ALIBAVA
SYSTEMS

Vdaq

The Alibava Data Acquisition program

Edifici Eureka. Campus UAB. 08193 Bellaterra, Barcelona, Spain
Tel +34 93 586 88 32 info@alibavasystems.com
www.alibavasystems.com

Vdaq

The Alibava Data Acquisition program

The Alibava Systems

This document describes Vdaq, the Alibava Data acquisition system. The document will describe the different components of the GUI that allow to control the acquisition process as well as the data structure. It will also describe the software that Alibava provides to read the data from the files.

Table of Contents

1. Introduction

1.1. What is Vdaq

Vdaq is an application to drive the data acquisition of a number of instruments. The application tries to be modular, in the sense that different instruments are handled by plugins that are dynamically loaded, providing a number of hooks to configure and monitor them.

Vdaq is based on DAQ++. DAQ++ is a C++ framework to develop data acquisition programs. I will not enter into the details of this. I will just mention the main *objects* defined in DAQ++. Those are the DAQmanager, the RunManager and the Module. The DAQmanager controls the data acquisition process and is a kind of *main gate* to access the DAQ internals. The RunManager is the object that controls the acquisition mode. We may want to have different types of runs, like for instance pedestal runs, scans on important parameters to produce curves like the gain of an ASIC, etc. It will distribute the acquisition commands (like start, stop, reset, etc,) to the Module objects (to be explained in a minute) and it is also what DAQ++ calls a data receiver. It will receive the data from other objects in the acquisition. We finally have the Module objects. These are the interface with the hardware. They *know* how to get the data out of a piece of hardware either through VME, USB, PCI, etc. They are, therefore, called data producers. Figure 1 shows how the DAQ++ objects are organized. Vdaq is a graphical user interface to this framework.

As already said, the different ways in which those instruments are handled during a run, or acquisition session, is controlled by a set of DAQ++ RunManager objects. There is a default set of them, but the user can also write his/her own and override the default Vdaq behavior. All this is driven by a configuration file which will be described later.

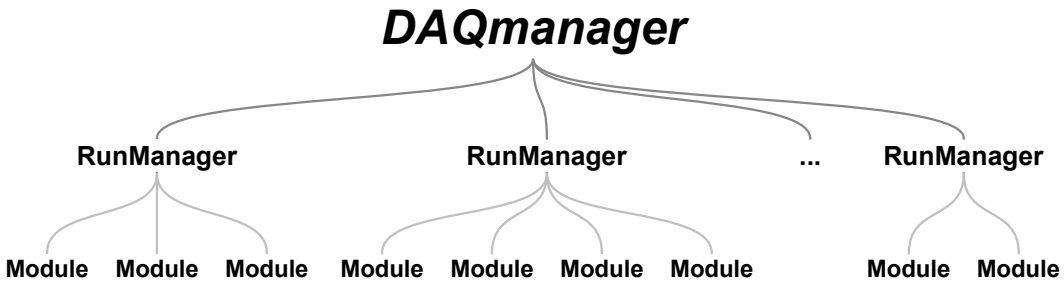

Figure 1: The DAQ++ object hyerarchy

Vdaq can handle the modules in different ways or styles. In other words, Vdaq provides various run modes according to:

- Trigger: The trigger can be selected to be either external or internal, where internal means that each module finds the way of generating whatever is needed to produce data by software or any internal means, and external refers to the case in which something else is generating the data for us.
- Interrupt: This is probably an *unfortunate* term, but it refers to the way in which the program realizes there is some data in the detector. There are two possibilities here as well
  - Poll: the program keeps on asking if there is data until there are some in the detectors, and
  - Interrupt: the program sleeps until some mechanism awakes it and then tries to read the data.

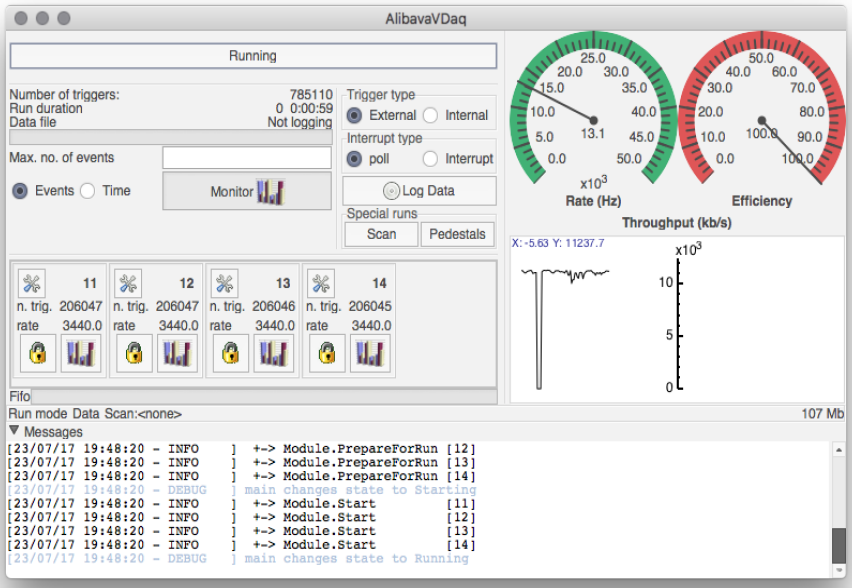

Figure 2: Vdaq main window

Figure 2 shows the main window of Vdaq. The different components are, on the right a couple of *speedometers* that show the instantaneous acquisition rate and the efficiency, which is the percentage of the events registered which are dumped to file (if logging data). Below those objects, there is a tracer showing the throughput in kb/s.

