Alibava

Carlos Lacasta
IFIC

Valencia

This is version 1.0 of the alibava-gui documentation

Use `alibava-gui` to control the ALIBAVA card

Table of Contents

Introduction.

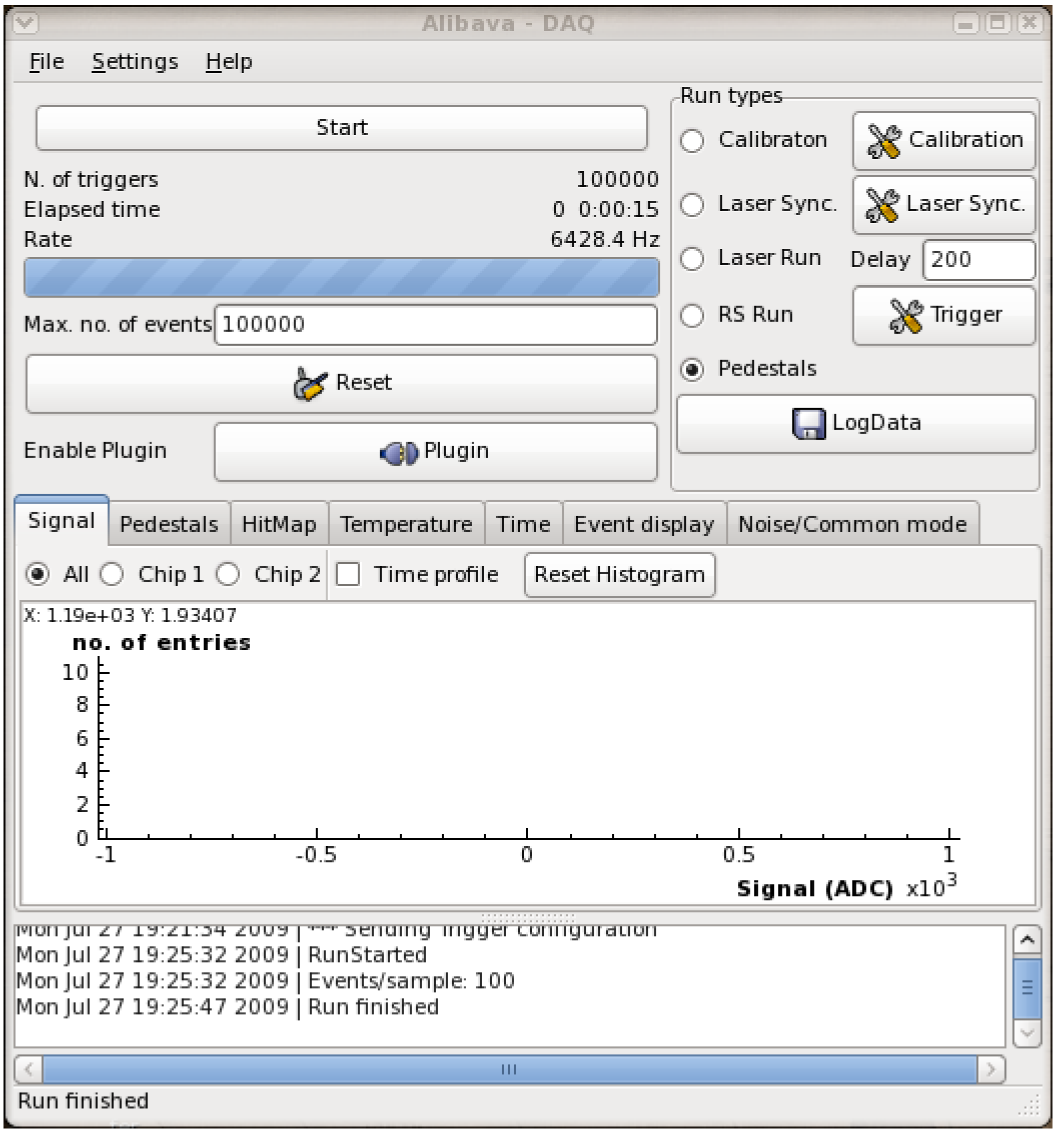
`alibava-gui` is a graphical user interface that controls the ALIBAVA card. It is able to configure the device, receive the data that the card sends via the USB bus and store it in a file for further analysis. `alibava-gui` also monitors the data while in acquisition mode so that the user can detect 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
- Calibration: makes a calibration run injecting calibration pulses to the Beetle chips
- Laser synchronization: scans the delay between the laser pulse and the acquisition
- Laser: makes a laser run
- 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



