EUDAQ User Manual

EUDAQ Development Team

Last update on March 2017
for EUDAQ version v2.0alpha

This document provides an overview of the EUDAQ software framework, the data acquisition framework used also by the EUDET-type beam telescopes [1]. It describes how to install and run the DAQ system and use many of the included utility programs, and how users may integrate their DAQ systems into the EUDAQ framework by writing their own Producer – for integrating the data stream into the acquisition, DataCollector– for merging the muiltple data streams and writing to disk – and DataConverter – for converting data for offline analysis.

Contents

1. License

2. Introduction

The EUDAQ software is a data acquisition framework, written in C++, and designed to be modular and portable, running on Linux, Mac OS X, and Windows. It was written primarily to run the EUDET-type beam telescope [2, 3], but is designed to be generally useful for other systems.
The hardware-specific parts are kept separate from the core, so that the core library can still be used independently. For example, hardware-specific parts are two components for the EUDET-type beam telescope: the Trigger Logic Unit (TLU) and the National Instrument system (NI) for Mimosa 26 sensor read out.
The raw data files generated by the DAQ can be converted to the Linear Collider I/O (LCIO) format, allowing for analysing the data using the EUTelescope package [4].

2.1. Architecture

It is split into a number of different processes, each communicating using TCP/IP sockets (compare Figure 1). A central Run Control provides an interface for controlling the whole DAQ system; other processes connect to the Run Control to receive commands and to report their status.

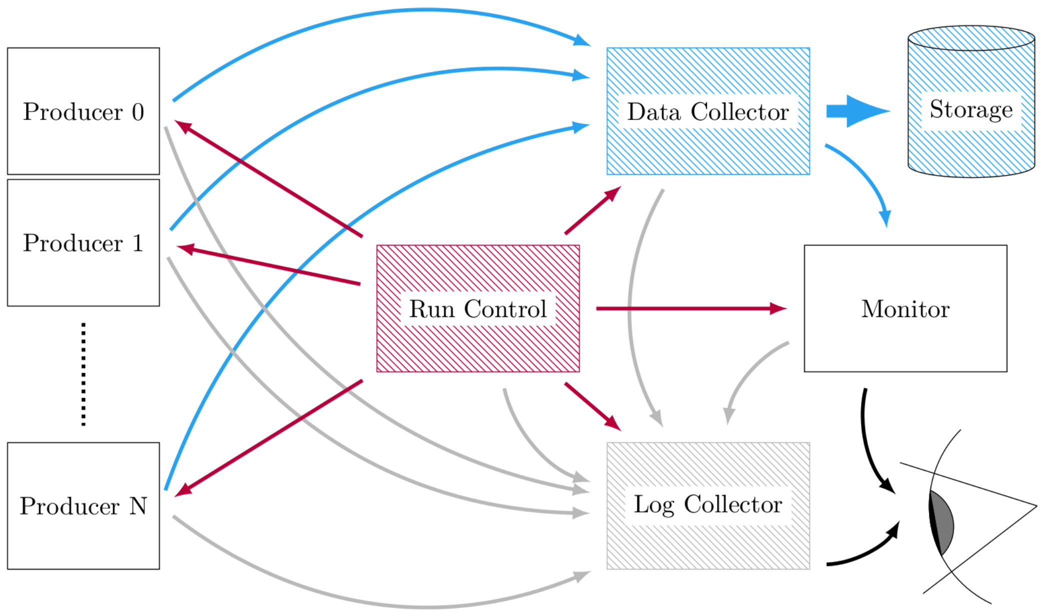


Figure 1: Schematic of the EUDAQ architecture [?].

The DAQ system is made up of a number of different processes that may all be run on the same, or on different computers.

2.1.1. Run Control

TODO

2.1.2. Producer

Each hardware that produces data will have a Producer process (on the left in Figure 1). This will initialize, configure, stop and start the hardware by receiving the commands from the Run Control (red arrows), read out the data and send it to the Data Collector (blue arrows).

2.1.3. Data Collector

The Data Collector is the process that collects all the raw data from the Producers, merges all the connected incoming streams into a single data stream, and writes it to file.

The Data Collector receives all the data streams from all the Producers, and combines them into a single stream that is written to disk (Storage). It writes the data in a native raw binary format, but it can be configured to write in other formats, such as LCIO.

2.1.4. Log Collector

The Log Collector receives log messages from all other processes (grey arrows), and displays them to the user, as well as writing them all to file. This allows for easier debugging, since all log messages are stored together in a central location.

2.1.5. Monitor

The Monitor reads the data file and generates online-monitoring plots for display. In the schematic it is shown to communicate with the Data Collector via a socket, but it actually just reads the data file from disk.

The Online Monitor can be run in one of two modes: online or offline. In online mode, it connects to the Run Control, so it will know when new runs are started, and it will automatically open each new data file as it is created.

2.2. Directory and File Structure

The EUDAQ software is split into several parts that can each be compiled independently, and are kept in separate subdirectories. The general structure is outlined below:

- `main` contains the core EUDAQ library with the parts that are common to most of the software, and several command-line programs that depend only on this library. All definitions in the library should be inside the `eudaq` namespace. It is organized into the following subdirectories:

  - `main/lib/core` contains the source code of core library,
  - `main/lib/lcio` contains the source code of LCIO extension library,
  - `main/lib/root` contains the source code of root extension library,
  - `main/module` contains the source code of module libraries,
  - `main/exe` contains the CLI (command line interface) executables source code,
- `gui` contains the graphical programs that are built with Qt, such as the RunControl and LogCollector.
- `user` contains all user provided code shipped with the EUDAQ distribution, for example:
  - `user/example` contains example code for the integration with user provides code.
  - `user/eudet` contains the parts that depend on the EUDET-type telescope.
  - e.g. `user/calice`, `user/itkstrip`... contain the code from third-party users.
- `extern` global folder which contains external software which themselves are not part of EUDAQ project.
- `cmake` global folder which contains cmake files.
- `etc` contains configuration files of EUDAQ installation.
- `conf` contains configuration files for running the beam telescope.
- `doc` contains source code of documentation, such as this manual.

Each directory containing code has its own `src` and `include` subdirectories, as well as a local `CMakeLists.txt` and a optional `cmake` folder containing the rules for building that directory using `CMake`. Header files have a `.hh` extension so that they can be automatically recognized as C++, and source files have either `.cc` for parts of libraries and `.cxx` for executables. In the case of any external dependencies are required, there could be a local `extern` folder
Each directory can contain a `README.md` file for brief documentation for this specific part, e.g. as installation advice. Using the `*.md` file ending allows for applying the Markdown language [5]. Accordingly, the content will be formatted on the the GitHub platform, where the code is hosted online.

2.3. Executables

All executable programs from the different subdirectories are placed inside the `bin` subdirectory. They should all accept a `-h` (or `--help`) command-line parameter, which will provide a summary of possible different command-line options.
The executable programs mainly split up in two different categories: Processes, which are used for the data acquisition and communicating with the Run Control (DAQ), and utilities, which are used before or after the data taking in order to access the data files (Test, Devel., Tools). In Table 1, you will find an overview of the most important EUDAQ executables.

Binary	**Category**	**GUI/CLI**	**Description**
`euRun`	Run Control	GUI	(Sec. 4.3.1)
`euCliRun`	Run Control	CLI	(Sec. 4.3.1)
`euLog`	Log Collector	GUI	(Sec. 4.3.2)
`euCliLog`	Log Collector	CLI	(Sec. 4.3.2)
`euCliDataCol`	Data Collector	CLI	(Sec. 4.3.3)
`euCliProducer`	Producer	CLI	(Sec. **??**)
`euConverter`	Offline Tool	CLI	raw data file converter (Sec. **??**)

Table 1: Overview of EUDAQ executables.

Binary	**Category**	**Source Code**	**Description**
`libeudaq_core`	Core	main\lib\core	core library
`libeudaq_lcio`	Extension	main\lib\lcio	lcio extension library
`libeudaq_std`	Extension	main\lib\std	standard extension library
`libeudaq_module_test`	Module	main\module\test	test module library
`libeudaq_module_example`	Module	user\example\module	module library by example user

Table 2: Overview of EUDAQ libraries.

2.4. Libraries

The libraries are installed to `lib` sub-folder under install path in Linux/MacOS system, and to `bin` sub-folder in Windows system. The suffix of libraries name can be `.so`, `.dylib`, `.dll` depending on operate systems. There are 3 category of libraries: core, extension and module. In Table 2, you will find an overview of the all EUDAQ libraries.

- Core: the core library. It should be always build and installed.
- Extension: optional features of EUDAQ (eg. support external data format). Static linked core library.
- Module: user module. Dynamic load by EUDAQ core library at run-time.

3. Installation

To install the EUDAQ on Linux, Windows or MacOS, you have a choice to download binary distribution package or build it from the source code.

3.1. Binary Package Installation

TODO

3.2. Building from Source Code

The installation is described in four steps:[1]

1. Installation of (required) prerequisites
2. Downloading the source code (GitHub)
3. Configuration of the code (CMake)
4. Compilation of the code

If you occur problems during the installation process, please have a look into the issue tracker on GitHub.[2] Here you can search, if your problem had already been experienced by someone else, or you can open a new issue (see **??**).

3.2.1. Prerequisites

EUDAQ has few dependencies on other software, but some features do rely on other packages:

- To get the code and stay updated with the central repository on GitHub git is used.
- To configure the EUDAQ build process, the CMake cross-platform, open-source build system is used.
- To compile EUDAQ from source code requires a compiler that implements the C++11 standard.
- Qt is required to build GUI of the e.g. Run Control Log Collector.
- ROOT is required for the Online Monitor.

[1]Quick installation instructions are also described on `http://eudaq.github.io/` or in the main `README.md` file of each branch, e.g. `https://github.com/eudaq/eudaq/blob/v1.6-dev/README.md`.
[2]Go to `https://github.com/eudaq/eudaq/issues`

Git Git is a free and open source distributed version control and is available for all of the usual platforms [6]. It allows for local version control and repositories, but also communicating with central online repositories like GitHub.
In order to get the EUDAQ code and stay updated with the central repository on GitHub git is used (see subsubsection 3.2.2). But also for developing the EUDAQ code having different versions (tags) or branches (development repositories), git is used (see ??).