On the left there is a button that shows the state of the acquisition and, also, controls the acquisition. When clicking that button, a pop-up menu will appear with a selection of possible DAQ commands to issue. Below the run menu there are a number of indicators showing the number of triggers seen so far, how long the run has been taking data and the name of the output data file.

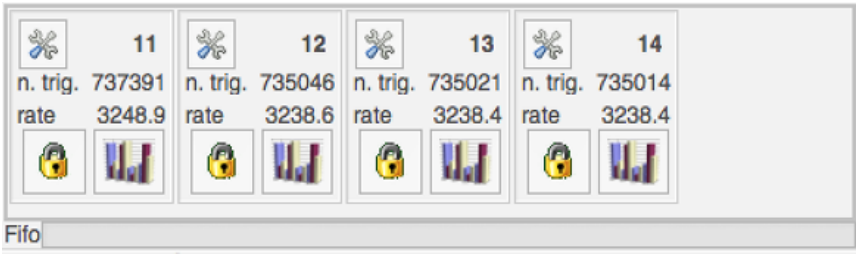

Figure 3: Modules in the main display

There is a small window at the bottom where all the devices configured are shown. This is shown with more detail in Figure 3. Each small window corresponds to a module. On the upper right corner, in bold letters, one sees the module identifier. At the left of this label there is a small button which, when clicked, will open a window to configure the module. If the module plug-in does not implement that, no window will appear. The button with a lock on it is to put the module in local. What is that ? It may happen that you have a configuration file with many modules defined, but in a given moment you only want to run with a few of them. In that case, click there, and if the module is *un-locked* it won't participate in the run. At the right of the lock there is another button that when clicked will open the monitoring window for that module (see section Monitoring). If clicked again, it will close the monitor window.

1.2. How to start Vdaq

To start Vdaq just type:

```
Vdaq [options] [config_file]
```

where `config_file` is the name of a file in which we have stored the Vdaq configuration (see section Configuring Vdaq). the most common and the options are:

--soap-port=port_number	The port where the SOAP server will be serving requests.
--no-gui	Does not start the graphical interface. The program will then run as a server which can be accessed via SOAP.
--no-cfg	Do not load the default configuration file if none is given at input. Vdaq will start with no modules loaded. This can be useful to create a new configuration from scratch.

2. Using Vdaq

2.1. Starting/Pausing/Stopping a run

Starting a run is quite easy. Be sure that all your modules are properly configured and click on the long, horizontal button in the main window (see Figure 2). A menu like the one showed in Figure 4 will pop up. You should click on start.

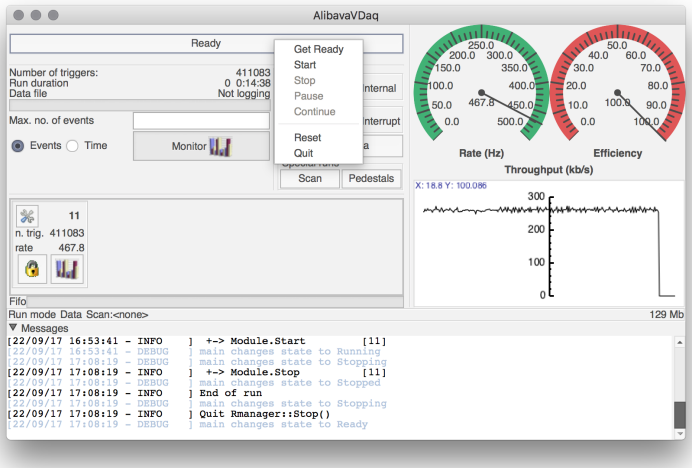

Figure 4: Run menu

Depending on the options selected for trigger type and interrupt the run will behave on one way or another. One can request the run to finish after a maximum number of triggers are registered if that number is specified in the Max. no. of events text entry. Otherwise Vdaq will run until it is stopped with the run menu button.

There are special run types, like scans and pedestal runs which are activated and/or configured with the Scan or Pedestal buttons (see section Run types).

2.2. Configuring Vdaq

The program is configurable and remembers the last settings. This is done with the help of a configuration file. The configuration file is stored in the default configuration folder. This is usually

1. $HOME/.config/Vdaq in Unix systems and
2. C:\Documents and Settings*username*\Local Settings\Application Data\Vdaq in windows systems.

To load a configuration file when the program is already started, use the Open item in the File menu.

The current settings can also be saved in a configuration file in order to retrieve them in a later session. This is done with the Save item in the File menu. A dialog box will pop-up. In the dialog box you will see a list of existing configuration files. To overwrite one of those, select it and click OK. To save the configuration in a new file, type the name of the file in the text box at the bottom of the dialog.