N. of triggers 100000
Elapsed time 0 0:00:15
Rate 6428.4 Hz
Max. no. of events 100000
Delay 200
Run type: [✓ Pedestals]
X: 1.19e+03 Y: 1.93407

```
Mon Jul 27 19:21:34 2009 | *** Sending Trigger configuration
Mon Jul 27 19:25:32 2009 | RunStarted
Mon Jul 27 19:25:32 2009 | Events/sample: 100
Mon Jul 27 19:25:47 2009 | Run finished
```

Run finished

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.

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).

Starting alibava

Setting up the environment

`alibava-gui` assumes that your computer has the package udev. 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 -G 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*).

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

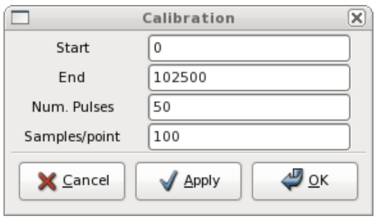
Taking data

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

Figure 2 shows the main parameters that can be set to define the calibration scan.



Start 0
End 102500
Num. Pulses 50
Samples/point 100

Figure 2. Setting properties of calibration scan

Start and end are the calibration charge at the beginning of the run and the last value, respectively. The number of pulses tells how many different calibration amplitudes are going to be generated, while the number of samples per point specifies the number of events that will be acquired for each calibration pulse amplitude.

Laser Synchronization

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.

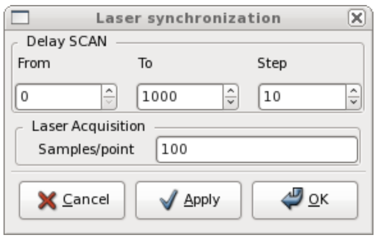


Figure 3. Laser Synchronization scan

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.