CMake (required) In order to generate configuration files for building EUDAQ (makefiles) independently from the compiler and the operating platform, the CMake (at least version 3.1) build system is used.
CMake is available for all major operating systems from `http://www.cmake.org/cmake/resources/software.html`. On most Linux distributions, it can usually be installed via the built-in package manager (aptitude/apt-get/yum etc.) and on MacOS using packages provided by e.g. the MacPorts or Fink projects.

C++11 compliant compiler (required) The compilation of the EUDAQ source code requires a C++11 compliant compiler and has been tested with GCC (at least version 4.8.1), Clang (at least version 3.1), and MSVC (Visual Studio 2013 and later) on Linux, OS X and Windows.

Qt (for GUI) The graphical interface of EUDAQ uses the Qt graphical framework. In order to compile the `gui` subdirectory, you must therefore have Qt installed. It is available in most Linux distributions as the package `qt5-devel` or `qt5-dev`. It can also be downloaded and installed from `http://download.qt.io/archive/qt/`.

ROOT (for ROOT extension and OnlineMonitor) To enable the EUDAQ extensions library support to ROOT, EUDAQ need to be compiled and linked to ROOT library. Online Monitor also use the ROOT package. It can be downloaded from `http://root.cern.ch`.
Make sure ROOt's `bin` subdirectory is in your path, so that the `root-config` utility can be run. This can be done by sourcing the `thisroot.sh` (or `thisroot.ch` for csh-like shells) script in the `bin` directory of the ROOT installation:

```
source /path-to/root/bin/thisroot.sh
```

LCIO (for LCIO extension) To enable the writing of LCIO files, or the conversion of native files to LCIO format, EUDAQ must be linked against the LCIO libraries. When LCIO option is enabled during the configuration step, source files will be download from internet and compiled automatically.

3.2.2. Obtain the source code

The EUDAQ source code is hosted on GitHub [7]. Here, we describe how to get the code and install a stable version release. In order to get information about the work flow of developing the EUDAQ code, please find the relevant information in see ??.

Downloading the code (clone) We recommend to obtain the software by using git, since this will allow you to easily update to newer versions. The source code can be downloaded with the following command:

```
git clone https://github.com/eudaq/eudaq.git eudaq
```

This will create the directory `eudaq`, and download the latest version into it.
Note: Alternatively and without version control, you can also download a zip/tar.gz file of EUDAQ releases (tags) from https://github.com/eudaq/eudaq/releases. By downloading the code, you can skip the next two subsections.

Changing to a release version (checkout) After cloning the code from GitHub, your local EUDAQ version is on the master branch (check with `git status`). For using EUDAQ without development or for production environments (e.g. at test beams), we strongly recommend to use the latest release version. Use

```
git tag
```

in the repository to find the newest stable version as the last entry. In order to change to this version in your local repository, execute e.g.

```
git checkout v2.0.0alpha
```

to change to version v2.0.0alpha.

Updating the code (fetch) If you want to update your local code, e.g to get the newest release versions, execute in the `eudaq` directory:

```
git fetch
```

and check for new versions with `git tag`.

3.2.3. Configuration via CMake

CMake supports out-of-source configurations and generates building files for compilation (makefiles). Enter the `build` directory and run CMake, i.e.

```
cd build
cmake ..
```

CMake automatically searches for required packages and verifies that all dependencies are met using the `CMakeLists.txt` scripts in the main folder and in all sub directories. You can modify this default behavior by passing the `-D[eudaq_build_option]` option to CMake where `[name]` refers to an optional component, e.g.

```
cmake -DEUDAQ_BUILD_GUI=ON  ..
```

to disable the GUI but enable additionally executable of the TLU producer. Find some of the most important building options in Table 3.
If you are not familiar with cmake, cmake-gui is nice GUI tool to help you configure the project.

option	default	dependency	comment
`EUDAQ_BUILD_EXECUTABLE`	`<auto>`	none	Builds main EUDAQ executables.
`EUDAQ_BUILD_GUI`	`<auto>`	Qt5	Builds GUI executables, such as the Run Control(euRun) and Log Collector(euLog).
`EUDAQ_BUILD_MANUAL`	`OFF`	LaTex	Builds Manual in pdf-format.
`EUDAQ_BUILD_NREADER`	`OFF`	EUTelescope, LCIO	Builds native reader Marlin processor used for data conversion into LCIO.
`EUDAQ_BUILD_ONLINE_MONITOR`	`auto`	ROOT6	Builds online monitor.
`EUDAQ_INSTALL_PREFIX`	`<source_folder>`	none	In order to install the executables into `bin` and the library into `lib` of a specific path, instead of into the `<source_folder>` path.
`EUDAQ_LIBRARY_BUILD_CLI`	`ON`	none	Builds extension library of command line interface
`EUDAQ_LIBRARY_BUILD_LCIO`	`<auto>`	LCIO	Builds LCIO extension library
`EUDAQ_LIBRARY_BUILD_ROOT`	`<auto>`	ROOT	Builds ROOT extension library
`EUDAQ_LIBRARY_BUILD_TEST`	`ON`	none	Builds extension library for develop test
`USER_EXAMPLE_BUILD`	`ON`	none	Builds example user code
`USER_{user_name}_BUILD`	`<unknown>`	`<unknown>`	Builds user code inside user folder {user_name}
`CMAKE_BUILD_TYPE`	`RelWithDebInfo`	none	Only affect the building on Linux/-MacOS, see CMake Manual

Table 3: Options for CMake.

Note: After generating building files by running `cmake ..`, you can list all possible option and their status by running `cmake -L`. Using a GUI version of CMake shows also all of the possible options.
Corresponding settings are cached, thus they will be used again next time CMake is running. If you encounter a problem during installation, it is recommended to clean the cache by just removing all files from the build folder, since it only contains automatically generated files.

3.2.4. Compilation

Universal command for all systems:

```
cmake --build {source_folder}/build --target install
```

On Windows/Visual Studio, CMake option CMAKE_BUILD_TYPE does not really affect the build type. It is possible to change the default build type Debug to Release by the build time option “–config Release”.

```
cmake --build {source_folder}/build --target install --config Release
```

Done!

4. Run an Example Setup

This section will describe running the DAQ system, mainly from the point of view of running an example setup as a demonstration without dedicated hardware. However, this description can be applied to DAQ system in general.

4.1. Target Setup

In this example, user hardware device is simulated and implemented as an example Producer which can be configured to generate fake data according the configuration. This works similarly to a real producer, but does not talk to any real hardware. Example DataCollector is also implemented.
The runtime setup is consisted of the following processes of EUDAQ.

`RunControl:` Standard RunContorl GUI

`LogCollector:` Standard LogCollector GUI

`Producer:` 2 Producers instanced by CLI command. One of them produce event at rate 1Hz, the other producer event at rate of 2Hz.

`DataCollector:` 2 DataCollectors instanced by CLI command. One of them receives event only from the producer at 1Hz rate, the other receives events from both DataCollectors, and sync the 2 stream by Timestamp.

`Monitor:` No monitor exists. (TODO: producer generate hit data)

4.2. Preparation

Some preparation is needed to make sure the environment is set up correctly and the necessary TCP ports are not blocked before the DAQ can run properly.

4.2.1. Directories

If no specified path is passed to EUDAQ (by configuration file or command line parameter), EUDAQ will assume the working folder where executable is started up is writable. Data and log files will be stored to working folder.

4.2.2. Init/Config-Files

`*.ini`-files for initialization and `*.conf`-files for configuration are text files in a specific format, containing name-value pairs separated into different sections. Any text from a `#` character until the end of the line is treated as a comment, and ignored. Each section in the config file is delimited by a name in square brackets (e.g. `[RunControl]`). The name represents the type of process to which it applies; if there are several such processes, then they can be differentiated by including the name after a period (e.g.

[Producer.Example]). Within each section, any number of parameters may be specified, in the form Name = Value. It is then up to the individual processes how these parameters are interpreted. https://en.wikipedia.org/wiki/INI_file
During the initialization and configuration, each process gets its section and do not know the other parts of ini/conf files.

```
[Producer.p1]
DUMMY_STRING = "abcdefghijklmnopqresqvwxyz"
DUMMY_FILE_PATH = "myp1.txt"

[Producer.p2]
DUMMY_STRING = "0123456789"
DUMMY_FILE_PATH = "myp2.txt"
```

```
[Producer.p1]
EUDAQ_ID = 11
EUDAQ_DC = "d1,d2"
DURATION_BUSY_MS = 1000
ENABLE_TIEMSTAMP = 0
ENABLE_TRIGERNUMBER = 1

[Producer.p2]
EUDAQ_ID = 12
EUDAQ_DC = "d2"
DURATION_BUSY_MS = 1000
ENABLE_TIEMSTAMP = 0
ENABLE_TRIGERNUMBER = 1

[DataCollector.d2]
DISABLE_PRINT = 0
```

4.2.3. Ports and Firewall

The different processes communicate between themselves using TCP/IP sockets. The ports can be configured when calling the the processors on the command line (see below). By default, TCP port 44000 is listened by RunContorl, and TCP port 44002 is listened by Log Collector to get connections from clients. Data Collector itself will pick a random TCP port to listen and get incoming data from its connected Producers if there is no specific port signed explicitly by user.
When all EUDAQ processes run at a same computer, common firewalls will not affect the TCP connections among them. However, running EUDAQ distributively on several computers may have issue from firewall blocking. You will have to open up those TCP port for incoming connection or temporally shut down the firewall. The instruction about firewall is operate system depended.

In this example, all processes will run at same Linux computer, we can skip the setup of TCP port.

4.3. Startup

To start EUDAQ, all of the necessary processes have to be started in the correct order. The first process must be the Run Control (euRun), since all other processes will attempt to connect to it when they start up. Then it is recommended to start the Log Collector, since any log messages it receives may be useful to help with debugging in case everything does not start as expected. Finally, the Data Collector, Producers and Monitor can be started.

4.3.1. RunControl

Two versions of the RunControl – a text-based version `euCliRun` and a graphical version `euRun` (see Figure 2).
The command line pattern to start up a Log Collector is:

```
euRun -a tcp://{listening_port}
```

`-a` ⟨`listening_addr`⟩: optional, `listening_port` default value is 44000.

For this example setup, we will startup the GUI version of Run Control and make it serve at TCP port 44000

```
euRun -a tcp://44000
```

After run the above command, a new GUI windows is showing Figure 2.