2.3. Acquiring data

If you want to log data into a file (the most common situation, I guess) you should set up the program to do it. Vdaq gives fixed names to the data files. The names have, however, a structure that will allow to differentiate one run from the other. The structure is the following:

```
base_name+run_number+"_"+file_number+".data"
```

The base_name can be set in the Settings menu, IO control tab (Figure 5). The run number is automatically incremented by Vdaq every time a new run is started, although the starting value can also be set in the IO control tab.

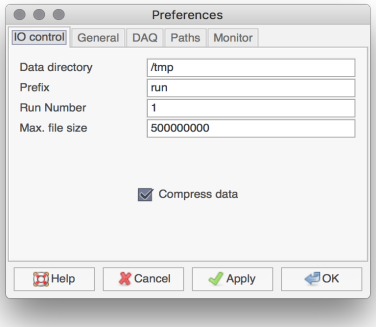

Figure 5: Input/Output settings

Vdaq will never let a file grow indefinitely. There is a maximum size a data file may have and this is also set in the IO control tab. Once a data file has reached this maximum size, it will be closed and a new one will be opened, with the same base_name and run number, but with the file_number incremented.

2.4. General parameters

The Settings menu allows to set some other general parameters used by Vdaq as shown in Figure 6.

Here one can set a few parameters that define how Vdaq will handle the calculation of pedestals. One can set the number of events that will be acquired in a Pedestal run. This can be set to -1 which means that the run will stay until stopped manually. This may be useful if you want to spy in the monitor window how stable and reliable are your pedestals.

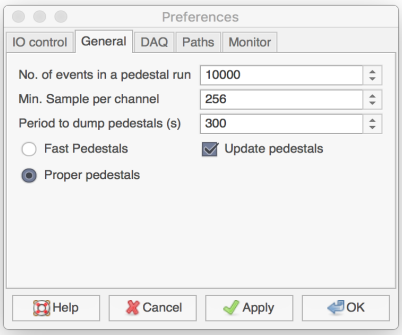

Figure 6: General settings

One can also specify whether pedestals and noise will be updated during monitoring and, finally, how the pedestals will be computed. The pedestals will be dumped in the data file with a fixed period that can be set in this dialog. The minimum number of samples per channel is only useful if one chooses the proper method for the pedestals. This is further explained in section Pedestals.

> **Warning**
>
> Vdaq will only update and, to a some extend, compute pedestals if monitoring is enabled. Otherwise, the only way of doing it is to make a pedestal run or load the pedestals from a file

2.5. Pedestals

The pedestals can be uploaded or saved into a file. Usually when logging data into a file the pedestals are also saved there, but it may happen that you have pedestals that have been calculated before hand and you want to use them as the starting point of your acquisition. This can be done with the dialog window that pops up when selecting either Save pedestals of Load pedestals from the Settings... menu as shown in Figure 7.

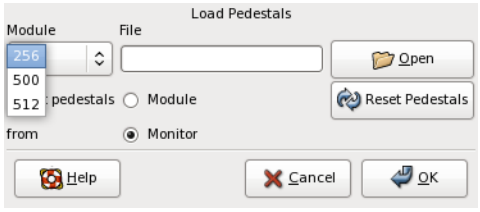

Figure 7: Pedestals Menu item

On this dialog one should select the module for which he/she wants to load or save the pedestals, the file where the pedestals should go or come from and, finally, choose which pedestals to save. This may require a few extra words on how pedestals are handled in Vdaq.

The pedestals are computed **only during monitoring** or during a **Pedestal Run**. While monitoring they are stored in an internal buffer of the monitor. Only after a Pedestal run the pedestals are copied into the module. On this dialog we can choose whether we want to operate with the pedestals stored in the monitor or the ones stored in the module.

> Choosing to load the pedestals into the monitor will not modify the pedestals on the module and after clicking on the Reset button in the monitor window they will be overwritten by the values in the module.

The behavior of Vdaq concerning the pedestals when a new run is started is as follows. If we have already pedestals computed from a previous run or uploaded from a file, Vdaq will start with those and update them if required. If there are no pedestals at all, and updating is allowed, then Vdaq will spend the first events to compute the pedestals, and only when a given channel has a pedestal calculated with a minimum of the minimum sample that we set in the dialog shown in Figure 6 will be displayed.

Pedestals can be computed with two methods:

- Fast method: Vdaq assumes that the initial value of the pedestals is 0 and starts updating the pedestals and the noise. This method's performance in terms on convergence and stability depends on the values given in for the weighting parameters in the module configuration.
- Proper or normal method: Vdaq uses the first events to get an estimate of the pedestals. The estimate is just the mean value over a minimum number of sampled events. From there on, Vdaq switches to the update method.

2.6. DAQ settings

There are a number parameters that control some of the DAQ aspects. Those can be set from the Settings menu, clicking on Preferences and choosing the DAQ tab. The dialog window is shown in Figure 8

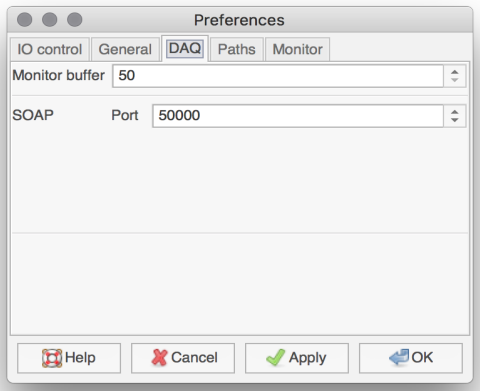

Figure 8: DAQ settings

The monitor buffer text entry specifies the maximum number of events that the monitoring buffer will store for further analysis. Vdaq inherints from the DAQ++ library a SOAP server which can be used to monitor the status of the data acquisition remotely or, even, control the data acquisition. The port where the SOAP server will serve the user requests can be set here.