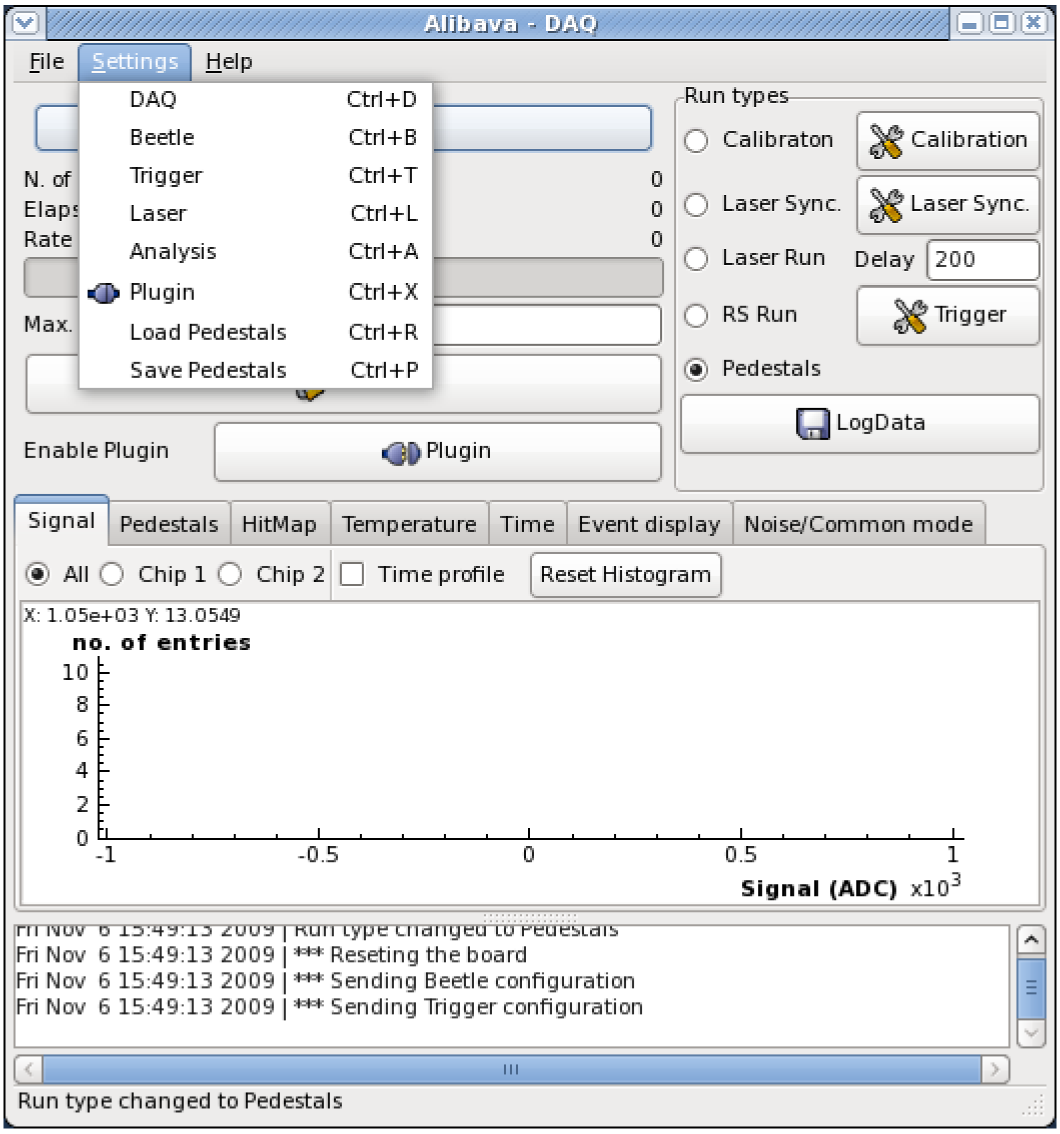
RS, Laser and Pedestal run modes

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 *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.



```
Fri Nov  6 15:49:13 2009 | Run type changed to Pedestals
Fri Nov  6 15:49:13 2009 | *** Reseting the board
Fri Nov  6 15:49:13 2009 | *** Sending Beetle configuration
Fri Nov  6 15:49:13 2009 | *** Sending Trigger configuration
```

Run type changed to Pedestals

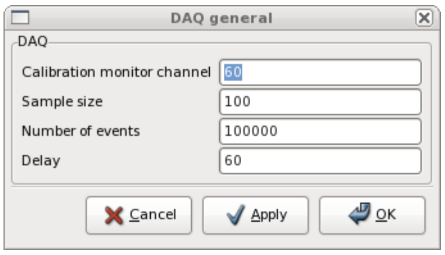
Figure 4. Settings accessible through the Settings item in the main window menu bar

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



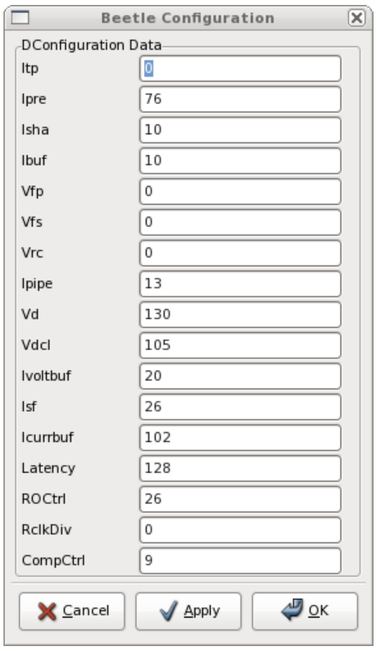
DAQ

Calibration monitor channel	60
Sample size	100
Number of events	100000
Delay	60

Figure 5. DAQ configuration window

Beetle configuration

The Beetle parameters are set by clicking on the Beetle item of the Settings menu as show in Figure 6



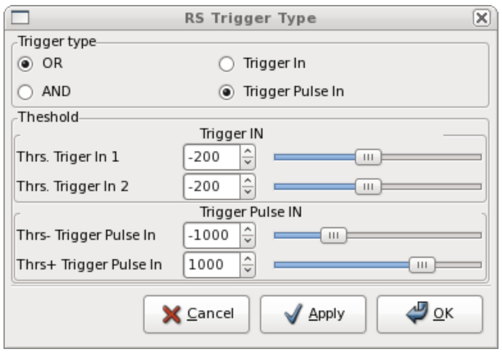
DConfiguration Data

Itp	0
Ipre	76
Isha	10
Ibuf	10
Vfp	0
Vfs	0
Vrc	0
Ipipe	13
Vd	130
Vdcl	105
Ivoltbuf	20
Isf	26
Icurrbuf	102
Latency	128
ROCtrl	26
RclkDiv	0
CompCtrl	9

