-2 mA	7.8 μA	80 μA	0x0A	shaper bias current
Ibuf	0-2 mA	7.8 μA	80 μA	0x0A	front-end buffer bias current
Vfp	0-2.5 V	9.8 mV	0 mV	0x00	preamplifier feedback voltage
Vfs	0-2.5 V	9.8 mV	0 mV	0x00	shaper feedback voltage
Icomp	0-2 mA	7.8 μA	40 μA	0x05	comparator bias current
Ithdelta	0-2 mA	7.8 μA	--	--	current defining incremental comparator threshold
Ithmain	0-2 mA	7.8 μA	--	--	current defining common comparator threshold
Vrc	0-1.25 V	4.9 mV	0 mV	0x00	comparator RC time constant
Ipipe	0-2 mA	7.8 μA	100 μA	0x0D	pipeamp bias current
Vd	0-2.5 V	9.8 mV	1275 mV	0x82	pipeamp reset potential
Vdcl	0-2.5 V	9.8 mV	1030 mV	0x69	pipeamp reference voltage
Ivoltbuf	0-2 mA	7.8 μA	160 μA	0x14	pipeamp buffer bias current
Isf	0-2 mA	7.8 μA	200 μA	0x1A	multiplexer buffer bias current
Icurrbuf	0-2 mA	7.8 μA	800 μA	0x66	output buffer bias current
Latency	10-160	--	160	0xA0	trigger latency
ROCtrl	--	--	cf. table C.11		readout control

Name	Range	Step	Nominal	Reg. content	Description
RclkDiv	0-255	--	0	0x00	ratio between Rclk and Sclk
CompCtrl	--	--	See Table A-2		comparator control
CompChTh	0-31	--	--	--	comparator channel threshold shift register implementation and Beetle revision Id.
CompMask	--	--	0	0x00	comparator mask shift register implementation
TpSelect	--	--	0	0x00	test pulse selection shift register implementation
SEUcounts	0-255	--	--	--	sum of Single Event Upsets

The following table describes the bits in the CompCtrl reagister

Table A-2. Bits in the CompCtrl register

Bit	Function	Description
0	DisableCompLVDS	0: enable comparator LVDS output ports 1: disable comparator LVDS output ports
1	CompPolarity	0: inverting 1: non-inverting
2	PipelineMode	0: analogue readout 1: binary readout
3	CompDisable	0: enable comparator 1: disable comparator
4	CompMode	0: track mode 1: pulse mode
5-7	not used	--

B. Start-up guide: getting the motherboard out of the box...

Basic connections and initialization of the system

There are a number of very easy steps required to get the system ready. Please read the `Alibava GUI` manual first to have the data acquisition software properly installed and ready. Then, the daughter board should be connected to the motherboard, provide power to the later and connect it to the USB port of the computer.

Figure B-1 shows the main ALIBAVA system connections.

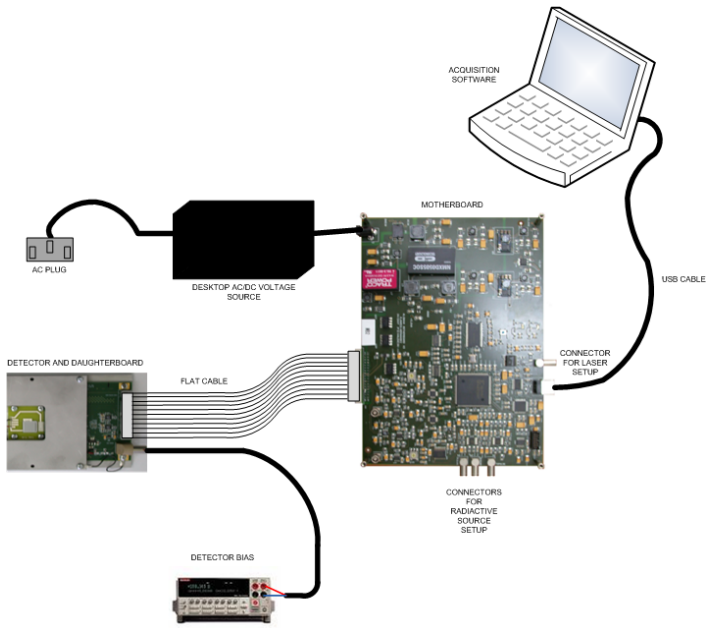

Figure B-1. Alibava system sketch with all the needed connections.