2.7. Setting the paths to search for libraries, configuration files, etc.

Vdaq will search for various objects which are spread in the file system. One can specify which are the paths that Vdaq should explore to find those objects while running. This is done from the Settings menu, clicking of Preferences and choosing the Paths tab.

There are three different kind of paths: library paths to search for plugins, glade paths to search for glade files and config paths to search for configuration files. This can be selected from the combo on the upper right part of the dialog.

Once a path type has been selected, the paths will be listed in the dialog window. One can add a new path or delete a selected path. When selecting a path, it can be moved up and down to establish the proper search order.

2.8. Plugins

Vdaq is a modular application that can be extended with a number of plugins. The plugins can be added to define new modules (the objects that are able to communicate with the hardware), run managers (the objects that define the acquisition mode, like for instance pedestals, scans, etc), data loggers (the objects that write data to a file) or interrupts (the objects that discover whether there is data to read from the hardware).

Plugins can be configured from the Plugins menu. A dialog window with a number of tabs will appear. To add a plugin to handle a new piece of hardware one needs to select the Plugin tab. The dialog window is like the one shown in Figure 9.

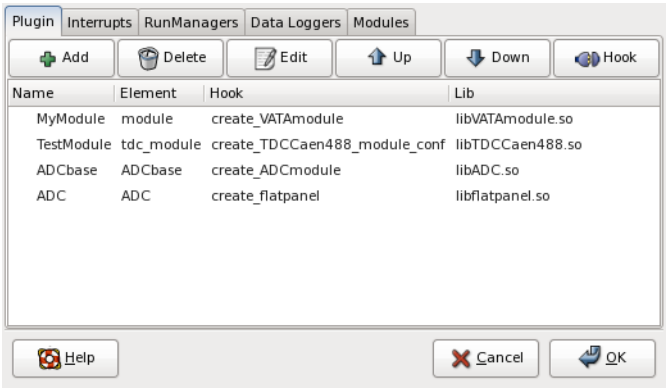

Figure 9: Plugin settings

A plugin is just a shared library that provides a sort of driver for the new hardware that we want to control. It has a name, and element, which is the name we give to that hardware and then a hook, which is the function that creates the module driver and a library where that function is defined. To create a new one, click on the Add button and set the plugin and element names. Then click on the Hook button to define the hook. A window like the one in Figure 10 will appear.

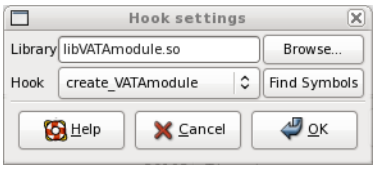

Figure 10: Plugin hook

Set the library name, or browse the file system to find it and then, click on Find Symbols to get the list of functions on that library and select one. This last step may not work if symbols have been stripped from the library.

One can also remove a plugin by selecting it and clicking on Delete.

This is not the end of the process, though. What we have done is to define the piece of software that will handle the hardware objects named by the element value of the plugin. Now we have to tell the system how many of those objects we have. This is done in the Modules tab of the dialog window, which is shown in Figure 11.

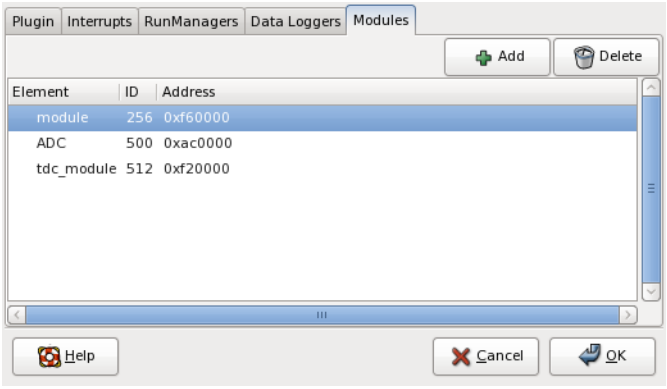

Figure 11: module definition

There you list of hardware objects that you want to have in the system by specifying the driver that will control them (the element value), an identifier and what is called a hardware address, which is usually required by the drives to communicate with the hardware.

To setup interrupts (the objects that figure out when data is ready to be read) select the Interrupts tab in the Plugin menu. A window like the one showed in Figure 12 will be displayed.