Figure 6. Beetle configuration

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 7



Trigger type
[✓ OR] AND
Trigger In [✓ Trigger Pulse In]

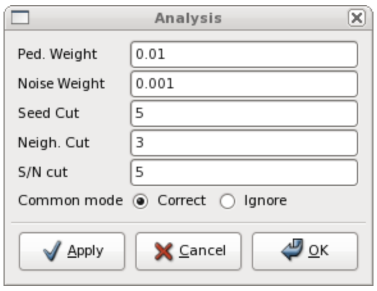
Theshold
Trigger IN
Thrs. Triger In 1 -200
Thrs. Triger In 2 -200
Trigger Pulse IN
Thrs- Trigger Pulse In -1000
Thrs+ Trigger Pulse In 1000

Figure 7. Trigger configuration

Units are in mV.

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 8

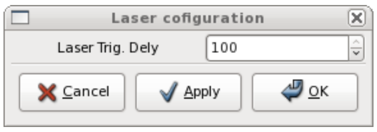


Ped. Weight	0.01
Noise Weight	0.001
Seed Cut	5
Neigh. Cut	3
S/N cut	5
Common mode	[✓ Correct] Ignore

Figure 8. 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.



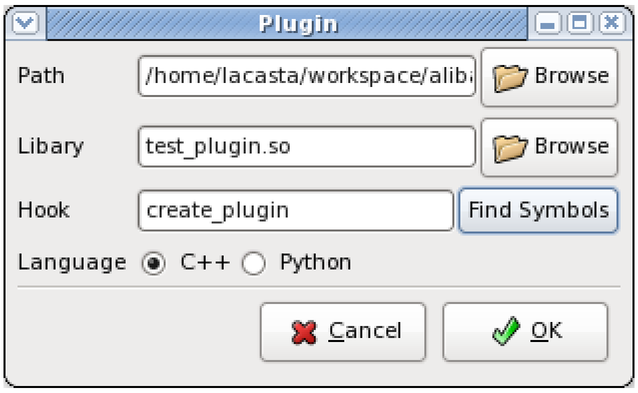
Laser Trig. Dely 100

Figure 9. Laser config

Units are in nano seconds

Plugin configuration

The plugin configuration dialog box appears in Figure 10.



Path: /home/lacasta/workspace/alib
Libary: test_plugin.so
Hook: create_plugin
Language: [✓ C++] Python

Figure 10. 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

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.

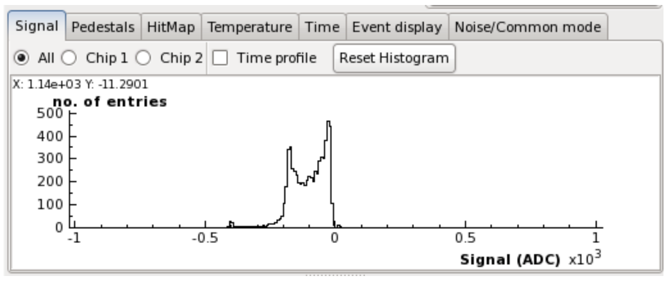


Figure 11. Signal histogram

Figure 11 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 12.

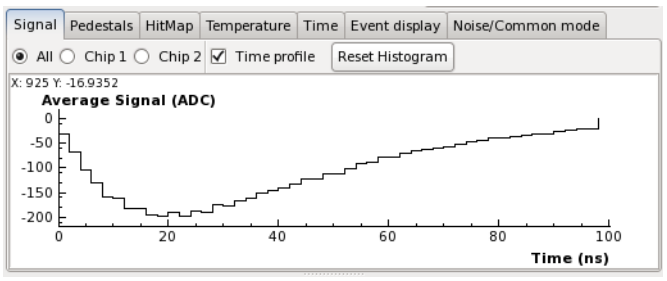


Figure 12. Signal histogram

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.