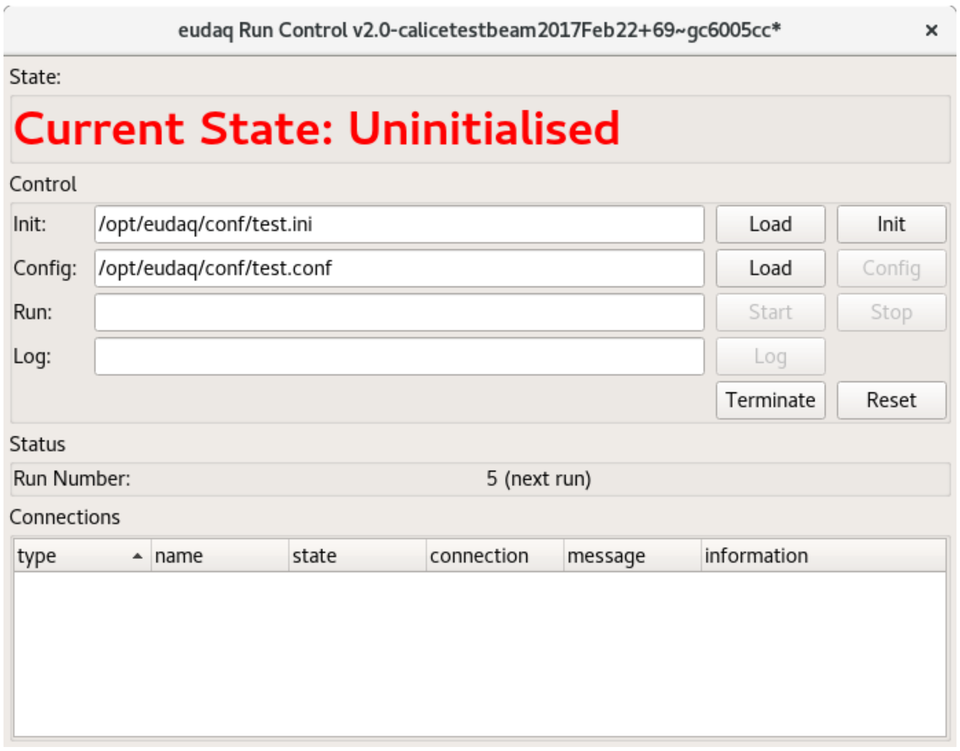
Initialization Section none

Configuration Section

```
[RunControl]
EventSizeLimit=10000
```

4.3.2. LogCollector

It is recommended to start the Log Collector directly after having started the Run Control and before starting other processors in order to collect all log messages generated by all other processes.
There are also two versions of the Log Collector. The graphical version is called `euLog`, and the text-based version is called `euCliLog`.
The command line pattern to startup a Log Collector is:


eudaq Run Control v2.0-calicetestbeam2017Feb22+69~gc6005cc*

State:

Current State: Uninitialised

Control

Init: /opt/eudaq/conf/test.ini

Config: /opt/eudaq/conf/test.conf

Run:

Log:

Status

Run Number: 5 (next run)

Connections

type	name	state	connection	message	information

Figure 2: The Run Control graphical user interface.

```
euLog -r tcp://{run_contorl_hostname}:{run_contorl_port} -a ↩
    tcp://{listening_port}
```

`-r` ⟨`runcontrol_addr`⟩ **:** optional, `run_control_hostname` default value: localhost; `run_contorl_port` default value: 44000.

`-a` ⟨`listening_addr`⟩ **:** optional, `listening_port` default value is 44002.

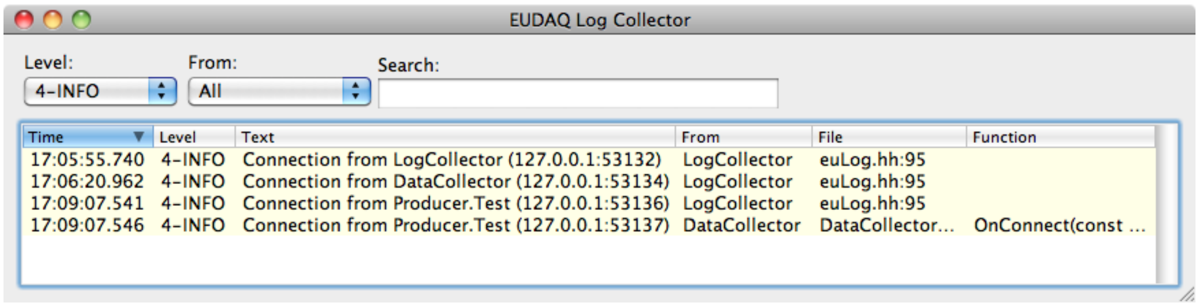
For this example setup, we will startup the GUI version of Log Collector, connect it to the Run Contorl at local port 44000 and make it serve at TCP port 44002

```
euLog -r tcp://localhost:44000 -a tcp://44002
```

After run the above command, a new GUI windows is showing Figure 3, and Run Contorl is displaying a new connection Figure 4

4.3.3. DataCollector

There is only a text-based version called `euCliDataCollector`. The command line pattern to startup a DataCollector is:



```
Time          Level   Text                                          From           File            Function
17:05:55.740  4-INFO  Connection from LogCollector (127.0.0.1:53132)  LogCollector   euLog.hh:95
17:06:20.962  4-INFO  Connection from DataCollector (127.0.0.1:53134) LogCollector   euLog.hh:95
17:09:07.541  4-INFO  Connection from Producer.Test (127.0.0.1:53136) LogCollector   euLog.hh:95
17:09:07.546  4-INFO  Connection from Producer.Test (127.0.0.1:53137) DataCollector  DataCollector...  OnConnect(const ...
```

Figure 3: The Log Collector graphical user interface.

```
euCliDataCollector -n {code_name} -t {runtime_name} -r ↩
    tcp://{run_control_hostname}:{run_contorl_port} -a tcp://{listening_port}
```

-n ⟨code_name⟩ : required.

-t ⟨runtime_name⟩ : required.

-r ⟨runcontrol_addr⟩ : optional, run_control_hostname default value: localhost; run_contorl_port default value: 44000.

-a ⟨listening_addr⟩ : optional, listening_port default value is random.

By default, example DataCollector Ex0DataCollector is available with standard installation of EUDAQ. For this example setup, we will startup 2 instances of Ex0DataCollector with runtime names my_dc and another_dc

```
euCliDataCollector -n Ex0DataCollector -t my_dc -r tcp://localhost:44000 -a ↩
    tcp://45001
euCliDataCollector -n Ex0DataCollector -t another_dc -r ↩
    tcp://localhost:44000 -a tcp://45002
```

Initialization Section

```
[DataCollector.my_dc]
KEY1=a

[DataCollector.another_dc]
KEY1=b
```

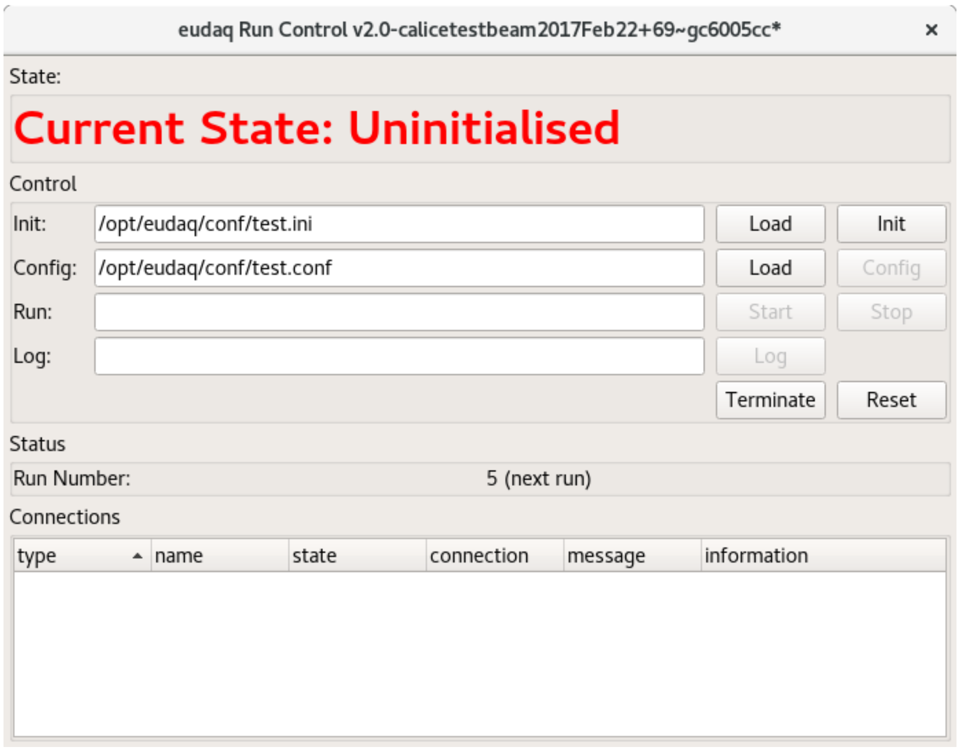

eudaq Run Control v2.0-calicetestbeam2017Feb22+69~gc6005cc*
State:
Current State: Uninitialised
Control
Init: /opt/eudaq/conf/test.ini
Load
Init
Config: /opt/eudaq/conf/test.conf
Load
Config
Run:
Start
Stop
Log:
Log
Terminate
Reset
Status
Run Number: 5 (next run)
Connections
type
name
state
connection
message
information

Figure 4: The Run Control with connection from Log Collector.

Configuration Section Each Data Collector has to write to a different file by including the `FilePattern` option in the corresponding section of your configuration file (also see section 4.2.2):

```
[DataCollector.my_dc]
FilePattern=run$6R$X

[DataCollector.another_dc]
FilePattern=dc2_run$6R$X
```

4.3.4. Producer

There is only a text-based version called `euCliProducer`. The command line pattern to startup a Producer is:

```
euCliProducer -n {code_name} -t {runtime_name} -r ↩
    tcp://{run_control_hostname}:{run_contorl_port}
```

`-n` ⟨`code_name`⟩ `:` required.