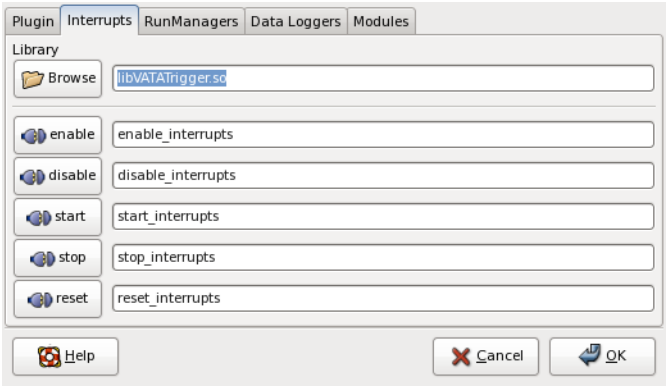

Figure 12: Settings to define interrupts

Type the name or the library or browse the file system to find it and then write the different functions which are needed for the interrupt. If the library has symbols, click on the button at the left of the function name text entry and you will get a list of functions in the library.

There are several elements that will define the behaviour when confronted to an interrupt. See the section What is Vdaq to see what Vdaq understands as interrupt. The meaning of the different functions is:

- *enable* is the name of a routine that will take the necessary actions to leave the system in a state ready for the detection of trigger conditions. However, after calling this, Vdaq may not be sensitive to them yet.
- *disable* is the name of the routine that will disable detection of interrupts.
- *start* is the name of a routine that will leave Vdaq in a state sensitive to interrupts.
- *stop* will stop detection of interrupts, although this is different to *disable* in the sense that Vdaq can still do it after calling the function in *start*
- *reset* provides the name of a routine that should leave the system in a state suitable to a start a new run. The system may not be sensitive after this call. Only when *start* is called the system should be sensitive to them.

As explained in section Run types there are several types of run managers. Vdaq offers default implementations which are, usually, just enough, but the user can define his/her own run managers.

The possible run manager types are:

- data: this is the normal run type
- pedestal: this is used for the pedestal runs
- int_trg: this is used when internal trigger is selected as run acquisition mode in the main window
- *pulse*: this is used when we want an external "pulse" or "stimulus" generator
- scan: this is used for the scan runs

Then, from the RunManagers tab of the Plugins menu one can load those new implementations. The dialog window is show in Figure 13.

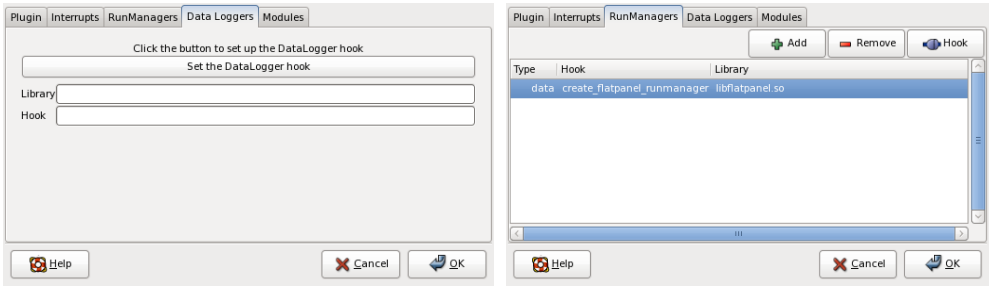

Figure 13. Datalogger (lefft)a nd Run Manager (right) configuration tabs.

Run managers can be added or removed clicking the appropriate buttons. To define the hook, that is, the function that creates the new run manager and the library where it can be found can be set by clicking on the Hook button, as explained for the module plugins.

Finally, data loggers (the objects that write the data on file) other than the default can be loaded from the Data Loggers tab in the Plugins menu as shown in Figure 13.

3. Run types

3.1. Introduction

Vdaq can make different types of runs:

1. Data RunManager
2. Pedestal Run
3. Scan

3.2. Normal Data Run

A normal data run is a run in which we just want to take data. This is the default when starting a run from the run menu. The only exception is a Scan run that will be described in section Scans.

3.3. Pedestal Run

A pedestal run is, a priory, like a normal data run but in this case we force monitoring every event so that Vdaq can calculate the pedestals over the full sample. Also, the minimal sample to trust the pedestals is set to the total number of events we plan for the pedestal run. When the run is over, the pedestals are saved in the module itself.

Pedestal runs are started by clicking the Pedestal button in the main window. The main parameters of a Pedestal Run are set through the Settings... menu item, on the General tab, as described in section General parameters.

3.4. Scans

Vdaq can make scans on a number variables:

- pulse
- threshold
- channel
- trim

In order to define a scan, click on the Scan button in the main window. A dialog like the one shown in Figure 14 will appear.

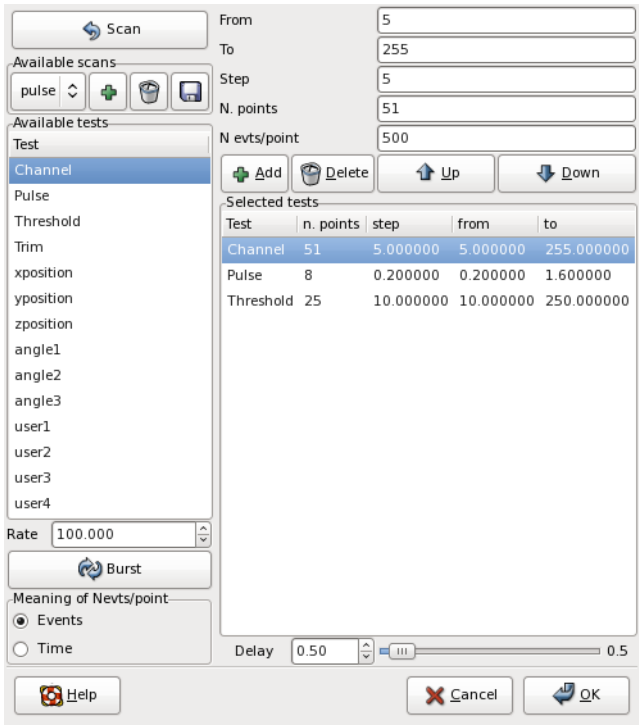

Figure 14: Scan definition dialog window.