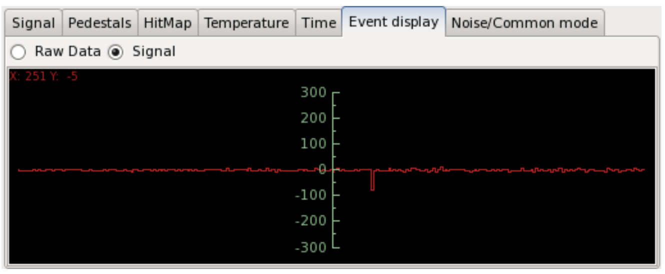


Figure 13. Event display

Figure 13 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.

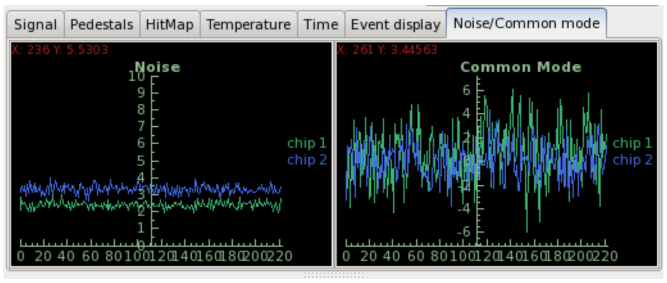
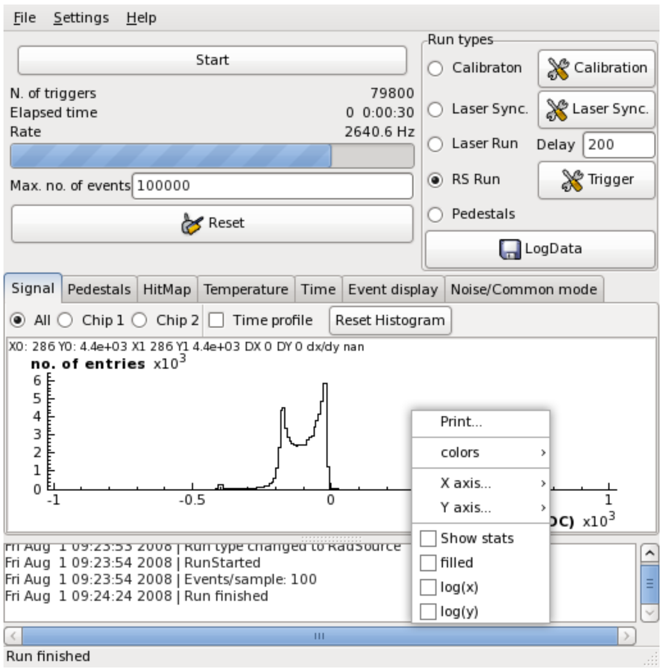


Figure 14. Noise and common mode tracer

Figure 14 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 15 .



N. of triggers 79800
Elapsed time 0 0:00:30
Rate 2640.6 Hz
Max. no. of events 100000
Delay 200

```
X0: 286 Y0: 4.4e+03 X1 286 Y1 4.4e+03 DX 0 DY 0 dx/dy nan
```

```
Fri Aug  1 09:23:53 2008 | Run type changed to RadSource
Fri Aug  1 09:23:54 2008 | RunStarted
Fri Aug  1 09:23:54 2008 | Events/sample: 100
Fri Aug  1 09:24:24 2008 | Run finished
```

Run finished

Figure 15. Histogram menu

Clicking on the print item will let you save the histogram as a picture. 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, 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, event, change the data format (for instance filtering out unwanted channels)
5. End of run

Figure 17 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 string new_file();
  virtual int start_of_run(int run_type, int nevts, int sample_size);
  virtual bool new_event(int ievt);
  virtual string new_point();
  virtual string end_of_run();
  virtual string filter_event(const EventData & data);
}
```

The main methods in a Plugin class are:

`string new_file();`

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 *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* idea 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.

```
string end_of_run();
```

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 *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.

```
string new_point();
```

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 *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 AsciiRoot class*), to access the CheckPoint data you will have to write your own class deriving from AsciiRoot implementing the method `check_point`.

