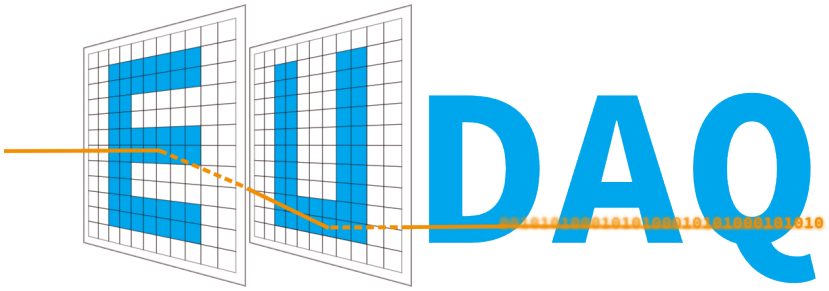

EUDAQ2 User Manual

EUDAQ2 Team

18th January 2022
EUDAQ v2.4.7

This document provides an overview of the EUDAQ software framework, the data acquisition framework originally developed for use 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 originally 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 (NI) system 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 analysis of 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 (see 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.

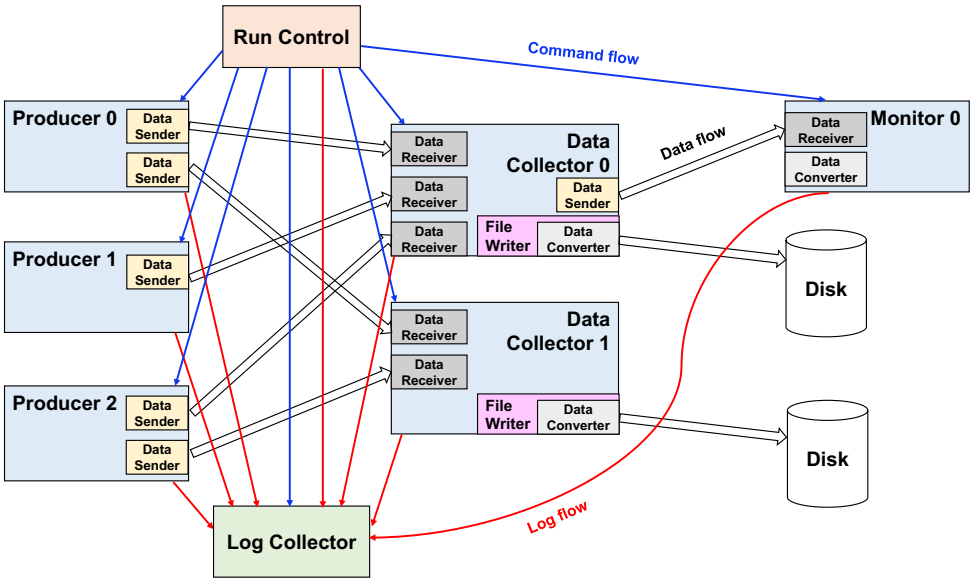

Figure 1: Schematic of the EUDAQ architecture [5].

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

The RunControl is the controller process which manages the full EUDAQ system. There should be always only one instance of RunControl for a runtime setup. All other data taking processes must have the network location of RunControl and anounces themelves to the RunControl. The RunControl takes in the paths of ini/config files and picks up and sends the correlated section to each of the connected processes. As the front end point to users, the RunControl will wait for user input by command line or GUI button click and issue the data taking command to whole EUDAQ system.

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 the core library,
  - `main/lib/lcio` contains the source code of the LCIO extension library,
  - `main/lib/root` contains the source code of the ROOT extension library,
  - `main/module` contains the source code of the 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 of user-provided 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 the EUDAQ project.
- `cmake` global folder which contains cmake files.
- `etc` contains configuration files of the EUDAQ installation.
- `conf` contains configuration files for running the beam telescope.
- `doc` contains source files of the documentation, such as this manual.

Each directory containing code has its own `src` and `include` subdirectories, as well as a local `CMakeLists.txt` and an 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 requiring any external dependencies, 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 an installation advice. Using the `*.md` file ending allows the Markdown language [6] to be applied. Accordingly, the content will be formatted on the the GitHub platform, where the code is hosted online.

Binary	Category	GUI/CLI	Description
`euRun`	Run Control	GUI	(Sec. 4.3.1)
`euLog`	Log Collector	GUI	(Sec. 4.3.2)
`StdEventMonitor`	legacy Monitor	GUI	
`euCliRun`	Run Control	CLI	(Sec. 4.3.1)
`euCliLogger`	Log Collector	CLI	(Sec. 4.3.2)
`euCliCollector`	Data Collector	CLI	(Sec. 4.3.3)
`euCliProducer`	Producer	CLI	(Sec. 4.3.4)
`euCliMonitor`	Monitor	CLI	(Sec. 4.3.5)
`euCliConverter`	Offline Tool	CLI	data file converter (Sec. 4.5.2)
`euCliReader`	Offline Tool	CLI	dump events from data file (Sec. 4.5.1)

Table 1: Overview of EUDAQ executables.

2.3. Executables

All executable programs from the different subdirectories are in 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 can be split into two different categories: firstly, processes, which are used for the data acquisition and communicating with the Run Control (DAQ); and, secondly, utilities, which are used before or after the data taking in order to access the data files (Test, Development, Tools). In Table 1, an overview of the most important EUDAQ executables is given.

2.4. Libraries

The libraries are installed in the `lib` directory under the install path in Linux/MacOS systems, and to the `bin` sub-folder in Windows systems. The suffix of libraries name can be `.so`, `.dylib`, `.dll` depending on operate systems. There are 3 category of libraries: core, extension and module:

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

In **??**, an overview of the all EUDAQ libraries is given.

,,

,,,,

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

The portable precompiled packages are avaliable from EUDAQ website. You could unpack the installation package to any folder and run the example directly. Please note, not all features of EUDAQ are enabled in the precompiled package, since it is not sure at compiling time whether the end user have the necessary dependency package installed.

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 problems occur during the installation process, please have a look at the issue tracker on GitHub.[2] Here you can search whether your problem had already been experienced by someone else, or you can open a new issue (see subsection 12.1).

3.2.1. Prerequisites

EUDAQ has some 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 GUIs of e.g. the RunControl or LogCollector.
- ROOT is required for the lagecy StdEventMonitor.

[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/master/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 [7]. It allows local version control and provides repositories, but also enables communication 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). For EUDAQ code development, requiring different versions (tags) or branches (development repositories), git is also used.

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 the StdEventMonitor) The Online Monitor uses 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 the LCIO option is enabled during the configuration step, source files will be download from the internet and compiled automatically.