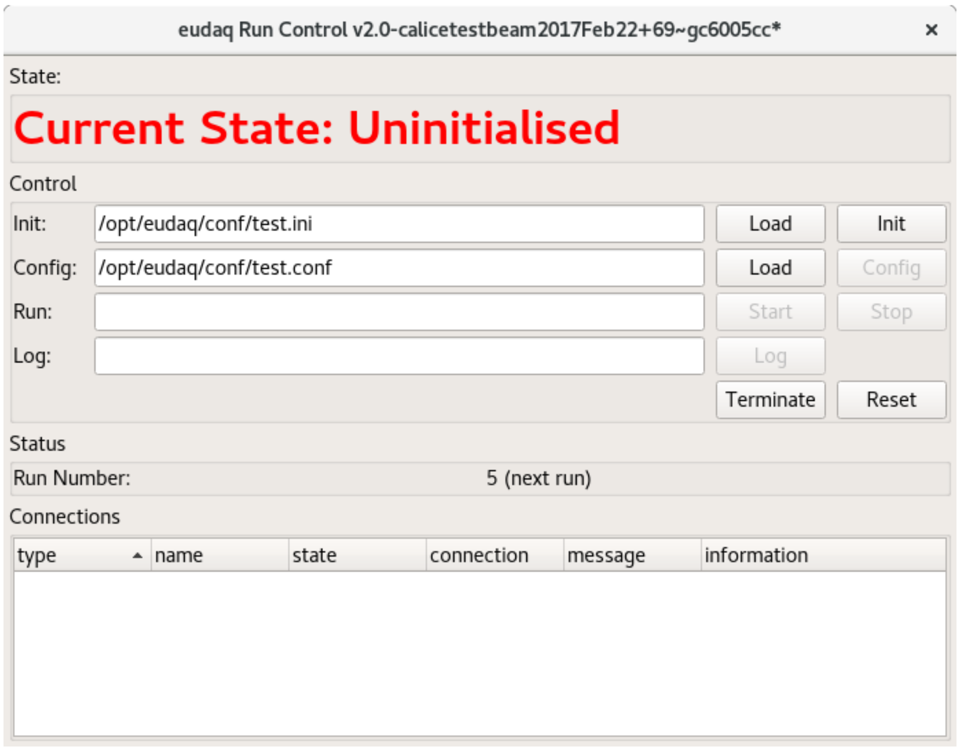
`-t` ⟨`runtime_name`⟩ `:` required.

-r ⟨runcontrol_addr⟩: optional, run_control_hostname default value: localhost; run_contorl_port default value: 44000.

By default, example Producer Ex0Producer is available with standard installation of EUDAQ. For this example setup, we will startup 2 instances of Ex0Producer with runtime names prd1hz and prd2hz

```
euCliProducer -n Ex0Producer -t prd1hz -r tcp://localhost:44000
euCliProducer -n Ex0Producer -t prd2hz -r tcp://localhost:44000
```

After run all the above commands, Run Contorl GUI window will be like Figure 5


eudaq Run Control v2.0-calicetestbeam2017Feb22+69~gc6005cc*

State:

Current State: Uninitialised

Control

Init: /opt/eudaq/conf/test.ini

Config: /opt/eudaq/conf/test.conf

Run:

Log:

Status

Run Number: 5 (next run)

Connections

type	name	state	connection	message	information

Figure 5: The Run Control with connections of 2 Producers.

Initialization Section

```
[Producer.prd1hz]
EUDAQ_DC=my_dc

[Producer.prd2hz]
EUDAQ_DC=my_dc,another_dc
```

Configuration Section

```
[Producer.prd1hz]
Rate=1
Random=true

[Producer.prd2hz]
Rate=2
Random=false
```

4.3.5. Monitor

There is a RootMonitor which can run in both online and offline mod. To run it in online mode, the command line pattern is:

```
OnlineMon -r tcp://{run_control_hostname}:{run_contorl_port} -a ↩
   tcp://{listenning_port}
```

`-r` ⟨`runcontrol_addr`⟩ **:** optional, `run_control_hostname` default value: localhost; `run_contorl_port` default value: 44000.

`-a` ⟨`listening_addr`⟩ **:** optional, `listening_port` default value is random.

In offline mode, there is no Run Control, To run it in online mode, the command line pattern is:

```
OnlineMon -f {path_to_data_file}
```

`-f` ⟨`path_to_data_file`⟩ **:** required

4.4. Operating

Once all the processes have been started, and by using the Run Control (see Figure 2), According to the machine state subsubsection 5.4.2, EUDAQ and all processes can be initialized, configured or re-configured, data taking (runs) can be started and stopped, and the software can be terminated.

- First, the appropriate initialization file should be selected (see subsubsection 4.2.2 for creating and editing init-files). Then the `Init` button can be pressed, which will send a initialization command to all connected processes.

- Second, the appropriate configuration should be selected (see subsubsection 4.2.2 for creating and editing configurations), and the `GeoID` should be verified (see **??**), before continuing. Then the `Config` button can be pressed, which will send a configuration command (with the contents of the selected configuration file) to all

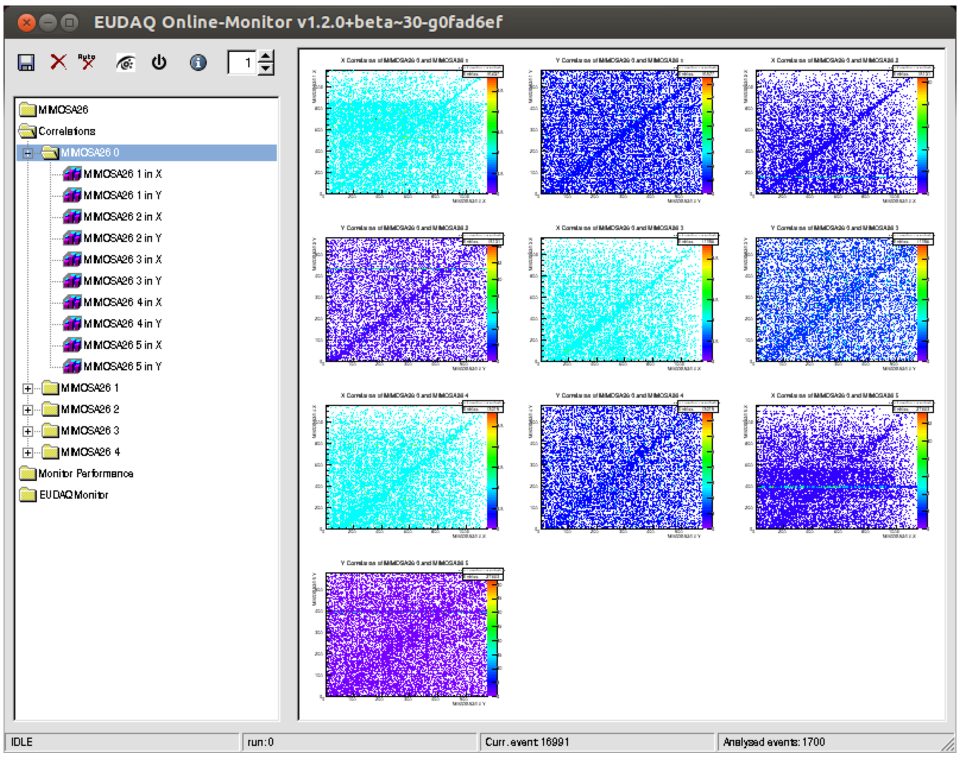


Figure 6: The OnlineMon showing correlation plots between different Mimosa26 planes of the EUDET telescope.

connected processes. The full contents of the configuration file will also be stored in the beginning-of-run-event (BORE) of the data file, so that this information is always available along with the data.

- Once all connected processes are fully configured, a run may be started, by pressing the `Start` button. Whatever text is in the corresponding text box (”`Run:`”) when the button is pressed will be stored as a comment in the data file. This can be used to help identify the different runs later.
- Once a run is completed, it may be stopped by pressing the `Stop` button. Runs will also stop and restart automatically when the data file reaches a threshold in size (by default this is 1 GB).[3] The threshold size for restarting a run may be configured in the config file (see subsubsection 4.2.2).
- At any time, a message may be sent to the log file by filling in the (”`Log:`”) text box and pressing the corresponding button. The text should appear in the LogCollector window, and will be stored in the log file for later access.

[3]This is because there is a file size limit of 2 GB for storage on the GRID, and the processed files can grow bigger than the original native files.

- Once the run is stopped, the system may be reconfigured with a different configuration, or another run may be started; or EUDAQ can be terminated.

Class	Description
`eudaq::Producer`	Sec. 6
`eudaq::DataCollector`	Sec. 7
`eudaq::RunControl`	Sec. 10
`eudaq::Event`	
`eudaq::LogCollector`	
`eudaq::Monitor`	Sec. 9
`eudaq::FileWriter`	Sec. 11.2
`eudaq::FileReader`	Sec. 11.3
`eudaq::StdEventConverter`	
`eudaq::LCEventConverter`	
`eudaq::TransportServer`	internal only
`eudaq::TransportClient`	internal only

Table 4: Derivable Classes.

5. Integration with User Hardware

EUDAQ itself is only a data taking framework which provides the common feature. It means that the users with their dedicated hardware and readout software are required to write some code to bridge the hardware specific readout software to EUDAQ frame work. The minimum adaption task is to have write a Producer for each piece of hardware, a Data Collector to receive the data (aka. Event) from Producers.

5.1. Announcement of Derived Class

The derived EUDAQ classes provided user will compiled and packed to a dynamic shared library (EUDAQ Module Library). At compiling/linking time of EUDAQ core library, it does not know the existing of any Module Library. When EUDAQ core library is been loading by any application, the core library will looking any library file with name prefix libeudaq_module_ in the module folder. All pattern matched library are going to be loaded. It is the time point when each derived EUDAQ class announces itself to the EUDAQ runtime environment.
Technically, the announcement of a derived class is done by a calling to the correlated static function provided by a generic C++ template (eudaq::Factory). Tab. Table 4 is the list of derivable EUDAQ classes

5.2. Serializable

As a distributed DAQ framework, a runtime setup of EUDAQ system include several applications. Data objects will go through the boundary of an application or a computer. Those data objects should have the capability to be serialized. When a data object is serialized, all the crucial data of this data object is feed to serialized memory which then can be send by plain binary to another application and reconstructed to a copy of

Class	Description
eudaq::Event	
eudaq::Configuration	
eudaq::LogMessage	
eudaq::Status	

Table 5: Serializable Classes.

variable	C++ type	Description
m_type	uint32_t	event type
m_version	uint32_t	version
m_flags	uint32_t	flags
m_stm_n	uint32_t	device/stream number
m_run_n	uint32_t	run number
m_ev_n	uint32_t	event number
m_tg_n	uint32_t	trigger number
m_extend	uint32_t	reserved word
m_ts_begin	uint64_t	timestamp at the begin of event
m_ts_end	uint64_t	timestamp at the end of event
m_dspt	std::string	description
m_tags	std::map<std::string, std::string>	tags
m_blocks	std::map<uint32_t, std::vector<uint8_t>>	blocks of raw data
m_sub_events	std::vector<EventSPC>	pointers of sub events

Table 6: Variables of eudaq::Event.

original data object.