These are the steps that must be followed in order to initialize the system:

1. Connect the daughter board to the mother board (item 2 of Figure B-2) by means of the flat cable (the IDC connectors have a defined position both in the daughter board and the mother board).
2. Power on the system by means of connecting the AC/DC adapter to the motherboard power connector (item 1 of Figure B-2). The red and the green LEDs of the motherboard (item 8 of Figure B-2) are switched on.
3. Connect the USB cable to a USB port on the computer where the software is installed and to the USB connector of the motherboard (item 6 of Figure B-2).
4. Push the reset button (item 9 of Figure B-2) of the motherboard to initialize the system.
5. Launch the software following the software documentation. Now the hardware has been synchronized with the software and the red LED of the motherboard will be switched off (item 8 of Figure B-2).

At this point the system has been initialized correctly and it is ready for preparing the required connections for the laser or the radioactive source runs.

The detector(s) of the daughter board must be powered on by means of an independent power supply. The daughter board has a Lemo power connector (with a defined position for connecting the power plug of the cable) dedicated for this supply.

Calibrations and pedestals acquisitions can be carried out at this point once the detector(s) have been powered on.

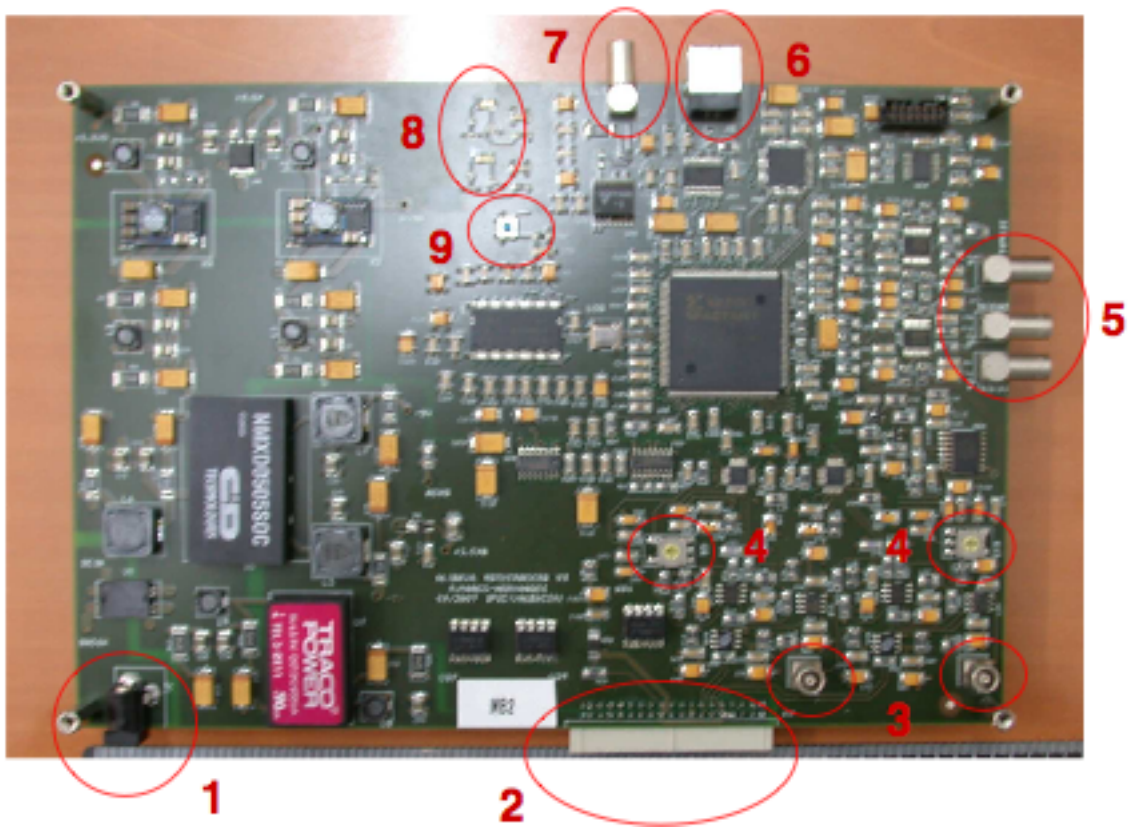

Figure B-2. Connectors, switches and LEDs of the ALIBAVA motheboard.

Laser setup connections

In order to take laser acquisitions the motherboard has a trigger output which uses the LEMO connector next to the USB connector (item 7 of Figure B-2) for exciting a pulse generator which will drive the laser. The name of this output (TRIG OUT) is printed on the motherboard next to the corresponding connector.

The input of the pulse generator must have a 50Ω termination and a coaxial cable of 50Ω must be used for this connection. The levels of this motherboard output are 3.3V LVTTL/LVCMOS compatible.

Radioactive setup connections

For this setup the motherboard has three trigger inputs which use three LEMO connectors (item 5 of Figure B-2). The name of each trigger input is printed on the motherboard next to the corresponding connector.