3.2.2. Obtaining the source code

The EUDAQ source code is hosted on GitHub [8]. 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 EUDAQ code, please find the relevant information in see section 12.

Downloading the code (clone) We recommend using git to download the software, 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. The most important building options are given 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 RunControl (euRun) and LogCollector (euLog).
EUDAQ_BUILD_MANUAL	OFF	LaTex	Builds manual in pdf-format.
EUDAQ_BUILD_STDEVENT_MONITOR	auto	ROOT6	Builds lagecy standardevent 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_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 building files by running `cmake ..`, you can list all possible options 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 run. 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
```

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 a detector DAQ system in general.

4.1. Target Setup

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

`RunControl:` an example RunControl instanced by GUI application.

`LogCollector:` a default LogCollector instanced by GUI application.

`Producer:` Two example Producers instanced by the CLI command. Both of them produces events at a rate of 1 Hz, they may have time offset of start run.

`DataCollector:` an example DataCollectors instanced by the CLI command.

`Monitor:` an example monitor which prints out the asambled Event from the DataCollector in the command line terminal.

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 in the 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.[3] 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 pair of names seperated by a perioid in square brackets (e.g. `[Producer.Example]`). The name before peroid represents the type of process to which it applies, the one after peroid is the runtime name of the applied process. . There

[3] https://en.wikipedia.org/wiki/INI_file

is an exception, the section for Run Control is always `[RunControl]` in which there is no runtime name. Within each section, any number of parameters may be specified, in the form `Name = Value`. The EUDAQ native supported parmaters have a common prefix `EUDAQ_`. It is then up to the individual processes how these parameters are interpreted. During the initialization and configuration, each process gets its section and does not know about the other parts of the ini/conf files.

```
# example init file: Ex0.ini
[LogCollector.log]
EULOG_GUI_LOG_FILE_PATTERN = myexample_$12D.log

#[Producer.my_pd0]
#EX0_DEV_LOCK_PATH = /tmp/mydev0.lock

[Producer.my_pd1]
EX0_DEV_LOCK_PATH = /tmp/mydev1.lock
```

```
# example config file: Ex0.conf
[RunControl]
EX0_STOP_RUN_AFTER_N_SECONDS = 60
# from the base RunControl.cc
EUDAQ_CTRL_PRODUCER_LAST_START = my_pd0
EUDAQ_CTRL_PRODUCER_FIRST_STOP = my_pd0
# Steer which values to display in the GUI: producerName and displayed value are seperated by a ",".
ADDITIONAL_DISPLAY_NUMBERS = "log,_SERVER"

[Producer.my_pd0]
# connection to the data collector
EUDAQ_DC = my_dc
# config-parameters of the example producer
EX0_PLANE_ID = 0
EX0_DURATION_BUSY_MS = 10
# EX0_ENABLE_TIMESTAMP = 0
EX0_ENABLE_TRIGERNUMBER = 1
EX0_DEV_LOCK_PATH = mylock0

[Producer.my_pd1]
# connection to the data collector
EUDAQ_DC = my_dc
# config-parameters of the example producer
EX0_PLANE_ID = 1
EX0_DURATION_BUSY_MS = 100
EX0_ENABLE_TRIGERNUMBER = 1
# EX0_ENABLE_TIMESTAMP = 0
EX0_DEV_LOCK_PATH = mylock1

```

```
30 [DataCollector.my_dc]
31 # connection to the monitor
32 EUDAQ_MN = my_mon
33 EUDAQ_FW = native
34 EUDAQ_FW_PATTERN = run$3R_$12D$X
35 EUDAQ_DATACOL_SEND_MONITOR_FRACTION = 10
36 # config-parameters of the example data collector
37 EX0_DISABLE_PRINT = 1
38
39 [Monitor.my_mon]
40 EX0_ENABLE_PRINT = 0
41 EX0_ENABLE_STD_PRINT = 0
42 EX0_ENABLE_STD_CONVERTER = 1
```

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 to by the RunControl, and TCP port 44002 is listened to by the LogCollector to get connections from clients. The DataCollector will pick a random TCP port to listen to and get incoming data from its connected Producers if there is no specific port assigned explicitly by user.
When all EUDAQ processes run on the 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 ports for incoming connections, temporally shut down the firewall. Shutting down the firewall is operating-system dependent.

In this example, all processes will run on the same Linux computer, so 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, 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 in any order you want.

4.3.1. RunControl

There are 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 -n {code_name} -a tcp://{listening_port}
```

`-n ⟨code_name⟩:` optional, if it is not specified, default RunControl will be instanced.

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

For this example setup, we will startup the euRun GUI which internally instances the default Run Control and make it serve at TCP port 44000:

```
euRun -n Ex0RunControl -a tcp://44000
```

After executing the above command, a new GUI windows is shown in Figure 2.

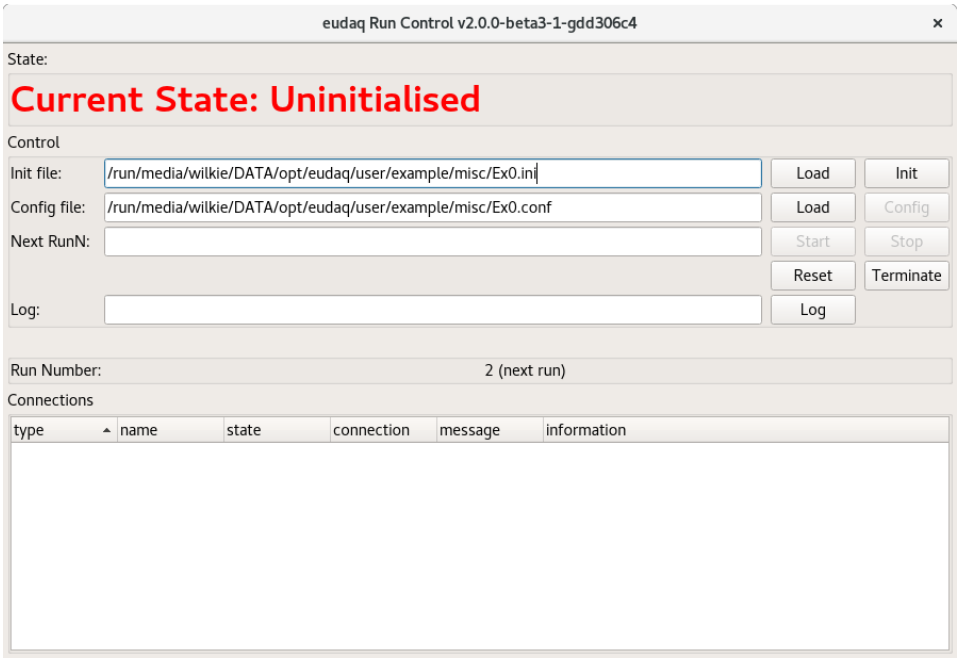

Figure 2: The Run Control graphical user interface.