```
string filter_event(const EventData & data);
```

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 *Data format*). I 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.

Warning

If you change the BlockData format then you will have to use a class which derives from AsciiRoot (see the Section called *The AsciiRoot 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.

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
         */
        std::string new_file();
        int start_of_run(int run_type, int nevts, int sample_size);
        std::string end_of_run();
        bool new_event(int evt);
        std::string new_point();
};

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

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;
}

std::string MyPlugin::end_of_run()
{
    std::string rc("end_of_run");
    std::cout << "end_of_run" << std::endl;
    return rc;
}

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

std::string MyPlugin::new_point()
{
    std::ostringstream ostr;
    ostr << "new point: " << current_event << std::endl;
    std::cout << ostr.str();
    return 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 (User-Makefile) 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,


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 16. 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 commands 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, three of those incarnations of USBport which are USBd2xx, USBserial and USBFifo.

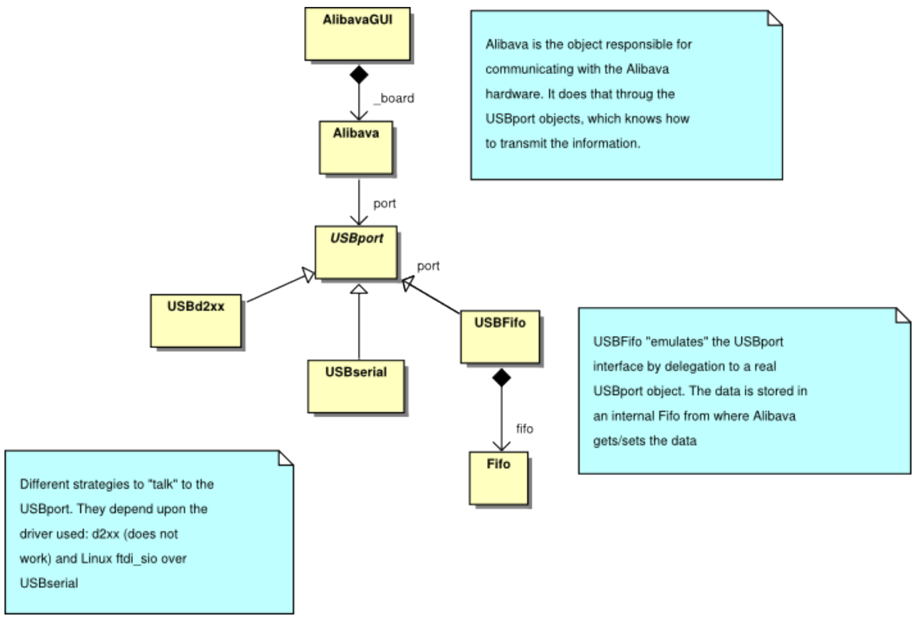

AlibavaGUI
_board
Alibava
port
USBport
port
USBd2xx
USBserial
USBFifo