One can define a scan with the help of this menu. On the left there is a list of the variables available for the scan. When you chose one of them, the range of the scan for that variable will appear on the text entries on the right. There one can set the number of points, the step, etc. When ready, click on the Add button and the current values will appear on the list below. One can select any of the variables in the Selected Tests frame. Its values will, again appear on the text entries where one can modify them. Again, in order to update the values, one should click on Add. To delete a variable from the list, select it and click on Delete. The order of the scan is that the lower variable corresponds to the innermost loop, and the upper variable to the outermost loop. One can change the order with the Up and Down buttons.

When the scan is ready, one can save it with a given name, so that it will be stored in the configuration and can be restored at a later run. To do that, click on the plus button on the Available scans list. Give the scan a name, and click on the save button to save it. In the combo box you will find the existing scans. You can select any of them and make it the active one for this run.

To activate the scan run, do not forget to click the Scan button on the upper left corner of the dialog.

4. Monitoring

4.1. Monitoring

Vdaq allows to monitor the data during the acquisition. This is, certainly, time consuming and if the acquisition rate should be high, one should foresee the possibility of disabling this feature.

Monitoring can be enabled or disabled by means of the monitoring button in the main window, as shown in Figure 2. Disabling monitoring will speed up the acquisition, if the rate is very high, but one will not be able to see how the acquisition proceeds. One should, therefore, decide, according to the acquisition rate, what to do.

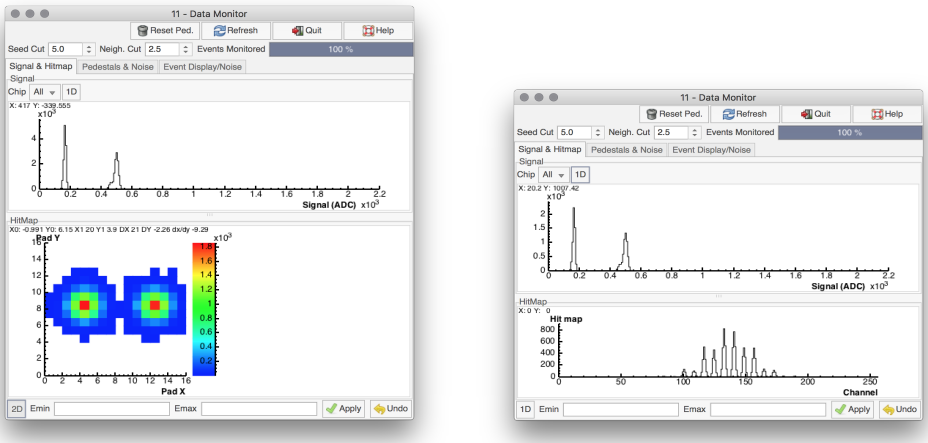

Figure 15. Main tab in the monitoring window. The spectrum is shown on the top. The bottom shows the hitmap. If there is information about the mapping of the channels one can see it in 2D (left) otherwise it is only shown in 1D.

If monitoring is enabled, one can see, for each module, the results of monitoring. This consists, usually, on pedestals, noise, hitmap and event display. The main monitor window is shown in Figure 15 and Figure 16. To show this window, one clicks on the small button with a histogram drawn in the small window corresponding to the given module (see Figure 3). The window has three tabs that show different things: Signal and hitmap, pedestals and noise and, in the last one, the event display and other properties displayed in the form of trace plots.

Figure 15 shows the spectrum at the top and the hitmap at the bottom. Both the hitmap and the signal histograms can be visualized in 1D or 2D. To toggle between the 2 modes press the small buttons labeled 1D (or 2D) by the histograms.

> **Warning**
>
> The hitmap histogram can only be shown in 2D if there is a padmap in the configuration file that will help translating electronic channels into pixels.

The signal histogram can be show as the signal distribution of all the channels, only the channels of a given chip or as a function of the channel number, that is, in 2D. This is done by selecting the corresponding option from the menu onn top of the histogram.

Figure 16 show the third tab of the monitoring window. It shows several properties as a funtion of time at the top like noise, common mode noise, temperature, etc. At the bottom it show, for a given event, the segnal seen by each channel or, in other words, the event display.

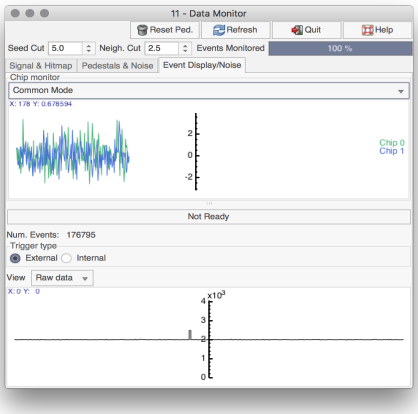

Figure 16. The tab of the monitor window showing the common mode noise of the ASICs as a trace plot at the top and the event display (the signal of each channel for a given event) at the bottom.