Initialization Section

```
[RunControl]
# The Ex0RunControl does not need any paramters.
```

Configuration Section

```
[RunControl]
EX0_STOP_RUN_AFTER_N_SECONDS = 60
```

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 `euCliLogger`.
The command line pattern to startup a Log Collector is:

```
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 executing the above command, a new GUI window, as shown in Figure 3, is opened and the RunControl displays a new connection.

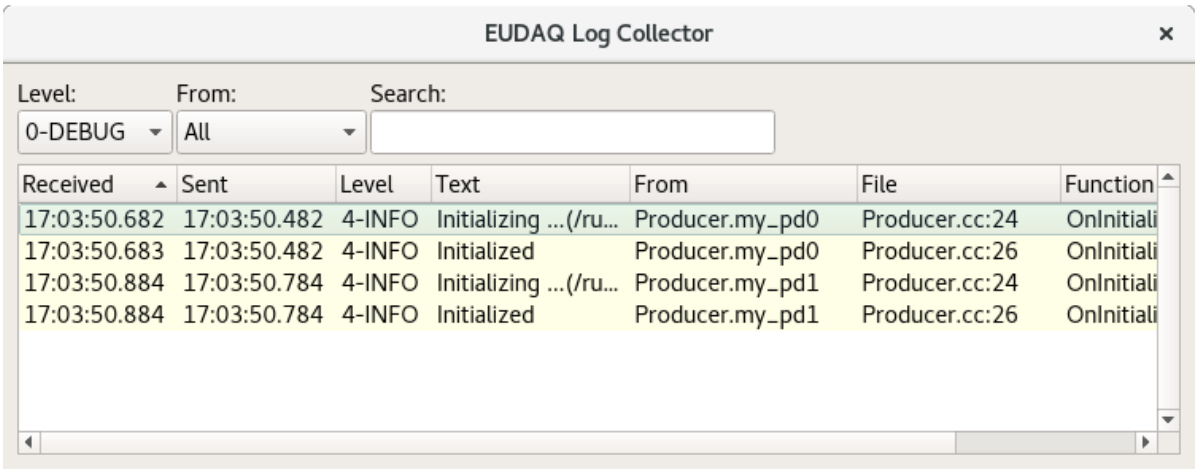

Figure 3: The Log Collector graphical user interface.

Initialization Section

```
[LogCollector.log]
# Currently, all LogCollectors have a hardcoded runtime name: log
EULOG_GUI_LOG_FILE_PATTERN = myexample_$12D.log
# the $12D will be converted a data/time string with 12 digits.
# the file path is allowed add as prefix in this file name pattern,
# otherwise the log file is saved in working folder.
```

Configuration Section

```
[LogCollector.log]
# Currently, all LogCollectors have a hardcoded runtime name: log
# nothing
```

4.3.3. DataCollector

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

```
euCliCollector -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, an example DataCollector `Ex0TgDataCollector` is available with the standard installation of EUDAQ. For this example setup, we will startup two instances of `Ex0TsDataCollector` with runtime names `my_dc` and `another_dc`

```
euCliCollector -n Ex0TgDataCollector -t my_dc -r tcp://localhost:44000 -a ↩
    tcp://45001
```

Initialization Section

```
[DataCollector.my_dc]
# nothing
```

Configuration Section

```
[DataCollector.my_dc]
EUDAQ_MN=my_mon
# send assambled event to the monitor with runtime name my_mon;
EUDAQ_FW=native
# the format of data file
EUDAQ_FW_PATTERN=$12D_run$6R$X
# the name pattern of data file
# the $12D will be converted a data/time string with 12 digits.
# the $6R will be converted a run number string with 6 digits.
# the $X will be converted the suffix name of data file.
# the file path is allowed add as a prefix to this name pattern,
# otherwise the data file is saved in working folder.
```

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, an example Producer `Ex0Producer` is available with the standard installation of EUDAQ. For this example setup, we will startup two instances of `Ex0Producer` with runtime names `my_pd0` and `my_pd0`

```
euCliProducer -n Ex0Producer -t my_pd0 -r tcp://localhost:44000
euCliProducer -n Ex0Producer -t my_pd1 -r tcp://localhost:44000
```

Initialization Section

```
[Producer.my_pd0]
EX0_DEV_LOCK_PATH = /tmp/mydev0.lock

[Producer.my_pd1]
EX0_DEV_LOCK_PATH = /tmp/mydev1.lock
```

Configuration Section

```
[Producer.my_pd0]
EUDAQ_DC=my_dc
# send events to the producer with runtime name my_dc.
# it is allowed to have a configure line as "EUDAQ_DC=his_dc,her_dc"
# to make the producer send events to muiltiple DataCollectors.
EX0_PLANE_ID=0
EX0_DURATION_BUSY_MS=1
EX0_ENABLE_TRIGERNUMBER=1

[Producer.my_pd1]
EUDAQ_DC=my_dc
# send events to the producer with runtime name my_dc.
EX0_PLANE_ID=1
EX0_DURATION_BUSY_MS=1
EX0_ENABLE_TRIGERNUMBER=1
```

4.3.5. Monitor

There is a text-based version called `euCliMonitor`. The command line pattern is:

```
euCliMonitor -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, an example Monitor `ExOMonitor` is available with the standard installation of EUDAQ. To simplify this example, the `ExOMonitor` has no graphic windows, it can only print Event in commnad line terminal. For this example setup, we will startup two instances of `ExOProducer` with runtime names `my_mon`

```
euCliMonitor -n ExOMonitor -t my_mon -r tcp://localhost:44000 -a tcp://45002
```

Initialization Section