All the classes which hold serializable data are derived from a base serializable class (eudaq::Serializable). All serializable data class should implement the function `Serialize` which serializes the inner data object and feeds a eudaq::Serializer, and a constructor function which takes the reference of eudaq::Deserializer as input parameter.

Table 5 is the list of serializable EUDAQ classes.

5.2.1. Event

eudaq::Event is most important serializable class which holds physics data from hardware. Producer is the EUDAQ component which create object of eudaq::Event and feed it by the physics data from measurement. Table 5 lists the variables inside the eudaq::Event.

The `m_blocks` is physics data which only can be known by the user who owns the hardware. There is a pair of timestamps to define the time slice when the physics event occurs, and a trigger number to identify the trigger sequence. Timestamps and trigger number are optional to be set if you are going to use them to synchronize data from

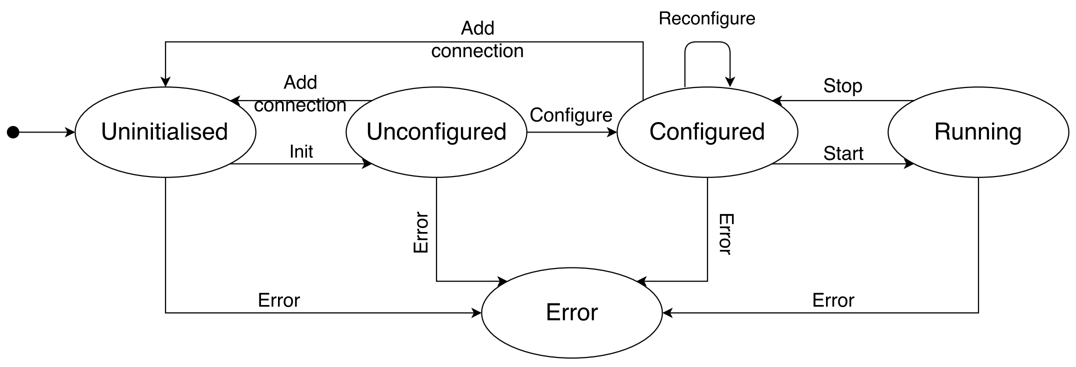


Figure 7: The FSM of EUDAQ.

multiple stream/device. It is also possible to have sub events inside an eudaq::Event object. The sub eudaq::Event objects is hold by std::shared_ptr. See next sub section.

5.3. Ownership

std::shared_ptr and std::shared_ptr is heavily used in EUDAQ to hold the object pointer of serializable class and derivable class. It gets rid of the unnecessary and ineffective memory copy and is exception safe for any memory leaking.

5.4. Command/Status Handling

5.4.1. RunControl and CommandReceiver

RunControl eudaq::RunControl is the command sender which issue command according to user actions from GUI or CLI.

CommandReceiver eudaq::Producer, eudaq::DataCollector, eudaq::LogCollector and eudaq::Monitor are command receivers (eudaq::CommandReceiver) which executes the correlated function according the command. The command receiver will set up a status (eudaq::Status) and report the status to RunControl.

5.4.2. State Model

The finite-state machine finite-state machine (FSM) is implemented to both RunControl and CommandReceiver (see Figure 7) [8].

Each command receiver can always be characterized by the current state (Table 7)
The state of RunControl is determined by the lowest state of the connected client in the following priority: ERROR, UNINITIALISED, UNCONFIGURED, CONFIGURED, RUNNING. It means, for example, that even if only one connection is in the ERROR

State	Enumerate Value	Acceptable Command	Description
`Error`	`eudaq::Status::STATE_ERROR`	DoReset	
`Uninitialised`	`eudaq::Status::STATE_UNINIT`	DoInitialise DoReset	the initial state c
`Unconfigured`	`eudaq::Status::STATE_UNCONF`	DoConfigure DoReset	
`Configured`	`eudaq::Status::STATE_CONF`	DoConfigure DoStartRun DoReset	
`Running`	`eudaq::Status::STATE_RUNNING`	DoStopRun	

Table 7: States of RunControl client

Command	RunControl Side	Client Side	Pre State	State of Success
`Initialise`	Initialise	DoInitialise	Uninitialised	Unconfigured
`Configure`	Configure	DoConfigure	Unconfigured Configured	Configured
`StartRun`	StartRun	DoStartRun	Configured	Running
`StopRun`	StopRun	DoStopRun	Running	Configured
`Reset`	Reset	DoReset	Error	Uninitialised

Table 8: List of Commands

state, the whole machine will also be in that state. This prevents such mistakes as running the system before every component has finished the configuration.

5.4.3. Command

Initialise When an EUDAQ process is on the state of Uninitialised, it accepts Initialise command and execute the DoInitialise function provided by user. When it is initialized, it does not accept further Uninitialised command any more until it is reseted to Uninitialised state.

Configure When an EUDAQ process is on the state of Unconfigured or Configured, it accepts Configure command and execute the DoConfigure function provided by user. Please be aware that the second Configure command after a successful execution of Configure command is allowed. It means EUDAQ process is re-configurable but not re-initialisible.

StartRun When an EUDAQ process is on the state of Configured, it also accepts StartRun command and execute the DoStartRun function provided by user. In case of Producer, the Producer should talk to the underline hardware and produce and send Event data. In case of DataCollector, the DataCollector will get the Event stream from its connected Producer.

StopRun When an EUDAQ process is on the state of Running, it only accepts StopRun command and execute the DoStopRun function provided by user. RunControl will not send the StopRun command to DataCollector until all Producers have returned from DoStopRun function and feedback their status.

Reset In any case when an EUDAQ process goes to the state of Error, only Reset command is allowed.

Terminate Except for the state of Running, the terminate command is allowed on all other state. When the Terminate command is called, the EUDAQ process are going to be closed. If user has to do something before the exiting of EUDAQ process, DoTerminate function is correct position to do the exiting route.

6. Writing a Producer

Producers are the binding part between a user DAQ and the central EUDAQ Run Control. The base class of Producer is eudaq::Producer. The eudaq::Producer is an inherited class of eudaq::CommandReceiver which receives commands from the eudaq::RunControl. It also maintains a set of eudaq::DataSender which allows to send binary data (eudaq::Event) to each destinations.

6.1. Producer Prototype

Listing 6.1 is part of the header file which declares the eudaq::Producer. You are required to write the user Producer derived from eudaq::Producer. According the the set of commands (subsubsection 5.4.3) in EUDAQ framework, there are 5 virtual functions should be implemented in user producer. They are DoInitialise(), DoConfigure(), DoStopRun(), DoStartRun(), DoReset() and DoTerminate(). These command related function should return as soon as possible since no further command can be execuseted until the finish the recent command.

```
class DLLEXPORT Producer : public CommandReceiver{
public:
  Producer(const std::string &name, const std::string &runcontrol);
  void OnInitialise() override final;
  void OnConfigure() override final;
  void OnStartRun() override final;
  void OnStopRun() override final;
  void OnReset() override final;
  void OnTerminate() override final;
  void Exec() override;

  virtual void DoInitialise(){};
  virtual void DoConfigure(){};
  virtual void DoStartRun() = 0;
  virtual void DoStopRun() = 0;
  virtual void DoReset() = 0;
  virtual void DoTerminate() = 0;

  void SendEvent(EventUP ev);
private:
  uint32_t m_pdc_n;
  uint32_t m_evt_c;
  std::map<std::string, std::unique_ptr<DataSender>> m_senders;
};
```

The virtual function Exec() is optional to be implemented in user producer. Internally, the base Exec() to create a thread for command execution and itself goes to a unlimited loop and can never return until the Terminate command. In case you are going to implement it yourselves, please read the source code to find detail information.

6.2. Example Source Code: Dummy Event Producer

6.2.1. Declaration

Let's go to a example implement ion of user Producer. The full source code is available at file path user/example/module/src/Ex0Producer.cc. Listing 6.2.2 is the declaration part. There are 6 command related functions, a constructor function and a `Mainloop` function.

```
class Ex0Producer : public eudaq::Producer {
  public:
  Ex0Producer(const std::string & name, const std::string & runcontrol);

  void DoInitialise() override;
  void DoConfigure() override;
  void DoStartRun() override;
  void DoStopRun() override;
  void DoTerminate() override;
  void DoReset() override;
  void Mainloop();

  static const uint32_t m_id_factory = eudaq::cstr2hash("Ex0Producer");
private:
  bool m_flag_ts;
  bool m_flag_tg;
  std::ifstream m_ifile;
  std::string m_dummy_data_path;
  std::chrono::milliseconds m_ms_busy;

  std::thread m_thd_run;
  bool m_exit_of_run;
};
```

In the line `static const uint32_t m_id_factory = eudaq::cstr2hash("Ex0Producer")`, it defines a static number which is a hash from name string. This number will be used later as a ID to register this `Ex0Producer` to EUDAQ runtime env.

6.2.2. Register to Factory

To make EUDAQ framework know the existing of `Ex0Producer`, this Producer should registered to `eudaq::Factory<eudaq::Producer>` with it static ID number. The input parameter types of constructor function is also provided to Register function. See the example code below.

```
namespace{
  auto dummy0 = eudaq::Factory<eudaq::Producer>::
    Register<Ex0Producer, const std::string&, const ↩
        std::string&>(Ex0Producer::m_id_factory);
}
```

6.2.3. Constructor

The Constructor function takes in 2 parameters, the runtime name of the instance of Producer and the address of RunControl.

```
Ex0Producer::Ex0Producer(const std::string & name, const std::string & ↩
    runcontrol)
  :eudaq::Producer(name, runcontrol), m_exit_of_run(false){
}
```

In this example of Ex0Producer, constructor function does nothing beside passing parameters to base constructor function of Producer and initialize the `m_exit_of_run` variable.