There is an extra feature that links the signal and the hitmap. One can select a signal range on the signal histogram with the left button of the mouse. The limits of the range will appear in the text entries below the hitmap histogram. When pressing the Apply button, the 2D hitmap will only display the pixels with counts within that range. Clicking on the Undo button will restore the hitmap histogram.

In the case of 1D histograms one can magnify one of the axis, or both. You can choose a given range in any of the axis by selecting it with the left button of the mouse while pressing:

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

The original range is restored pressing the mouse middle button. Click the mouse right button to explore the popup menu and change some of the histogram attributes.

For a 2D histogram, one can select the range of Z values by moving the pointer on the color scale while pressing the left button of the mouse. To restore the original scale, click on the middle button of the mouse.

5. Vdaq data format

Vdaq stores the data using the HDF5 library.

The data in the output file is organized in different blocks:

- header
- modules
- scan

Use HDFView to see and browse the data format. It will help understanding the text below.

5.1. The header.

The header has an attribute with the definition of the run. The run description contains the type of run, the run number the number of events and the time. The header gruop has as attribute a run_record structure (see Table 1).

The header contains 2 groups that describe the modules that take part of the run and the key moments of the run.

5.1.1. /header/modules

This is a list with the module descriptions. They are encoded as 0xnn0tiii with

- nn = number of chips
- t = data type
- iii = module identifier

5.1.2. /header/run_records

Vdaq stores run records every now and again and certainly at the start of run and at the end of the run. The run records contain the following information

Variable	**Description**
run_type	The type of run. Runs can be Data (0), Pedestal (1), Internal Trigger (2), Scan (3) or Pulse (4). See Vdaq manual for a version of it.
run_number	The run number
nevts	The number of events at the time of the writing of the record
time	The time when writing the record. The time is given as an array like [second, minute, hour, day, month, year].

Table 1. Description of variables in the run_records group of the data header.

5.2. The module

The module group has one subgroup per module named with the module ID

/modules/iii

where iii is the module ID.

The module group has the *data* group where the user defined data format resides and the *configuration* array that contains an array of integers with the configurations. This is module dependent. Finally, there is the pedestal group that contains:

/modules/iii/pedestals/pedestal - the pedestals
/modules/iii/pedestals/noise - the noise

5.3. The scan

The scan block has two records, one that defines the scan (/scan/def) and the second that tells when Vdaq changed to the new scan point and the values of the variables.

/scan/def/nevt - number of events per scanpoint
/scan/def/nvar - number o variables to scan
/scan/def/variables - the description of the variables

The scans of the different variables are described by a structure having the members shown in Table 2.

Variable	Description
type	identifies the variable being scanned
npoint	number of points in the scan
from	first value of the scan
to	the last value of the scan

Table 2. Description of the variables that are scanned in a Scan Run.

6. The Vdaq analysis software

With HDF5, writing your own analysis software is not too difficult if the data is properly described. However, we provide a framework that makes the HDF5 handling transparent for the user.

There are 2 main classes that provide access to Vdaq data.

- VdaqModuleData: contains the information relative to an individual module in one of the events. It also provides an iterator that helps to navigate through the module data.
- VDAQData: is the main portal to the data. It opens the file and is able to navigate through the file structure and create VdaqModuleData classes which get the data of the individual Vdaq modules. It also provides a very simple iterator that navigates through all the data of all the modules. Data can be retrieved ordered according to some of the data values like, for instance, time stamp.
  It also maintains a list of all the modules found in the data file and provides ways of accessing those modules by the module id or via iterators.

We also provide iterators to navigate through all the data from all modules (assuming they are all of the same kind). One of the iterators returns the data as it comes (vdaq_iterator) and the second (vdaq_sorted_iterator) sorts the data according to the given compare function.

6.1. The VDAQData class

This is the main class to access the data. It is responsible for opening the data file and it creates the list of modules that have data in the file.

To handle the specifics of each module data format, one should register a module creation function that creates the corresponding instance of the user defined implementation of the VdaqModuleData interface. This is done via register_type().

6.1.1. Methods

This section describes the class most important methods. See the code for additional documentation.

`VdaqData (const char *file_name=0)`

This is the constructor. If a file name is given it will be opened.

`Int open(const char *file_name)`

Opens the given file. This one is usually called when the object instance is created without a file to open or when the operations on a given file are finished and we proceed to analyse a new file.

`void close()`

Closes the file.

`uint32_t nevts() const`

Returns the number of events in the data block.

`bool has_scan() const`

Tells wether the file has scan data.

`VDaqModuleData *module(int id)`

Returns the module with identifier `id`.

`int n_modules() const`

Tells the number of modules in the system.

`ModuleList::iterator begin()`

```
ModuleList::iterator end()
```

The iterators to the list of modules.

The following is a list of template functions used to obtain iterators to go through the data. Note that these do not belongto the VDAQData class

```
template<typename _ModuleData> vdaq_iterator<_ModuleData>
  create_vdaq_iterator(VDaqData &M)
```

Returns an iterator of the data of all the modules in the data block. _ModuleData is the structure defining the data definition. The events are given as they appear in each of the module data block. The iterator iterates over the data of several modules of the same kind. Note that it is a forward iterator, it does not implement the increment or decrement operators. Instead it uses the next method.