```
[Monitor.my_mon]
# nothing
```

Configuration Section

```
[Monitor.my_mon]
EX0_ENABLE_PRINT=0
EX0_ENABLE_STD_PRINT=0
EX0_ENABLE_STD_CONVERTER=1
```

4.4. Operating

Once all the processes have been started, the RunControl GUI window will be as in Figure 4.

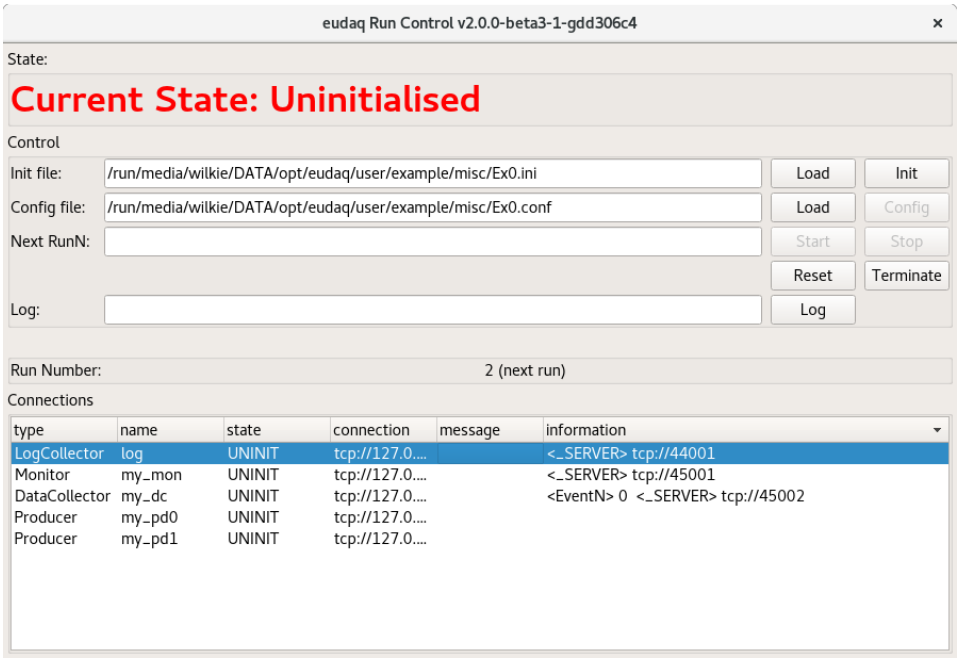

Figure 4: The RunControl with all connections of the example setup.

All of the processes report their status to the RunControl. Depending on their status, 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 an initialization command to all connected processes.
- Second, the appropriate configuration should be selected (see subsubsection 4.2.2 for creating and editing configurations). Then the `Config` button can be pressed, which will send a configuration command (with the contents of the selected configuration file) to all connected processes. 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.
- 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.
- 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.

4.5. After data taking

By default, EUDAQ provides tools to manage the the data saved in file by native EUDAQ format. Those tools could also be examples for users to write new code for more specific purpose.

4.5.1. Dump data

To dump print Event from data file, the tool `euCliReader` is provided. The command line pattern is: The command line pattern is:

```
euCliReader -i {input_file} -e {event_number_begin} -E {event_number_end} ↩
    -tg {trigger_number_begin} -TG {tigger_number_end} -ts {timestamp_begin} ↩
    -TS {timestamp_end} -s -std
```

`-i` ⟨`input_file`⟩`:` required, the path of the input data file

`-e` ⟨`event_number_begin`⟩`:` optional, the low limit of event number to be printed

`-E` ⟨`event_number_end`⟩`:` optional, the high limit of event number to be printed

`-tg` ⟨`trigger_number_begin`⟩`:` optional, the low limit of trigger number to be printed

`-TG` ⟨`trigger_number_end`⟩`:` optional, the high limit of trigger number to be printed

`-ts` ⟨`timestamp_begin`⟩`:` optional, the low limit of timestamp to be printed

`-TS` ⟨`timestamp_end`⟩`:` optional, the high limit of timestamp to be printed

`-s:` optional, enable the print of statistics

`-std:` optional, enable the Standard Event Converter and print out StdEvent

The option pairs `-e -E`, `-tg -TG` and `-ts -TS` apply range limites and pick up the most intreasting Event from data file. If an option pair is not specified by user, there will be not range limit for this option pair.

4.5.2. Convert data format

To convert Event from data file, the tool `euCliConverter` is provided. The command line pattern is: The command line pattern is:

```
euCliConverter -i {input_file} -o {output_file} -ip
```

`-i` ⟨`input_file`⟩`:` required, the path of the input data file

`-o` ⟨`output_file`⟩`:` required, the path of the output data file.

`-ip:` optional, enable the print of input Event

If the output file has the suffix `slcio` and LCIO feature of EUDAQ is enabled at compiling time, it will generate LCIO data file.

Class	Description
`eudaq::Producer`	Sec. 6
`eudaq::DataCollector`	Sec. 7
`eudaq::RunControl`	Sec. 10
`eudaq::Event`	Sec. 5.2.1
`eudaq::LogCollector`	Sec. 4.3.2
`eudaq::Monitor`	Sec. 9
`eudaq::FileWriter`	Sec. 11.2
`eudaq::FileReader`	Sec. 11.3
`eudaq::StdEventConverter`	Sec. 8
`eudaq::LCEventConverter`	Sec. 8
`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. 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 the EUDAQ framework. The minimum adaptation task is to write a Producer for each piece of hardware, a Data Collector to receive the data (a.k.a. Event) from the Producers.

5.1. Announcement of Derived Class

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

5.2. Serializable

As a distributed DAQ framework, a runtime setup of the 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 fed to serialized memory which then can be sent by plain binary to another application and reconstructed as a copy of the

Class	Description
`eudaq::Event`	Sec. 5.2.1
`eudaq::Configuration`	Contains configuration information
`eudaq::LogMessage`	Messages reported to the LogCollector
`eudaq::Status`	States reported to the RunControl

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 an 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 the hardware. The Producer is the EUDAQ component which creates the object eudaq::Event and feeds it the physics data from measurements. Table 6 lists the variables inside the eudaq::Event.

The `m_blocks` is physics data which can only 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

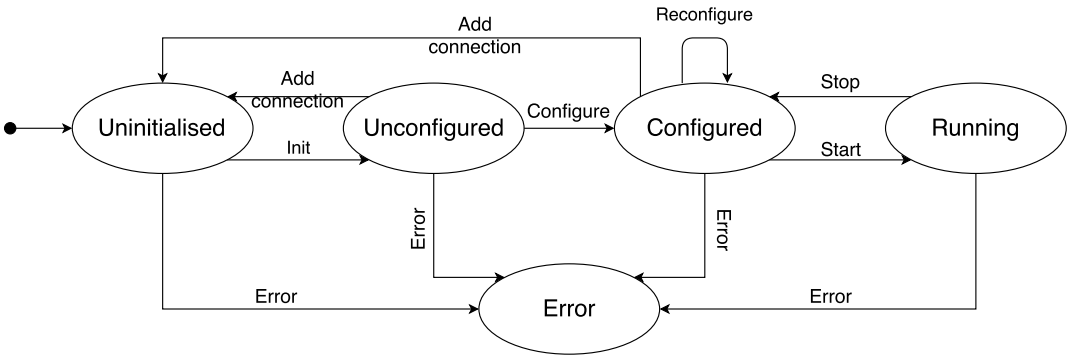

Figure 5: The FSM of EUDAQ.

number are optional to be set if you are going to use them to synchronize data from multiple stream/device. It is also possible to have sub events inside an eudaq::Event object. The sub eudaq::Event objects are held by std::shared_ptr, see the next section.

5.3. Ownership

std::shared_ptr and std::unique_ptr are heavily used in EUDAQ to hold the object pointers of the serializable class and the derivable class. They get rid of the unnecessary and ineffective memory copy and are exception safe for any memory leakage.

5.4. Command/Status Handling

5.4.1. RunControl and CommandReceiver

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

CommandReceiver eudaq::Producer, eudaq::DataCollector, eudaq::LogCollector and eudaq::Monitor are command receivers (eudaq::CommandReceiver) which executes the correlated function according to 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 (FSM) is implemented in both the RunControl and the CommandReceiver (see Figure 5) [9].

Each CommandReceiver 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
Error	eudaq::Status::STATE_ERROR	DoReset
Uninitialised	eudaq::Status::STATE_UNINIT	DoInitialise DoReset
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
Terminate	Terminate	DoTerminate	Any apart from Running	-

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 in the state of Uninitialised, it accepts the Initialise command and executes the DoInitialise function provided by the user. When it is initialized, it does not accept further Uninitialised commands until it is reset to the Uninitialised state.

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

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

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

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

Terminate Except for the state of Running, the terminate command is allowed on all other states. When the Terminate command is called, the EUDAQ processes will be closed. If the user has to do something before exiting the EUDAQ process, this should be entered into the DoTerminate function which will then carry out the process before exiting.

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 comamnds which allow binary data (eudaq::Event) to be sent to each destination.

6.1. Producer Prototype

Listing 6.1, below, 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 the EUDAQ framework, there are six virtual functions which should be implemented in the user producer. They are DoInitialise(), DoConfigure(), DoStopRun(), DoStartRun(), DoReset() and DoTerminate(). These command-related functions should return as soon as possible since no further command can be execuseted until the current command is finished.