6.2.4. DoInitialise

This method is called whenever an initialize command is received from the Run Control. When This function is called, the correlated section of initialization file have arrived to the Producer from RunControl. This initialization section can be obtained by
`eudaq::ConfigurationSPC eudaq::CommandReceiver::GetInitConfiguration()`

Here is example `DoInitialise` of Ex0Producer.

```
void Ex0Producer::DoInitialise(){
  auto ini = GetInitConfiguration();
  std::ofstream ofile;
  std::string dummy_string;
  dummy_string = ini->Get("DUMMY_STRING", dummy_string);
  m_dummy_data_path = ini->Get("DUMMY_FILE_PATH", "ex0dummy.txt");
  ofile.open(m_dummy_data_path);
  if(!ofile.is_open()){
    EUDAQ_THROW("dummy data file (" + m_dummy_data_path +") can not open ↩
        for writing");
  }
  ofile << dummy_string;
  ofile.close();
}
```

The Configuration object `ini` is obtained. According to the value DMMMY_STRING and UMMY_FILE_PATH in `ini` object, a new file is opened and filled by dummy data. The path of file is saved as variable member for later access of this dummy data file.

6.2.5. DoConfigure

This method is called whenever an configure command is received from the Run Control. When This function is called, the correlated section of configuration file have arrived to the Producer from RunControl. The configuration section named by Producer runtime name can be obtained by
`eudaq::ConfigurationSPC eudaq::CommandReceiver::GetConfiguration()`

Here is example `DoConfigure` of Ex0Producer.

```
void Ex0Producer::DoConfigure(){
  auto conf = GetConfiguration();
  conf->Print(std::cout);
  m_ms_busy = std::chrono::milliseconds(conf->Get("DURATION_BUSY_MS", 1000));
  m_flag_ts = conf->Get("ENABLE_TIEMSTAMP", 0);
  m_flag_tg = conf->Get("ENABLE_TRIGERNUMBER", 0);
  if(!m_flag_ts && m_flag_tg){
    EUDAQ_WARN("Both Timestamp and TriggerNumber are disabled. Now, ↩
        Timestamp is enabled by default");
    m_flag_ts = false;
    m_flag_tg = true;
  }
}
```

The variable `m_ms_busy`, `m_flag_ts` and `m_flag_tg` are set according the Configuration object.

6.2.6. DoStartRun

This method is called whenever an StartRun command is received from the Run Control. When This function is called, the run-number have already been increased by 1. For real hardware specific Producer, the hardware is told to startup. Here is example `DoStartRun` of Ex0Producer.

```
void Ex0Producer::DoStartRun(){
  m_exit_of_run = false;

  m_ifile.open(m_dummy_data_path);
  if(!m_ifile.is_open()){
    EUDAQ_THROW("dummy data file (" + m_dummy_data_path +") can not open ↩
        for reading");
  }
  m_thd_run = std::thread(&Ex0Producer::Mainloop, this);
}
```

Instead of talking to real hardware, a file is opened. Then, a new thread is started using the function `Mainloop`.

6.2.7. DoStopRun

This method is called whenever an StopRun command is received from the Run Control. Here is example `DoStopRun` of Ex0Producer.

```
void Ex0Producer::DoStopRun(){
  m_exit_of_run = true;
  if(m_thd_run.joinable())
    m_thd_run.join();
  m_ifile.close();
}
```

6.2.8. DoReset

When the Producer goes into the error state. Only Reset command is acceptable. It is recommend to reset all member variable to original value when the producer object is instanced.
Here is example `DoReset` of Ex0Producer.

```
void Ex0Producer::DoReset(){
  m_exit_of_run = true;
  if(m_thd_run.joinable())
    m_thd_run.join();

  m_ifile = std::ifstream();
  m_thd_run = std::thread();
  m_ms_busy = std::chrono::milliseconds();
  m_exit_of_run = false;
}
```

For any case the data thread is running, it should be stopped.

6.2.9. DoTerminate

This method is called whenever an StopRun command is received from the Run Control. After the return of `DoTerminate`, the application will exit.
Here is example `DoStopRun` of Ex0Producer.

```
void Ex0Producer::DoTerminate(){
  m_exit_of_run = true;
  if(m_thd_run.joinable())
    m_thd_run.join();
}
```

The data thread is closed.

6.2.10. Send Event

In `DoStartRun`, a data thread is created with the function `Mainloop`. Here is the implementation of this function. The generated data Event is depending the value set by `DoInitialise` and `DoConfigure`.

```
void Ex0Producer::Mainloop(){
  auto tp_start_run = std::chrono::steady_clock::now();
  uint32_t trigger_n = 0;
  while(m_exit_of_run){
    auto ev = eudaq::Event::MakeUnique("Ex0Event");

    auto tp_trigger = std::chrono::steady_clock::now();
    auto tp_end_of_busy = tp_trigger + m_ms_busy;
    if(m_flag_ts){
      std::chrono::nanoseconds du_ts_beg_ns(tp_trigger - tp_start_run);
      std::chrono::nanoseconds du_ts_end_ns(tp_end_of_busy - tp_start_run);
      ev->SetTimestamp(du_ts_beg_ns.count(), du_ts_end_ns.count());
    }
    if(m_flag_tg)
      ev->SetTriggerN(trigger_n);

    std::vector<uint8_t> data_a(100, 0);
    uint32_t block_id_a = 0;
    ev->AddBlock(block_id_a, data_a);
    std::filebuf* pbuf = m_ifile.rdbuf();
    size_t len = pbuf->pubseekoff(0,m_ifile.end, m_ifile.in);
    pbuf->pubseekpos (0,m_ifile.in);
    std::vector<uint8_t> data_b(len);
    pbuf->sgetn ((char*)&(data_b[0]), len);
    uint32_t block_id_b = 2;
    ev->AddBlock(block_id_b, data_b);

    if(trigger_n == 0){
      ev->SetBORE();
    }
    SendEvent(std::move(ev));
    trigger_n++;
    std::this_thread::sleep_until(tp_end_of_busy);
  }
}
```

In each loop, a new object of Event named Ex0Event is created. Trigger number and

Timestamp are optional to be set depending the flags. A data block with 100 zeros is filled to the Event object by id 0. Another data block read from file is filled to some Event object by id 2.Then, this Event object is sent out by `SendEvent(std::move(ev))`. The first Event object has a flag BORE by method `eudaq::Event::SetBORE()`

Tags The `Event` class also provide the option to store tags. Tags are name-value pairs containing additional information which does not qualify as regular DAQ data which is written in the binary blocks. Particularly in the BORE this is very useful to store information about the exact sensor configuration which might be required in order to be able to decode the raw data stored. A tag is stored as follows:

```
event.SetTag("Temperature", 42);
```

The value corresponding to the tag can be set as an arbitrary type (in this case an integer), it will be converted to a STL string internally.

6.2.11. Error

The the case Producer fail to run the command function. Throw an exception like this `EUDAQ_THROW("dummy data file (" + m_dummy_data_path +") can not open for writing")` Since Mainloop function is running as an independent thread, the exception throw from it can not catches from outside.

7. Writing a Data Collector

DataCollector can take data from Producers and merge/sync those data. A DataCollector consisted of a CommandReceiver and a DataReceiver, where the first receives commands from the Run Control while the latter allows to receive binary data events from one or more Producers. A dedicated sync method should implemented inside of DataCollector to pack the incoming data. the DataCollector base class is provided in order to simplify the integration. Example code for producers is provided.

7.1. DataCollector Prototype

Listing 7.1 is part of the header file which declares the eudaq::Producer. You are required to write the user DataCollector derived from eudaq::DataCollector. There are 9 virtual methods, belonging to 2 categories, should be implemented at user side. The first category includes the methods `DoInitialise`, `DoConfigure`, `DoStopRun`, `DoStartRun`, `DoReset` and `DoTerminate` which are called by command received and should return as soon as possible. The other category includes the methods `DoConnect`, `DoDisconnect`, and `DoReveive` which response to a connection in establishing or deleting, or a new coming Event.

```
class DLLEXPORT DataCollector : public CommandReceiver {
public:
  DataCollector(const std::string &name, const std::string &runcontrol);
  void OnInitialise() override final;
  void OnConfigure() override final;
  void OnStartRun() override final;
  void OnStopRun() override final;
  void OnTerminate() override final;
  void OnStatus() override final;
  void Exec() override;

  //running in commandreceiver thread
  virtual void DoInitialise(){};
  virtual void DoConfigure(){};
  virtual void DoStartRun(){};
  virtual void DoStopRun(){};
  virtual void DoTerminate(){};

  //running in dataserver thread
  virtual void DoConnect(ConnectionSPC id) {}
  virtual void DoDisconnect(ConnectionSPC id) {}
  virtual void DoReceive(ConnectionSPC id, EventUP ev) = 0;

  void WriteEvent(EventUP ev);
  void SetServerAddress(const std::string &addr){m_data_addr = addr;};
  void StartDataCollector();
  void CloseDataCollector();
```

```
    bool IsActiveDataCollector(){return m_thd_server.joinable();}
  private:
    void DataHandler(TransportEvent &ev);
    void DataThread();

  private:
    std::thread m_thd_server;
    bool m_exit;
    std::unique_ptr<TransportServer> m_dataserver;
    std::string m_data_addr;
    FileWriterUP m_writer;
    std::string m_fwpatt;
    std::string m_fwtype;
    std::vector<std::shared_ptr<ConnectionInfo>> m_info_pdc;
    uint32_t m_dct_n;
    uint32_t m_evt_c;
    std::unique_ptr<const Configuration> m_conf;
  };
```

The virtual function Exec() is optional to be implemented in user DataCollector. Internally, the base Exec() to create 2 threads for command execution and and data receiving, itself then goes to a unlimited loop and can never return until the Terminate command. In case you are going to implement it yourselves, please read the source code to find detail information.

7.2. Example Code: DirectSave

Here is a example code of a derived DataCollector, named DirectSaveDataCollector. As what its name is hinting, it does nothing except saving all coming Event to disk directly. Printing out of the Event to screen is optional for debugging.

```
#include "eudaq/DataCollector.hh"
#include <iostream>
class DirectSaveDataCollector :public eudaq::DataCollector{
public:
  using eudaq::DataCollector::DataCollector;
  void DoConfigure() override;
  void DoReceive(eudaq::ConnectionSPC id, eudaq::EventUP ev) override;
  static const uint32_t m_id_factory = ↩
     eudaq::cstr2hash("DirectSaveDataCollector");

private:
  uint32_t m_noprint;
};

namespace{
  auto dummy0 = eudaq::Factory<eudaq::DataCollector>::
    Register<DirectSaveDataCollector, const std::string&, const std::string&>
    (DirectSaveDataCollector::m_id_factory);
}

void DirectSaveDataCollector::DoConfigure(){
  m_noprint = 0;
  auto conf = GetConfiguration();
  if(conf){
    conf->Print();
    m_noprint = conf->Get("DISABLE_PRINT", 0);
  }
}

void DirectSaveDataCollector::DoReceive(eudaq::ConnectionSPC id, ↩
    eudaq::EventUP ev){
  if(!m_noprint)
    ev->Print(std::cout);
  WriteEvent(std::move(ev));
}
```

Tow virtual methods are implemented in DirectSaveDataCollector, `DoConfigure()` and `DoReceive(eudaq::ConnectionSPC id, eudaq::EventUP ev)`. it registers itself to the correlated `eudaq::factory` by the hash number from the name string `DirectSaveDataCollector`.