fifo
Fifo
Alibava is the object responsible for communicating with the Alibava hardware. It does that throug the USBport objects, which knows how to transmit the information.
USBFifo "emulates" the USBport interface by delegation to a real USBport object. The data is stored in an internal Fifo from where Alibava gets/sets the data
Different strategies to "talk" to the USBport. They depend upon the driver used: d2xx (does not work) and Linux ftdi_sio over USBserial

Figure 16. 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 alibava-gui is `USBFifo` using `USBserial`.

The DAQ loop

The DAQ loop is sketched in Figure 17. 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 17 we have illustrated a calibration run, but the behaviour is the same for other run modes of `alibava-gui`.

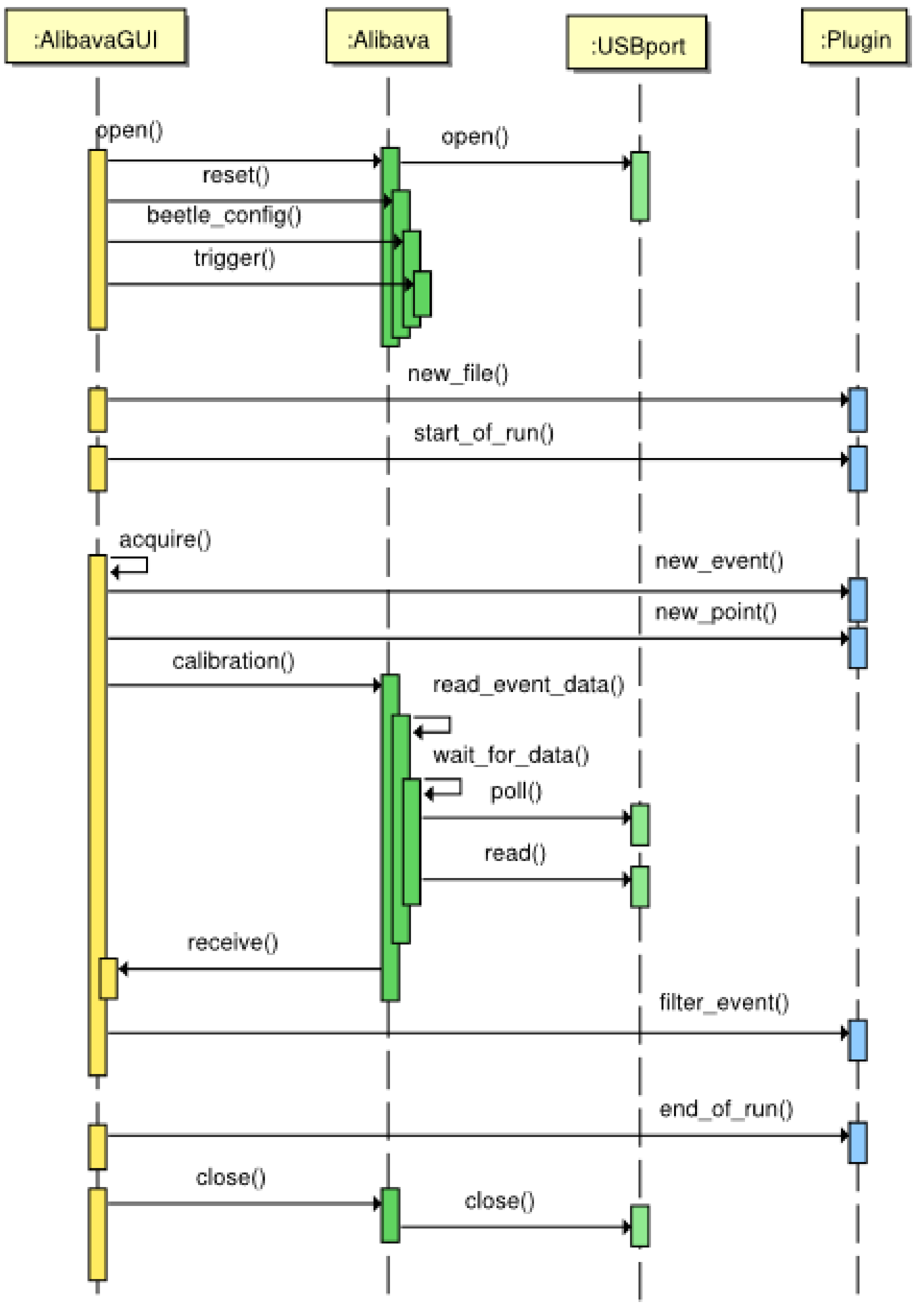


Figure 17. DAQ loop

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 18 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.

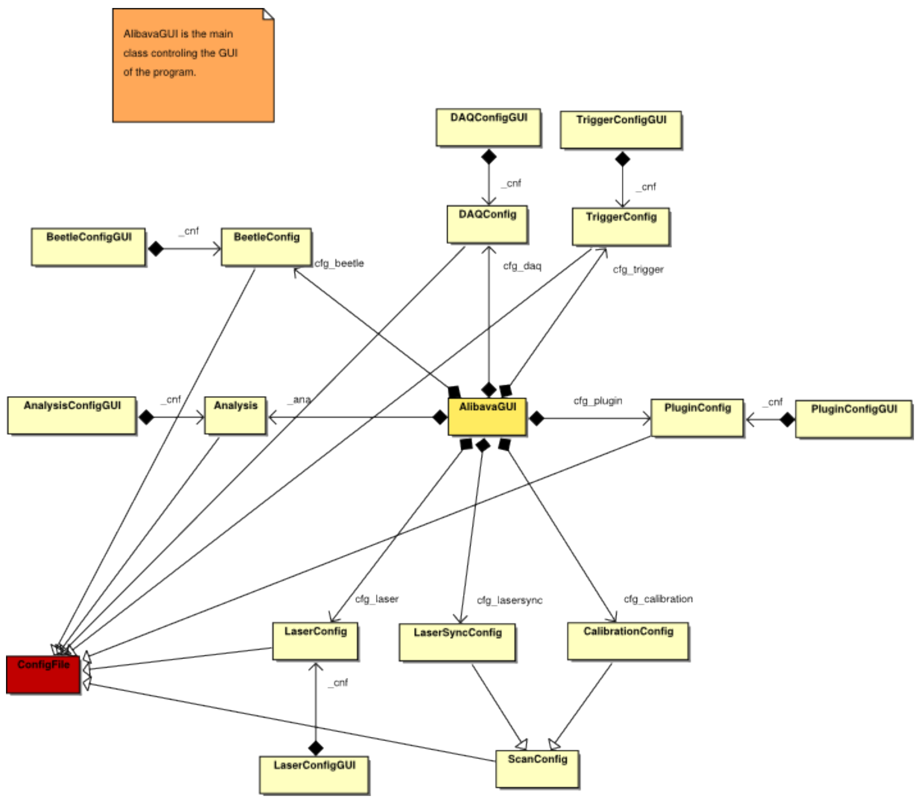


Figure 18. Configuration objects

Data analysis.

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 we use in the tables:

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

Data size and type	Meaning
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)
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

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	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.

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 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 A.

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