```
32  class DLLEXPORT Producer : public CommandReceiver{
33  public:
34    Producer(const std::string &name, const std::string &runcontrol);
35
36    virtual void DoInitialise(){};
37    virtual void DoConfigure(){};
38    virtual void DoStartRun(){};
39    virtual void DoStopRun(){};
40    virtual void DoReset(){};
41    virtual void DoTerminate(){};
42    virtual void DoStatus(){};
43
44    void SendEvent(EventSP ev);
45    static ProducerSP Make(const std::string &code_name, const std::string ↩
          &run_name,
46          const std::string &runcontrol);
47
48  private:
49    void OnInitialise() override final;
50    void OnConfigure() override final;
51    void OnStartRun() override final;
52    void OnStopRun() override final;
53    void OnReset() override final;
54    void OnTerminate() override final;
55    void OnStatus() override;
56
```

```
57   private:
58     uint32_t m_pdc_n;
59     uint32_t m_evt_c;
60     std::mutex m_mtx_sender;
61     std::map<std::string, std::shared_ptr<DataSender>> m_senders;
62   };
```

The virtual function Exec() can optionally be implemented in the user Producer. If this is not implemented, the default Exec() will connect to the RunControl and goes into an infinite loop and will only return when the Terminate command is executed. In case you are going to implement an Exec() function, please read the source code to find detailed information.

6.2. Example Source Code: Dummy Event Producer

6.2.1. Declaration

Here we outline an example implementation of a user Producer. The full source code is available at file path user/example/module/src/Ex0Producer.cc. There are six command related functions, a constructor function and a `Mainloop` function in the declaration part:

```
12 class Ex0Producer : public eudaq::Producer {
13   public:
14   Ex0Producer(const std::string & name, const std::string & runcontrol);
15   void DoInitialise() override;
16   void DoConfigure() override;
17   void DoStartRun() override;
18   void DoStopRun() override;
19   void DoTerminate() override;
20   void DoReset() override;
21   void RunLoop() override;
22
23   static const uint32_t m_id_factory = eudaq::cstr2hash("Ex0Producer");
24 private:
25   bool m_flag_ts;
26   bool m_flag_tg;
27   uint32_t m_plane_id;
28   FILE* m_file_lock;
29   std::chrono::milliseconds m_ms_busy;
30   bool m_exit_of_run;
31 };
```

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

6.2.2. Register to Factory

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

```
34 namespace{
35   auto dummy0 = eudaq::Factory<eudaq::Producer>::
36     Register<Ex0Producer, const std::string&, const ↩
         std::string&>(Ex0Producer::m_id_factory);
37 }
```

6.2.3. Constructor

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

```
40 Ex0Producer::Ex0Producer(const std::string & name, const std::string & ↩
     runcontrol)
41   :eudaq::Producer(name, runcontrol), m_file_lock(0), m_exit_of_run(false){
42 }
```

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

6.2.4. DoInitialise

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

Here is an example `DoInitialise` of the Ex0Producer:

```
44 void Ex0Producer::DoInitialise(){
45   auto ini = GetInitConfiguration();
46   std::string lock_path = ini->Get("EX0_DEV_LOCK_PATH", "ex0lockfile.txt");
47   m_file_lock = fopen(lock_path.c_str(), "a");
48 #ifndef _WIN32
49   if(flock(fileno(m_file_lock), LOCK_EX|LOCK_NB)){ //fail
50     EUDAQ_THROW("unable to lock the lockfile: "+lock_path );
51   }
52 #endif
53 }
```

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

6.2.5. DoConfigure

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

Here is an example `DoConfigure` of Ex0Producer:

```
56 void Ex0Producer::DoConfigure(){
57   auto conf = GetConfiguration();
58   conf->Print(std::cout);
59   m_plane_id = conf->Get("EX0_PLANE_ID", 0);
60   m_ms_busy = std::chrono::milliseconds(conf->Get("EX0_DURATION_BUSY_MS", ↩
         1000));
61   m_flag_ts = conf->Get("EX0_ENABLE_TIMESTAMP", 0);
62   m_flag_tg = conf->Get("EX0_ENABLE_TRIGERNUMBER", 0);
63   if(!m_flag_ts && !m_flag_tg){
64     EUDAQ_WARN("Both Timestamp and TriggerNumber are disabled. Now, ↩
         Timestamp is enabled by default");
65     m_flag_ts = false;
66     m_flag_tg = true;
67   }
68 }
```

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

6.2.6. DoStartRun

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

```
70 void Ex0Producer::DoStartRun(){
71   m_exit_of_run = false;
72 }
```

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 a StopRun command is received from the RunControl. Here is an example `DoStopRun` of Ex0Producer:

```
74 void Ex0Producer::DoStopRun(){
75   m_exit_of_run = true;
76 }
```

6.2.8. DoReset

When the Producer goes into an error state, only the Reset command is acceptable. It is recommended to reset all member variables to their original value when the producer object is instanced.
Here is an example `DoReset` of Ex0Producer:

```
78 void Ex0Producer::DoReset(){
79   m_exit_of_run = true;
80   if(m_file_lock){
81 #ifndef _WIN32
82     flock(fileno(m_file_lock), LOCK_UN);
83 #endif
84     fclose(m_file_lock);
85     m_file_lock = 0;
86   }
87   m_ms_busy = std::chrono::milliseconds();
88   m_exit_of_run = false;
89 }
```

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

6.2.9. DoTerminate

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

```
91 void Ex0Producer::DoTerminate(){
92   m_exit_of_run = true;
93   if(m_file_lock){
94     fclose(m_file_lock);
95     m_file_lock = 0;
96   }
97 }
```

The data thread is then 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 depends on the value set by `DoInitialise` and `DoConfigure`.