7.3. Example Code: SyncTrigger

Now, a more realistic example. The full source code is available here Listing 7.3. It can merge the eduaq::Event by trigger number from the connected eudaq::Producer.

```
#include "eudaq/DataCollector.hh"
#include <mutex>
#include <deque>
#include <map>

class Ex2DataCollector:public eudaq::DataCollector{
public:
  Ex2DataCollector(const std::string &name,
       const std::string &rc);
  void DoConnect(eudaq::ConnectionSPC id) override;
  void DoDisconnect(eudaq::ConnectionSPC id) override;
  void DoReceive(eudaq::ConnectionSPC id, eudaq::EventUP ev) override;

  static const uint32_t m_id_factory = eudaq::cstr2hash("Ex2DataCollector");
private:
  std::mutex m_mtx_map;
  std::map<eudaq::ConnectionSPC, std::deque<eudaq::EventSPC>> m_conn_evque;
};
```

```

namespace{
  auto dummy0 = eudaq::Factory<eudaq::DataCollector>::
    Register<Ex2DataCollector, const std::string&, const std::string&>
    (Ex2DataCollector::m_id_factory);
}

Ex2DataCollector::Ex2DataCollector(const std::string &name,
           const std::string &rc):
  DataCollector(name, rc){
}

void Ex2DataCollector::DoConnect(eudaq::ConnectionSPC idx){
  std::unique_lock<std::mutex> lk(m_mtx_map);
  m_conn_evque[idx].clear();
}

void Ex2DataCollector::DoDisconnect(eudaq::ConnectionSPC idx){
  std::unique_lock<std::mutex> lk(m_mtx_map);
  m_conn_evque.erase(idx);
}

void Ex2DataCollector::DoReceive(eudaq::ConnectionSPC idx, eudaq::EventUP ↩
    ev){
  std::unique_lock<std::mutex> lk(m_mtx_map);
  eudaq::EventSP evsp = std::move(ev);
  if(!evsp->IsFlagTrigger()){
    EUDAQ_THROW("!evsp->IsFlagTrigger()");
  }
  m_conn_evque[idx].push_back(evsp);

  uint32_t trigger_n = -1;
  for(auto &conn_evque: m_conn_evque){
    if(conn_evque.second.empty())
      return;
    else{
      uint32_t trigger_n_ev = conn_evque.second.front()->GetTriggerN();
      if(trigger_n_ev< trigger_n)
  trigger_n = trigger_n_ev;
    }
  }

  auto ev_sync = eudaq::Event::MakeUnique("myEx2_Event");
  ev_sync->SetTriggerN(trigger_n);
  for(auto &conn_evque: m_conn_evque){
    auto &ev_front = conn_evque.second.front();
    if(ev_front->GetTriggerN() == trigger_n){
      ev_sync->AddSubEvent(ev_front);
      conn_evque.second.pop_front();
    }
  }
  WriteEvent(std::move(ev_sync));
}
```

Comparing to previous DirectSaveDataCollector example, 2 more virtual methods are implemented. There are more virtual methods implemented instead of only two in the DirectSaveDataCollector case. They are `DoConnect`, `DoDisconnect`. The first one will be called when a new connection from eudaq::Producer is created, and the other will be called when the connection is expired. The information of the correlated connection is provided by the incoming parameter. The lifetime of the connection between the eudaq::DataCollector and eudaq::Producer is a data-taking run. No `DoConfigure` method is implemented, and this DataCollector is not Configurable.

8. Writing a Data Converter

As a framework, EUDAQ itself is developed with no knowledge of how hardware data can be decoded. Only the user can know the specific detail of the readout data from hardware, so users are required to write a data converter derived from eudaq::DataConverter to convert to other format. The converted data then can be stored on disk or used as input data of any non-EUDAQ software.

Currently, as historical legacy, there are two different formats are provided along with the native raw eudaq::Event. They are eudaq::StandardEvent for pixel detector, and eudaq::LCEvent as an EUDAQ wrapper for LCIO counteraction.

8.1. Event Structure

eudaq::Event is the most important data container in EUDAQ system. eudaq::Event should be filled by detector data properly in order to transmit among eudaq clients, eg. Producer and DataCollector. Table 9 lists all the member variables inside of eudaq::event. Not all of these variable are mandated to be filled. For example, in case there is trigger number but no timestamp information, the m_ts_begin and m_ts_end can be left untouched.

Variable	Type	Default Value	Comment
m_type	uint32_t	-	Event type
m_version	uint32_t	-	Version of EUDAQ
m_flag	uint32_t	-	Event flag
m_stm_n	uint32_t	-	Stream/device number
m_run_n	uint32_t	-	Run number
m_ev_n	uint32_t	-	Event number
m_tg_n	uint32_t	-	Trigger number
m_extend	uint32_t	-	Reserved word
m_ts_begin	uint64_t	-	Timestamp at the begin of trigger
m_ts_end	uint64_t	-	Timestamp at the end of trigger
m_dspt	std::string	-	Description String
m_tags	std::map<std::string, std::string>	-	Tag pairs
m_blocks	std::map<uint32_t, std::vector<uint8_t>>	-	Raw data blocks
m_sub_events	std::vector<EventSPC>	-	Sub events

Table 9: all member variables in eudaq::Event

The eudaq::Event is also capable to contain sub event inside of itself.

8.1.1. RawDataEvent

eudaq::RawDataEvent is derived from eudaq::Event with the m_type always set to eudaq::cstr2hash("RawDataEvent"). There is no more member variables or methods than the base eudaq::Event. However the member variable m_extend is used as a identification number of sub type of event.

8.1.2. StandardEvent

eudaq::StandardEvent is derived from eudaq::Event with the m_type always set to eudaq::cstr2hash("StandardEvent"). Historically, it presents a beam telescope tracking-hit event with all hit information of sensor planes. The hit information is contained by eudaq::StandardPlane. The beam telescope planes are pixel sensors. The spatial position and signal amplitude of the fired pixels are Zero-Compressed inside eudaq::StandaredPlane.

8.2. Example Code: RawEvent2StdEvent

This example DataConverter is named Ex0RawEvent2StdEventConverter. As the hint by the name, it takes responsible to convert the eudaq::RawDataEvent to eudaq::StandardEvent. The sub type of eudaq::RawDataEvent is "my_ex0" which also used to calculate the hash and register to EUDAQ factory. If an eudaq::RawDataEvent object announcing its sub-type by "my_ex0" exists when doing the data converting, this object will be forwarded to that Ex0RawEvent2StdEventConverter.

```
#include "eudaq/StdEventConverter.hh"
#include "eudaq/RawDataEvent.hh"

class Ex0RawEvent2StdEventConverter: public eudaq::StdEventConverter{
public:
  bool Converting(eudaq::EventSPC d1, eudaq::StdEventSP d2, ↩
      eudaq::ConfigSPC conf) const override;
  static const uint32_t m_id_factory = eudaq::cstr2hash("my_Ex0");
};

namespace{
  auto dummy0 = eudaq::Factory<eudaq::StdEventConverter>::
    Register<Ex0RawEvent2StdEventConverter>(Ex0RawEvent2StdEventConverter::m_id_factory);
}

bool Ex0RawEvent2StdEventConverter::Converting(eudaq::EventSPC d1, ↩
    eudaq::StdEventSP d2, eudaq::ConfigSPC conf) const{
  auto ev = std::dynamic_pointer_cast<const eudaq::RawDataEvent>(d1);
  size_t nblocks= ev->NumBlocks();
  auto block_n_list = ev->GetBlockNumList();
  for(auto &block_n: block_n_list){
    std::vector<uint8_t> block = ev->GetBlock(block_n);
    std::vector<bool> channels;
    eudaq::uchar2bool(block.data(), block.data() + block.size(), channels);
    eudaq::StandardPlane plane(block_n, "my_ex0_plane", "my_ex0_plane");
    plane.SetSizeZS(channels.size(), 1, 0);
    uint32_t x = 0;
    for(size_t i = 0; i < channels.size(); ++i) {
      if(channels[i] == true) {
```

```
  plane.PushPixel(x, 1 , 1);
      }
      ++x;
    }
    d2->AddPlane(plane);
  }
  return true;
}
```

9. Writing a Monitor

eudaq::Monitor provides the the base class and common methods to monitor the data quality online. The internal implementation of eudaq::Monitor is very similar to eudaq::DataCollector. The only difference between them is that eudaq::Monitor accepts only a single connection from an eudaq::DataCollector or eudaq::Producer. (not implemented)

9.1. Monitor Prototype

```
class DLLEXPORT Monitor : public CommandReceiver {
public:
  Monitor(const std::string &name, const std::string &runcontrol,
          const unsigned lim, const unsigned skip_,
          const unsigned int skip_evts, const std::string &datafile = "");
  void OnIdle() override;
  void OnStartRun() override;
  void OnStopRun() override;
  virtual void OnEvent(EventSPC /*ev*/) = 0;
  void Exec() override;

  bool ProcessEvent();
protected:
  bool m_callstart;
  bool m_done;
  FileReaderSP m_reader;
  unsigned limit;
  unsigned skip;
  unsigned int skip_events_with_counter;
  unsigned int counter_for_skipping;
};
```

9.2. Example Code

TODO

9.3. Graphical User Interface

TODO

10. Writing a RunControl

In most user cases, the base eudaq::RunControl can meet the requirement of the controlling flow process upon the detector system which adapts EUDAQ framework. It is still possible to overload the default behavior of base eudaq::RunControl by implemented a derived RunContorl class with overload the virtual methods which are correlated to the command sending and client status checking.

10.1. RunControl Prototype

The command methods are `Initialise()`, `Configure()`, `StartRun()`, `StopRun()`, `Reset()` and `Terminate()`.

The Methods `DoConnect()`, `DoDisconnect()`, `DoStatus()` are also provided for the case the derived RunControl want be informed in time by the connection changing and eudaq::Status object coming.