The iterator has

1. the module_id() method to know the module from which the current data chunk comes.
2. The iterator acts as a pointer to the _ModuleData structure.

```
template<typename _ModuleData> vdaq_sorted_iterator< _ModuleData >
  create_vdaq_sorted_iterator(VDaqData &M,
                              typename vdaq_sorted_iterator< _ModuleData >::Compare _comp)
```

Returns an iterator of the data of all the modules in the data block. _ModuleData is the structure defining the data definition. The events are ordered by the comparison function. Note that for this to work we need that each module has the data sorted already. Note that it is a forward iterator, it does not implement the increment or decrement operators. Instead it uses the next method.

The iterator has

1. the module_id() method to know the module from which the current data chunk comes.
2. The iterator acts as a pointer to the _ModuleData structure.

The rest of the operations, which usually have to do with data access are responsibility of the VdaqModuleData class.

6.2. The VdaqModuleData class

This class contains the Vdaq generic information about a module. It provides and iterator to go through the data. The iterator, however, is a template since this class knows nothing about the actual data format.

To provide access to the data details the best is to create a class deriving from VdaqModuleData that gives access to the different components of the data. To have it done in an automatic way one should call VdaqData::register_type().

6.2.1. The class methods

This section describes the most used methods of the class. Note that this class knows nothing about the actual data format. This knowledge belongs to the classed that inherit from this and handle the specificities of the module data.

```
int get_module_id() const
```

Returns the module identifier.

```
uint32_t get_nevts() const
```

Returns the number of events in the module data block

```
int get_ntot() const
```

Returns the total number of channels in the module.

```
int get_nchip() const
```

Returns the number of chips in the module.

```
int get_n_config() const
```

Returns the number of configuration words

```
const uint32_t *get_config_data() const
```

Returns a pointer to the array of configuration words. The meaning of those words depends on the different modules.

```
double get_noise(int ichan) const
```

Returns the noise of channel ichan.

```
double get_pedestal(int ichan) const
```

Returns the pedestal of channel ichan.

```
void read_pedestals(const char *fname)
```

Reads pedestals and noise from an ASCII file. The file should have one line per channel and each line contains the pedestal and the noise separated by a white space character.

```
void read_map_file(const char *fname)
```

Reads the channel map from an ASCII file.

```
const PadMap *map() const
```

Returns the PadMap object. See the software documentation for more information about the PadMap. However if your sensor is 1D, there is no need to worry about this.

```
template<typename _ModuleData> ModuleDataIterator<_ModuleData> begin()
template<typename _ModuleData> ModuleDataIterator<_ModuleData> end()
```

Those are the begin and end iterators to go through this particular module data. The iterator acts as a pointer to the _ModuleData structure. The increment operator will read the data from the file.

7. The VATA GP7/HDR data

When using the Vdaq data iterators one will retrieve, for each event, a pointer to a data structure containing the event information of the Vdaq module that handles boards with the GP7 and HDR chips. This data structure is as shown below.

```
struct Data
{
    double   temp;        // Temperature
    double   daq_time;    // Time of arrival of event to DAQ program
    uint32_t time;        // Time stamp from MadDAQ
    uint16_t evtcnt;      // Event counter
    uint16_t nchan;       // Number of channels in the data array
    uint16_t romode;      // readout mode: 1) serial, 2) sparse, 4)  adj.
    union
    {
            uint16_t data[1]; // this is for serial
            struct
            {
                    // Use this for sparse modes
                    uint16_t chan;
                    uint16_t data[1];
                    /*
                     * For sparse+neighbors, the channels come as
                     * chan, chan+1, chan-1, chan+2, chan-2, etc...
                     */
            } sparse;
    };
};
```

The MadDAQModule class provides methods to interpret the configuration data which are listed below.

```
int get_adj_size()
int get_adj_index(int i) const
```

When the system runs in sparse with adjacent channels, this will tell how many neighbours are to be considered and, also, give the index in the list of adjacent channels. The index is 0 for the central channel, ±1 for the right and left neighbours, ±2 for the next outer neighbours, etc.

```
int get_firmware()  const
```

Returns the version of the firmware

```
int get_hold_Delay()  const
```

Returns the value of the hold delay

```
int get_nadj()   const
```

Returns the number of adjacent channels

```
int get_ro_Mode()   const
```

Tells the read out mode used: serial or sparse

```
int get_threshold() const
```

Gives the value of the threshold in DAC units

```
int get_trg_Type()  const
```

Gives the trigger type

There are also a number of methods that will help with the analysis of the data.

void compute_pedestals(int nevts=-1)

Computes the module pedestals from the data. This is particularly usefull when in serial mode. The arguments tells hown many events will be used to make the pedestal calculation. The default is to use all the evetns.

HitList process_event(Data *evt)

Process the data of the given event. It will remove pedestals, compute common mode noise and try to find clusters. It will return a list of hits. The actual data types are as follows:

```
typedef std::pair<int, double> Hit;
typedef std::vector< Hit > HitList;
```