```
99  void Ex0Producer::RunLoop(){
100   auto tp_start_run = std::chrono::steady_clock::now();
101   uint32_t trigger_n = 0;
102   uint8_t x_pixel = 16;
103   uint8_t y_pixel = 16;
104   std::random_device rd;
105   std::mt19937 gen(rd());
106   std::uniform_int_distribution<uint32_t> position(0, x_pixel*y_pixel-1);
107   std::uniform_int_distribution<uint32_t> signal(0, 255);
108   while(!m_exit_of_run){
109     auto ev = eudaq::Event::MakeUnique("Ex0Raw");
110     ev->SetTag("Plane ID", std::to_string(m_plane_id));
111     auto tp_trigger = std::chrono::steady_clock::now();
112     auto tp_end_of_busy = tp_trigger + m_ms_busy;
113     if(m_flag_ts){
114       std::chrono::nanoseconds du_ts_beg_ns(tp_trigger - tp_start_run);
115       std::chrono::nanoseconds du_ts_end_ns(tp_end_of_busy - tp_start_run);
116       ev->SetTimestamp(du_ts_beg_ns.count(), du_ts_end_ns.count());
117     }
118     if(m_flag_tg)
119       ev->SetTriggerN(trigger_n);
120
121     std::vector<uint8_t> hit(x_pixel*y_pixel, 0);
122     hit[position(gen)] = signal(gen);
123     std::vector<uint8_t> data;
124     data.push_back(x_pixel);
125     data.push_back(y_pixel);
126     data.insert(data.end(), hit.begin(), hit.end());
127
128     uint32_t block_id = m_plane_id;
129     ev->AddBlock(block_id, data);
130     SendEvent(std::move(ev));
131     trigger_n++;
132     std::this_thread::sleep_until(tp_end_of_busy);
133   }
134 }
```

In each loop, a new object of Event named Ex0Event is created. Trigger number and Timestamp are options to be set depending on 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 the 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 beginning-of-run-event (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

In the case when the Producer fails to run a command function an exception like this will be produced

```
EUDAQ_THROW("dummy data file (" + m_dummy_data_path +") can not open for writing")
```

An error message is then sent to the RunControl.

7. Writing a DataCollector

A DataCollector can take data from Producers and merge/synchronise those data. A DataCollector consists of a CommandReceiver and a DataReceiver, where the first receives commands from the Run Control while the latter receives binary data events from one or more Producers. A dedicated synchronisation method should be implemented inside of the 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, below, 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 nine virtual methods, belonging to two categories, which should be implemented by the user. 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 respond to a connection in establishing or deleting, or a new coming Event.

```
32  class DLLEXPORT DataCollector : public CommandReceiver, public ↩
        DataReceiver {
33  public:
34    DataCollector(const std::string &name, const std::string &runcontrol);
35    ~DataCollector() override;
36    virtual void DoInitialise();
37    virtual void DoConfigure();
38    virtual void DoStartRun();
39    virtual void DoStopRun();
40    virtual void DoReset();
41    virtual void DoTerminate();
42    virtual void DoStatus();
43    virtual void DoConnect(ConnectionSPC id);
44    virtual void DoDisconnect(ConnectionSPC id);
45    virtual void DoReceive(ConnectionSPC id, EventSP ev);
46    void WriteEvent(EventSP ev);
47    void SetServerAddress(const std::string &addr);
48    static DataCollectorSP Make(const std::string &code_name,
49        const std::string &run_name,
50        const std::string &runcontrol);
51  private:
52    void OnInitialise() override final;
53    void OnConfigure() override final;
54    void OnStartRun() override final;
55    void OnStopRun() override final;
56    void OnReset() override final;
```

```
57      void OnTerminate() override final;
58      void OnStatus() override final;
59      void OnConnect(ConnectionSPC id) override final;
60      void OnDisconnect(ConnectionSPC id) override final;
61      void OnReceive(ConnectionSPC id, EventSP ev) override final;
62    private:
63      std::string m_data_addr;
64      FileWriterSP m_writer;
65      std::mutex m_mtx_sender;
66      std::map<std::string, std::shared_ptr<DataSender>> m_senders;
67      std::string m_fwpatt;
68      std::string m_fwtype;
69      uint32_t m_dct_n;
70      uint32_t m_evt_c;
71      uint32_t m_fraction;
72      ConfigurationSPC m_conf;
73    };
```

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

7.2. Example Code: DirectSave

Here is an example code of a derived DataCollector, named DirectSaveDataCollector. As its name hints, it does nothing except save all incoming Event data to disk directly. Printing out of the Event to screen is optional for debugging.

```
1  #include "eudaq/DataCollector.hh"
2  //#include <iostream>
3
4
5  namespace eudaq {
6    class DirectSaveDataCollector:public DataCollector{
7      public:
8        using DataCollector::DataCollector;
9        void DoConfigure() override;
10       void DoReceive(ConnectionSPC id, EventSP ev) override;
11       static const uint32_t m_id_factory = ↩
             cstr2hash("DirectSaveDataCollector");
12
13     private:
14       uint32_t m_noprint;
15   };
16
```

```
17   namespace{
18     auto dummy0 = Factory<DataCollector>::
19       Register<DirectSaveDataCollector, const std::string&, const ↩
             std::string&>
20       (DirectSaveDataCollector::m_id_factory);
21   }
22
23   void DirectSaveDataCollector::DoConfigure(){
24     m_noprint = 0;
25     auto conf = GetConfiguration();
26     if(conf){
27       conf->Print();
28       m_noprint = conf->Get("DISABLE_PRINT", 0);
29     }
30   }
31
32   void DirectSaveDataCollector::DoReceive(ConnectionSPC id, EventSP ev){
33     if(!m_noprint)
34       ev->Print(std::cout);
35     WriteEvent(ev);
36   }
37 }
```

Two 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 eudaq::Event by trigger number from the connected eudaq::Producer.