10.2. RunControl Mode

There are two options when making the integration among EUDAQ RunControl and other DAQ systems which do not adapt the eudaq::Producer approach and can not talk with base eudaq::RunControl.

10.2.1. EUDAQ Working as Master DAQ

In this case, RunControl should behave as an entrance point to the full detector system. As the base eudaq::RunControl can not manager the controlling of the non-EUDAQ slaver DAQ, users are required to implement a specific RunControl class derived from eudaq::RunControl. With overloaded virtual methods, the user RunControl not only govern EUDAQ system but also talk to non-EUDAQ component by user specific communication protocol.
By this approach, the user RunControl can be instanced by the standard GUI RunControl launcher (euRun).

10.2.2. EUDAQ Working as Slaver DAQ

In this case, the base eudaq::RunControl can be left as what it is without modification or derivation. However, the standard GUI RunControl launcher is no reusable. The master DAQ should instance an object of eudaq::RunControl and call the correlated method whenever it want issue the command to EUDAQ subsystem and get the status report.

11. Support User Defined Event Type

In EUDAQ framework, the definition `data` usually means a object of eudaq::Event or its derivations.

11.1. Event

eudaq::Event is serializable as it is derived from eudaq::Serializable and implements

```
Event(Deserializer &);
virtual void Serialize(Serializer &) const;
```

There are 3 eudaq::Event derivations existing by default installation of EUDAQ. They are `RawDataEvent` (subsubsection 8.1.1), `StandardEvent` (subsubsection 8.1.2) and `LCEvent`. Certainly, the eudaq::Event derivations are serializable. But if there are additional member variable intoruded to derived Event, the new variable should be serialized along with the other variables defined in eudaq::Event. Otherwise, part data of user defined Event object are losted when doing the Event writing to disk by EUDAQ native format (subsection 11.2).

It is recommend to reuse the raw data blocks `m_blocks` of eudaq::Event. Then the base eudaq::Event will take care of serialize and deserialize the `m_blocks`.

11.2. FileWriter

Usually, the objects of eudaq::Event and its derivations are the only data going to be stored in disk for later offline analysis. User may prefer to write Event data in other format which is human readable or widely used by other software. User can defined a class derived from eudaq::FileWriter to take response the writing of specific Event type.

11.2.1. FileWriter Prototype

```
class DLLEXPORT FileWriter {
public:
  FileWriter();
  virtual ~FileWriter() {}
  void SetConfiguration(ConfigurationSPC c) {m_conf = c;};
  ConfigurationSPC GetConfiguration() const {return m_conf;};
  virtual void WriteEvent(EventSPC ) = 0;
  virtual uint64_t FileBytes() const {return 0;};
private:
  ConfigurationSPC m_conf;
};
```

11.2.2. Example Code: NativeFileWriter

No matter if the user defined Event reuses the `m_blocks` raw data block or does serialization of new variables itself, it can be write to disk in binary format (aka native) which can only be read by EUDAQ.

```cpp
#include "eudaq/FileNamer.hh"
#include "eudaq/FileWriter.hh"
#include "eudaq/FileSerializer.hh"

class NativeFileWriter : public eudaq::FileWriter {
public:
  NativeFileWriter(const std::string &patt);
  void WriteEvent(eudaq::EventSPC ev) override;
  uint64_t FileBytes() const override;
private:
  std::unique_ptr<eudaq::FileSerializer> m_ser;
  std::string m_filepattern;
  uint32_t m_run_n;
};

namespace{
  auto dummy0 = eudaq::Factory<eudaq::FileWriter>::
    Register<NativeFileWriter, std::string&>(eudaq::cstr2hash("native"));
  auto dummy1 = eudaq::Factory<eudaq::FileWriter>::
    Register<NativeFileWriter, std::string&&>(eudaq::cstr2hash("native"));
}

NativeFileWriter::NativeFileWriter(const std::string &patt){
  m_filepattern = patt;
}

void NativeFileWriter::WriteEvent(eudaq::EventSPC ev) {
  uint32_t run_n = ev->GetRunN();
  if(!m_ser || m_run_n != run_n){
    std::time_t time_now = std::time(nullptr);
    char time_buff[13];
    time_buff[12] = 0;
    std::strftime(time_buff, sizeof(time_buff),
      "%y%m%d%H%M%S", std::localtime(&time_now));
    std::string time_str(time_buff);
    m_ser.reset(new eudaq::FileSerializer((eudaq::FileNamer(m_filepattern).
             Set('X', ".raw").
             Set('R', run_n).
             Set('D', time_str))));
    m_run_n = run_n;
  }
  if(!m_ser)
    EUDAQ_THROW("NativeFileWriter: Attempt to write unopened file");
  m_ser->write(*(ev.get())); //TODO: Serializer accepts EventSPC
  m_ser->Flush();
}

uint64_t NativeFileWriter::FileBytes() const {
  return m_ser ?m_ser->FileBytes() :0;
}
```

11.3. FileReader

To reconstruct the Event from disk file in native format or user-defined format, class eudaq::FileReader is provided. Each writing data format should have its correlated implementation of FileReader to access the disk file.

11.3.1. FileReader Prototype

```
  class DLLEXPORT FileReader{
  public:
    FileReader();
    virtual ~FileReader();
    void SetConfiguration(ConfigurationSPC c) {m_conf = c;};
    ConfigurationSPC  GetConfiguration() const {return m_conf;};
    virtual EventSPC GetNextEvent() = 0;
  private:
    ConfigurationSPC m_conf;
  };
```

11.3.2. Example Code: NativeFileReader

```
#include "eudaq/FileSerializer.hh"
#include "eudaq/FileReader.hh"

class NativeFileReader : public eudaq::FileReader {
public:
  NativeFileReader(const std::string& filename);
  eudaq::EventSPC GetNextEvent()override;
private:
  std::unique_ptr<eudaq::FileDeserializer> m_des;
  std::string m_filename;
};

namespace{
  auto dummy0 = eudaq::Factory<eudaq::FileReader>::
    Register<NativeFileReader, std::string&>(eudaq::cstr2hash("native"));
  auto dummy1 = eudaq::Factory<eudaq::FileReader>::
    Register<NativeFileReader, std::string&&>(eudaq::cstr2hash("native"));
}

NativeFileReader::NativeFileReader(const std::string& filename)
  :m_filename(filename){
}

eudaq::EventSPC NativeFileReader::GetNextEvent(){
  if(!m_des){
    m_des.reset(new eudaq::FileDeserializer(m_filename));
    //TODO: check sucess or fail
  }
  eudaq::EventUP ev;
  uint32_t id = -1;
  m_des->PreRead(id);
  if(id != -1)
    ev = eudaq::Factory<eudaq::Event>::
      Create<eudaq::Deserializer&>(id, *m_des);
  return std::move(ev);
}
```

A. Platform Dependent Issues/ Solutions

A.1. Linux

TODO

A.2. MacOS

TODO

A.3. Windows

A.3.1. MSBUILD

This is the program that processes the project (solution) files and feeds it to the compiler and linker. If you have a working project file it is more or less straight forward. It has a very simple syntax:

```
MSBUILD.exe MyApp.sln /t:Rebuild /p:Configuration=Release
```

myApp.sln is the file you want to Process. The parameter `/target` (short `/t`) tells msbuild what to do in this case rebuild. You have all the options you need like: clean, build and rebuild. You can also specify your own targets. With the "parameter property" switch you can change the properties of your Project. Let's say you want to compile EUDAQ, you go in the build folder where the solution (sln) file is and type:

```
MSBUILD.exe EUDAQ.sln /p:Configuration=Release
```

One thing one has to keep in mind is that there are some default configurations. The default is a debug build for x86. If you want to have it different then you need to specify it in the command line. And one thing you want to have is a release build! With the /p switch you can overwrite properties like in this case the configuration. But you could also overwrite the compiler version it should use. Let's say you want to use VS 2013 then you have to specify it by writing:

```
MSBUILD.exe EUDAQ.sln /p:PlatformToolset=v120 /p:Configuration=Release
```

But be careful when changing the compiler settings. It is possible that some then link against an incompatible version of your external libraries.

A.3.2. Known Problems

- The environment variables are pulled in as properties therefore they can be overwritten in the project file or in the "vcxproj.user" file. So if for example your QT Project won't compile and keeps complaining about not finding the correct directory make sure you are not overwriting the QTDIR environment Variable with a Property.

Glossary

BORE beginning-of-run-event, basically a run header.
FSM finite-state machine.
LCIO Linear Collider I/O, the file format used by the analysis software.
NI the National Instrument system, for reading out the Mimosa 26 sensors.
TLU the Trigger Logic Unit.

Acknowledgements

This project receives funding from the European Unions Horizon 2020 Research and Innovation programme under Grant Agreement no. 654168. The support is gratefully acknowledged. *Disclaimer*: The information herein only reflects the views of its authors and not those of the European Commission and no warranty expressed or implied is made with regard to such information or its use.
Before, this work was supported by the Commission of the European Communities under the 6th Framework Programme "Structuring the European Research Area," contract number RII3–026126, and received funding from the European Commission under the FP7 Research Infrastructures project AIDA, grant agreement no. 262025.

References

[1] J. Dreyling-Eschweiler and H. Jansen, "EUDET-type beam telescopes", *Online Wiki*.
URL `https://telescopes.desy.de/`

[2] P. Roloff, "The EUDET high resolution pixel telescope", *Nucl. Instrum. Meth.*, **A604**, (2009), 265–268.

[3] H. Jansen, S. Spannagel, J. Behr, A. Bulgheroni, G. Claus *et al.*, "Performance of the EUDET-type beam telescopes", *EPJ Techniques and Instrumentation*, **3** (1), (2016), 7.
URL `http://dx.doi.org/10.1140/epjti/s40485-016-0033-2`

[4] A. Bulgheroni, "EUTelescope, the JRA1 tracking and reconstruction software: a status report", *EUDET-Memo-2008-48*.
URL `http://www.eudet.org/e26/e28/e615/e835/eudet-memo-2008-48.pdf`

[5] GitHub, "Mastering Markdown", *GitHub*.
URL `https://guides.github.com/features/mastering-markdown/`

[6] Git, "Git – local-branching-on-the-cheap", *Online Article*.
URL `https://git-scm.com/`

[7] E. Developers, "EUDAQ code on GutHub", *GitHub*.
URL `https://github.com/eudaq/eudaq`

[8] D. Shirokova, “Software Development for a common DAQ at test”, *DESY Summer-studetnts*.