The TRIG IN1 and TRIG IN2 trigger inputs are intended for signals coming from a photomultiplier. They are terminated with a 50Ω resistor and the input range is ±5V (do not exceed this input range). Therefore a 50Ω coaxial cable should be used for these inputs. These inputs are discriminated using a discrimination level. Look the software documentation for configuring the discrimination threshold of each input (Trigger Configuration). These inputs can be ORed or ANDed (coincidence) once thay have been discriminated. Look the software documentation to carry out this configuration (Trigger Configuration).

The TRIG PULSE IN input is intended for a digital current/voltage pulsed signal (for instance a signal photomultiplier signal discriminated externally). It can accept positive and negative pulses (NIM logic, CMOS logic and TTL logic). This input is terminated with a 50Ω resistor and the input range is ±5V (±100mA). This input has a positive threshold and a negative threshold in order to distinguish the input signal levels (for example if a 3.3V LVCMOS logic is used, a valid value for the positive threshold could be 1000mV and -1000mV for the negative threshold). Look the software documentation to configure these thresholds (Trigger configuration).

The selection between the trigger inputs can be carried out with the software as well (Trigger Configuration).

Probing the Beetle ouput signals with an oscilloscope

There are two analogue outputs at the motherboard in order to probe the analogue output signal of each Beetle chip before they are digitized. These output signals use the two vertical LEMO connectors of the motherboard (item 3 of the Figure B-2). These outputs must be connected to the 50Ω input of an oscilloscope. A 50Ω coaxial cable must be used for these connections.

ADC input range modification

There are two rotary switches of three positions on the motherboard (item 4 of Figure B-2) to modify the ADC input range for each Beetle chip. The position 2 corresponds to an input range of ±512mV (with a resolution of 1 mV). The position 3 corresponds to an input range of 1024mV (with a resolution of 1 mV) only for positive signals. The position 1 corresponds to a input range of -1024mV (with a resolution of 1 mV) only for negative signals.

C. Installing the software

Basic installation

Briefly, the shell commands ‘./configure; make; make install’ should configure, build, and install this package. The following more-detailed instructions are generic; see the ‘README’ file for instructions specific to this package.

The ‘configure’ shell script attempts to guess correct values for various system-dependent variables used during compilation. It uses those values to create a ‘Makefile’ in each directory of the package. It may also create one or more ‘.h’ files containing system-dependent definitions. Finally, it creates a shell script

'config.status' that you can run in the future to recreate the current configuration, and a file 'config.log' containing compiler output (useful mainly for debugging 'configure').

It can also use an optional file (typically called 'config.cache' and enabled with '--cache-file=config.cache' or simply '-C') that saves the results of its tests to speed up reconfiguring. Caching is disabled by default to prevent problems with accidental use of stale cache files.

If you need to do unusual things to compile the package, please try to figure out how 'configure' could check whether to do them, and mail diffs or instructions to the address given in the 'README' so they can be considered for the next release. If you are using the cache, and at some point 'config.cache' contains results you don't want to keep, you may remove or edit it.

The file 'configure.ac' (or 'configure.in') is used to create 'configure' by a program called 'autoconf'. You need 'configure.ac' if you want to change it or regenerate 'configure' using a newer version of 'autoconf'.

The simplest way to compile this package is:

1. 'cd' to the directory containing the package's source code and type './configure' to configure the package for your system.

   Running 'configure' might take a while. While running, it prints some messages telling which features it is checking for.

2. Type 'make' to compile the package.
3. Optionally, type 'make check' to run any self-tests that come with the package.
4. Type 'make install' to install the programs and any data files and documentation.
5. You can remove the program binaries and object files from the source code directory by typing 'make clean'. To also remove the files that 'configure' created (so you can compile the package for a different kind of computer), type 'make distclean'. There is also a 'make maintainer-clean' target, but that is intended mainly for the package's developers. If you use it, you may have to get all sorts of other programs in order to regenerate files that came with the distribution.

Compilers and Options

Some systems require unusual options for compilation or linking that the 'configure' script does not know about. Run './configure --help' for details on some of the pertinent environment variables.

You can give 'configure' initial values for configuration parameters by setting variables in the command line or in the environment. Here is an example:

./configure CC=c99 CFLAGS=-g LIBS=-lposix

*Note Defining Variables::, for more details.

Compiling For Multiple Architectures

You can compile the package for more than one kind of computer at the same time, by placing the object files for each architecture in their own directory. To do this, you can use GNU 'make'. 'cd' to the directory where you want the object files and executables to go and run the 'configure' script. 'configure' automatically checks for the source code in the directory that 'configure' is in and in '..'.

With a non-GNU ‘make’, it is safer to compile the package for one architecture at a time in the source code directory. After you have installed the package for one architecture, use ‘make distclean’ before reconfiguring for another architecture.