```
1  #include "eudaq/DataCollector.hh"
2
3  #include <mutex>
4  #include <deque>
5  #include <map>
6  #include <set>
7
8  class ExOTgDataCollector:public eudaq::DataCollector{
9  public:
10   ExOTgDataCollector(const std::string &name,
11        const std::string &rc);
12   void DoConnect(eudaq::ConnectionSPC id) override;
13   void DoDisconnect(eudaq::ConnectionSPC id) override;
14   void DoConfigure() override;
15   void DoReset() override;
```

```
16   void DoReceive(eudaq::ConnectionSPC id, eudaq::EventSP ev) override;
17
18   static const uint32_t m_id_factory = ↩
         eudaq::cstr2hash("Ex0TgDataCollector");
19 private:
20   std::mutex m_mtx_map;
21   std::map<eudaq::ConnectionSPC, std::deque<eudaq::EventSPC>> m_conn_evque;
22   std::set<eudaq::ConnectionSPC> m_conn_inactive;
23   uint32_t m_noprint;
24 };
25
26 namespace{
27   auto dummy0 = eudaq::Factory<eudaq::DataCollector>::
28     Register<Ex0TgDataCollector, const std::string&, const std::string&>
29     (Ex0TgDataCollector::m_id_factory);
30 }
31
32 Ex0TgDataCollector::Ex0TgDataCollector(const std::string &name,
33                const std::string &rc):
34   DataCollector(name, rc){
35 }
36
37 void Ex0TgDataCollector::DoConnect(eudaq::ConnectionSPC idx){
38   std::unique_lock<std::mutex> lk(m_mtx_map);
39   m_conn_evque[idx].clear();
40   m_conn_inactive.erase(idx);
41 }
42
43 void Ex0TgDataCollector::DoDisconnect(eudaq::ConnectionSPC idx){
44   std::unique_lock<std::mutex> lk(m_mtx_map);
45   m_conn_inactive.insert(idx);
46   if(m_conn_inactive.size() == m_conn_evque.size()){
47     m_conn_inactive.clear();
48     m_conn_evque.clear();
49   }
50 }
51
52 void Ex0TgDataCollector::DoConfigure(){
53   m_noprint = 0;
54   auto conf = GetConfiguration();
55   if(conf){
56     conf->Print();
57     m_noprint = conf->Get("EX0_DISABLE_PRINT", 0);
58   }
59 }
60
```

```
61 void ExOTgDataCollector::DoReset(){
62   std::unique_lock<std::mutex> lk(m_mtx_map);
63   m_noprint = 0;
64   m_conn_evque.clear();
65   m_conn_inactive.clear();
66 }
67
68 void ExOTgDataCollector::DoReceive(eudaq::ConnectionSPC idx, ↩
     eudaq::EventSP evsp){
69   std::unique_lock<std::mutex> lk(m_mtx_map);
70   if(!evsp->IsFlagTrigger()){
71     EUDAQ_THROW("!evsp->IsFlagTrigger()");
72   }
73   m_conn_evque[idx].push_back(evsp);
74
75   uint32_t trigger_n = -1;
76   for(auto &conn_evque: m_conn_evque){
77     if(conn_evque.second.empty())
78       return;
79     else{
80       uint32_t trigger_n_ev = conn_evque.second.front()->GetTriggerN();
81       if(trigger_n_ev< trigger_n)
82   trigger_n = trigger_n_ev;
83     }
84   }
85
86   auto ev_sync = eudaq::Event::MakeUnique("ExOTg");
87   ev_sync->SetFlagPacket();
88   ev_sync->SetTriggerN(trigger_n);
89   for(auto &conn_evque: m_conn_evque){
90     auto &ev_front = conn_evque.second.front();
91     if(ev_front->GetTriggerN() == trigger_n){
92       ev_sync->AddSubEvent(ev_front);
93       conn_evque.second.pop_front();
94     }
95   }
96
97   if(!m_conn_inactive.empty()){
98     std::set<eudaq::ConnectionSPC> conn_inactive_empty;
99     for(auto &conn: m_conn_inactive){
100      if(m_conn_evque.find(conn) != m_conn_evque.end() &&
101    m_conn_evque[conn].empty()){
102   m_conn_evque.erase(conn);
103   conn_inactive_empty.insert(conn);
104       }
105     }
```

```
106     for(auto &conn: conn_inactive_empty){
107       m_conn_inactive.erase(conn);
108     }
109   }
110   if(!m_noprint)
111     ev_sync->Print(std::cout);
112   WriteEvent(std::move(ev_sync));
113 }
```

Compared to previous DirectSaveDataCollector example, two more virtual methods are implemented. They are `DoConnect`, `DoDisconnect`. The first, `DoConnect`, will be called when a new connection from eudaq::Producer is created, and the other, `DoDisconnect`, 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 details of the readout data from hardware, so users are required to write a data converter derived from eudaq::DataConverter to convert to another format. The converted data can then be stored on disk or used as input data for any non-EUDAQ software.

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

8.1. Event Structure

eudaq::Event is the most important data container in the EUDAQ system. eudaq::Event should be filled by detector data properly in order to transmit among eudaq clients, e.g. Producer and DataCollector. Table 6 lists all the member variables inside of eudaq::event. Not all of these variables have to be filled. For example, in case there is a trigger number but no timestamp information, the m_ts_begin and m_ts_end can be left untouched.
The eudaq::Event maye also have sub events.

8.1.1. RawDataEvent

eudaq::RawDataEvent is derived from eudaq::Event with the m_type always set to eudaq::cstr2hash("RawDataEvent"). It has the same member variables and functions as the base eudaq::Event. However the member variable m_extend is used as an identification number of the 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.

8.1.3. TTreeEvent

Another working example of converting the `raw` event to `ROOT TTree` format is also provided. The details and flow of converter is described in Annexe B.

8.2. Example Code: RawEvent2StdEvent

This example DataConverter is named Ex0RawEvent2StdEventConverter. As indicated by the name, it converts the eudaq::RawDataEvent to eudaq::StandardEvent. The sub type of eudaq::RawDataEvent is "my_ex0" which is also used to calculate the hash and register it to the 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/RawEvent.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("Ex0Raw");
};

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::RawEvent>(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);
    if(block.size() < 2)
      EUDAQ_THROW("Unknown data");
    uint8_t x_pixel = block[0];
    uint8_t y_pixel = block[1];
    std::vector<uint8_t> hit(block.begin()+2, block.end());
    if(hit.size() != x_pixel*y_pixel)
      EUDAQ_THROW("Unknown data");
    eudaq::StandardPlane plane(block_n, "my_ex0_plane", "my_ex0_plane");
    plane.SetSizeZS(hit.size(), 1, 0);
    for(size_t i = 0; i < y_pixel; ++i) {
      for(size_t n = 0; n < x_pixel; ++n){
  plane.PushPixel(n, i , hit[n+i*x_pixel]);
      }
    }
    d2->AddPlane(plane);
  }
```

```
37   return true;
38 }
```