```
//()
{
   // Add INSTALL_DIR/lib/alibava/root in the ROOT macro path
   const char *install_dir = "INSTALL_DIR/lib/alibava/root";
   char line[1024];

   sprintf(line,"%s:%s", gSystem->GetDynamicPath(), install_dir);
   gSystem->SetDynamicPath(line);

   sprintf(line,"%s:%s", gROOT->GetMacroPath(), install_dir);
   gROOT->SetMacroPath(line);

   // now load the things
   std::cout << "==================================" << std::endl;
   gROOT->LoadMacro("compile.C");
   load_alibava_libs();
   std::cout << "==================================" << std::endl;

   // Add your own ROOT initialization stuff below

}
```

If you want to make modifications to the source of the ROOT macros then, to use them, go to the root_macros folder and create there a rootlogon.C file that looks like Example 6.

Example 6. rootlogon.C file

```
//()
{
    std::cout << "==================================" << std::endl;
    gROOT->LoadMacro("compile.C");
    compile(false);
    std::cout << "==================================" << std::endl;

    gROOT->SetStyle("Plain");
    gStyle->SetPalette(1);
}
```

This will load some of the macros that will analyze the data.

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

```
sin_preguntas(data_file, cal_file, polarity, dofit);
```

Example 7. The make-all-for-you function prototype

```
void sin_preguntas(AsciiRoot *A, const char* cal_file, const char
*ped_file, int polarity, bool dofit, int tcd0, int tdc1);
```

where the arguments have the following meaning:

A

a pointer to a user supplied AsciiRoot (or descendant) object. Here one should pass either a pointer to either a pure AsciiRoot class or to a user defined "son/daughter" of AsciiRoot defining the virtual methods used to decode the special data blocks. See the Section called *The AsciiRoot class*

data_file

The path of the data file to be analyzed.

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

The AsciiRoot 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 AsciiRoot class can still serve as a good tool to read the files and to access the current data to make your own analysis.

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

Example 8. The AsciiRoot class definition

```
 class AsciiRoot {
  AsciiRoot(const char * data_file);
  ~AsciiRoot();
  enum BlockType = {NewFile=0, StartOfRun, DataBlock, CheckPoint,
            EndOfRun};
  bool valid();
  void open(const char * data_file);
  void close();
  void rewind();
  int read_event();
// Plugin extra data Blocks
  virtual void new_file(int size, const char * data);
  virtual void start_of_run(int size, const char * data);
  virtual void check_point(int size, const char * data);
  virtual void new_data_block(int size, const char * data);
  virtual void end_of_run(int size, const char * data);
  void set_data(int size, const unsigned short * data);
// Analysis methods
  TH2 * compute_pedestals(int mxevts = -1, bool do_cmmd = true);
  void compute_pedestals_fast(int mxevts = -1, double ped_weight = 0.01, double noise_weight
  void load_pedestals(const char * file_name);
  void save_pedestals(const char * file_name);
  void load_gain(const char * file_name);
// Debugging methods
  void spy_data(bool with_signal = false, int nevt = 1);
  TH1 * show_pedestals();
  TH1 * show_noise();
}
```

By default, AsciiRoot 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.

`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).

`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).

`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 nois
```

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`.

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

A. 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.