Installation names

By default, ‘make install’ installs the package’s commands under ‘/usr/local/bin’, include files under ‘/usr/local/include’, etc. You can specify an installation prefix other than ‘/usr/local’ by giving ‘configure’ the option ‘--prefix=PREFIX’.

You can specify separate installation prefixes for architecture-specific files and architecture-independent files. If you pass the option ‘--exec-prefix=PREFIX’ to ‘configure’, the package uses PREFIX as the prefix for installing programs and libraries. Documentation and other data files still use the regular prefix.

In addition, if you use an unusual directory layout you can give options like ‘--bindir=DIR’ to specify different values for particular kinds of files. Run ‘configure --help’ for a list of the directories you can set and what kinds of files go in them.

If the package supports it, you can cause programs to be installed with an extra prefix or suffix on their names by giving ‘configure’ the option ‘--program-prefix=PREFIX’ or ‘--program-suffix=SUFFIX’.

Optional Features

Some packages pay attention to ‘--enable-FEATURE’ options to ‘configure’, where FEATURE indicates an optional part of the package. They may also pay attention to ‘--with-PACKAGE’ options, where PACKAGE is something like ‘gnu-as’ or ‘x’ (for the X Window System). The ‘README’ should mention any ‘--enable-’ and ‘--with-’ options that the package recognizes.

For packages that use the X Window System, ‘configure’ can usually find the X include and library files automatically, but if it doesn’t, you can use the ‘configure’ options ‘--x-includes=DIR’ and ‘--x-libraries=DIR’ to specify their locations.

Specifying the System Type

There may be some features ‘configure’ cannot figure out automatically, but needs to determine by the type of machine the package will run on. Usually, assuming the package is built to be run on the _same_ architectures, ‘configure’ can figure that out, but if it prints a message saying it cannot guess the machine type, give it the ‘--build=TYPE’ option. TYPE can either be a short name for the system type, such as ‘sun4’, or a canonical name which has the form:

CPU-COMPANY-SYSTEM

where SYSTEM can have one of these forms:

OS KERNEL-OS

See the file ‘config.sub’ for the possible values of each field. If ‘config.sub’ isn’t included in this package, then this package doesn’t need to know the machine type.

If you are _building_ compiler tools for cross-compiling, you should use the option ‘--target=TYPE’ to select the type of system they will produce code for.

If you want to _use_ a cross compiler, that generates code for a platform different from the build platform, you should specify the "host" platform (i.e., that on which the generated programs will eventually be run) with ‘--host=TYPE’.

Sharing Defaults

If you want to set default values for 'configure' scripts to share, you can create a site shell script called 'config.site' that gives default values for variables like 'CC', 'cache_file', and 'prefix'. 'configure' looks for 'PREFIX/share/config.site' if it exists, then 'PREFIX/etc/config.site' if it exists. Or, you can set the 'CONFIG_SITE' environment variable to the location of the site script. A warning: not all 'configure' scripts look for a site script.

Defining Variables

Variables not defined in a site shell script can be set in the environment passed to 'configure'. However, some packages may run configure again during the build, and the customized values of these variables may be lost. In order to avoid this problem, you should set them in the 'configure' command line, using 'VAR=value'. For example:

./configure CC=/usr/local2/bin/gcc

causes the specified 'gcc' to be used as the C compiler (unless it is overridden in the site shell script).

Unfortunately, this technique does not work for 'CONFIG_SHELL' due to an Autoconf bug. Until the bug is fixed you can use this workaround:

CONFIG_SHELL=/bin/bash /bin/bash ./configure CONFIG_SHELL=/bin/bash

'configure' Invocation

'configure' recognizes the following options to control how it operates.

'--help'

'-h'

Print a summary of the options to 'configure', and exit.

'--version'

'-V'

Print the version of Autoconf used to generate the 'configure' script, and exit.

'--cache-file=FILE'

Enable the cache: use and save the results of the tests in FILE, traditionally 'config.cache'. FILE defaults to '/dev/null' to disable caching.

'--config-cache'

'-C'

Alias for '--cache-file=config.cache'.

'--quiet'

'--silent'

'-q'

Do not print messages saying which checks are being made. To suppress all normal output, redirect it to '/dev/null' (any error messages will still be shown).

'--srcdir=DIR'

Look for the package's source code in directory DIR. Usually 'configure' can determine that directory automatically.

'configure' also accepts some other, not widely useful, options. Run 'configure --help' for more details.

Notes

1. Robert A. van Engelen and Kyle Gallivan, The gSOAP Toolkit for Web Services and Peer-To-Peer Computing Networks, in the proceedings of the 2nd IEEE International Symposium on Cluster Computing and the Grid (CCGrid2002), pages 128-135, May 21-24, 2002, Berlin, Germany.
2. http://sourceforge.net/projects/pywebsvcs/files/SOAP.py