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

```
31  class DLLEXPORT Monitor : public CommandReceiver, public DataReceiver{
32  public:
33    Monitor(const std::string &name, const std::string &runcontrol);
34    virtual void DoInitialise();
35    virtual void DoConfigure();
36    virtual void DoStartRun();
37    virtual void DoStopRun();
38    virtual void DoReset();
39    virtual void DoTerminate();
40    virtual void DoStatus();
41    virtual void DoReceive(EventSP ev);
42    void SetServerAddress(const std::string &addr);
43    static MonitorSP Make(const std::string &code_name,
44        const std::string &run_name,
45        const std::string &runcontrol);
46  private:
47    void OnInitialise() override final;
48    void OnConfigure() override final;
49    void OnStartRun() override final;
50    void OnStopRun() override final;
51    void OnReset() override final;
52    void OnTerminate() override final;
53    void OnStatus() override final;
54    void OnReceive(ConnectionSPC id, EventSP ev) override final;
55  private:
56    std::string m_data_addr;
57    uint32_t m_evt_c;
58  };
```

9.2. Example Code

Here is an example code of a derived Monitor, named Ex0Monitor. It gets configration paratmeter at configration time and do print out the infomation of the received Events. It might do an Event converting depending on how it is configured.

```cpp
#include "eudaq/Monitor.hh"
#include "eudaq/StandardEvent.hh"
#include "eudaq/StdEventConverter.hh"
#include <iostream>
#include <fstream>
#include <ratio>
#include <chrono>
#include <thread>
#include <random>

//----------DOC-MARK-----BEG*DEC-----DOC-MARK----------
class Ex0Monitor : public eudaq::Monitor {
public:
  Ex0Monitor(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 DoReceive(eudaq::EventSP ev) override;

  static const uint32_t m_id_factory = eudaq::cstr2hash("Ex0Monitor");

private:
  bool m_en_print;
  bool m_en_std_converter;
  bool m_en_std_print;
};

namespace{
  auto dummy0 = eudaq::Factory<eudaq::Monitor>::
    Register<Ex0Monitor, const std::string&, const
        std::string&>(Ex0Monitor::m_id_factory);
}

Ex0Monitor::Ex0Monitor(const std::string & name, const std::string &
    runcontrol)
  :eudaq::Monitor(name, runcontrol){
}

void Ex0Monitor::DoInitialise(){
  auto ini = GetInitConfiguration();
  ini->Print(std::cout);
}

```

```
45 void Ex0Monitor::DoConfigure(){
46   auto conf = GetConfiguration();
47   conf->Print(std::cout);
48   m_en_print = conf->Get("EX0_ENABLE_PRINT", 1);
49   m_en_std_converter = conf->Get("EX0_ENABLE_STD_CONVERTER", 0);
50   m_en_std_print = conf->Get("EX0_ENABLE_STD_PRINT", 0);
51 }
52
53 void Ex0Monitor::DoStartRun(){
54 }
55
56 void Ex0Monitor::DoStopRun(){
57 }
58
59 void Ex0Monitor::DoReset(){
60 }
61
62 void Ex0Monitor::DoTerminate(){
63 }
64
65 void Ex0Monitor::DoReceive(eudaq::EventSP ev){
66   if(m_en_print)
67     ev->Print(std::cout);
68   if(m_en_std_converter){
69     auto stdev = std::dynamic_pointer_cast<eudaq::StandardEvent>(ev);
70     if(!stdev){
71       stdev = eudaq::StandardEvent::MakeShared();
72       eudaq::StdEventConverter::Convert(ev, stdev, nullptr); //no conf
73       if(m_en_std_print)
74   stdev->Print(std::cout);
75     }
76   }
77 }
```

Two virtual methods are implemented in Ex0Monitor, `DoConfigure()` and `DoReceive(eudaq::EventUP ev)`. It registers itself to the correlated `eudaq::factory` by the hash number from the name string `Ex0Monitor`.

9.3. Graphical User Interface

Normally, to monitoring the data quality, graphic plots are required. It means that an external graphic library is required. ROOT, Qt and gnuplot can be the possible graphic library.

9.3.1. StdEventMonitor

StdEventMonitor is an EUDAQ version 1 legacy. It was named as OnlineMonitor. Actually, it can only display the plots from StandardEvent. If the Converter to StandardEvent of the incoming Event exists, StdEventMonitor will call the correlated Converter and do the converting itself. StdEventMonitor depends ROOT to generate and plot graph.

10. Writing a RunControl

In most user cases, the base eudaq::RunControl can meet the requirements of the controlling flow process to the detector system which use the EUDAQ framework. It is still possible to override the default behavior of the base eudaq::RunControl by implementing a derived RunControl class.

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 where the derived RunControl wants to be informed of any connection change and the object eudaq::Status is sent to the RunControl.

10.2. Example Code: auto stop

Here is an example code of a derived RunControl, named Ex0RunControl. It can check the elapsed time after the start of data taking, and stop the data taking after a configurable time period.

```
#include "eudaq/RunControl.hh"

class Ex0RunControl: public eudaq::RunControl{
public:
  Ex0RunControl(const std::string & listenaddress);
  void Configure() override;
  void StartRun() override;
  void StopRun() override;
  void Exec() override;
  static const uint32_t m_id_factory = eudaq::cstr2hash("Ex0RunControl");

private:
  uint32_t m_stop_second;
  bool m_flag_running;
  std::chrono::steady_clock::time_point m_tp_start_run;
};

namespace{
  auto dummy0 = eudaq::Factory<eudaq::RunControl>::
    Register<Ex0RunControl, const std::string&>(Ex0RunControl::m_id_factory);
}

Ex0RunControl::Ex0RunControl(const std::string & listenaddress)
```

```
24   :RunControl(listenaddress){
25   m_flag_running = false;
26 }
27
28 void Ex0RunControl::StartRun(){
29   RunControl::StartRun();
30   m_tp_start_run = std::chrono::steady_clock::now();
31   m_flag_running = true;
32 }
33
34 void Ex0RunControl::StopRun(){
35   RunControl::StopRun();
36   m_flag_running = false;
37 }
38
39 void Ex0RunControl::Configure(){
40   auto conf = GetConfiguration();
41   m_stop_second = conf->Get("EX0_STOP_RUN_AFTER_N_SECONDS", 0);
42   RunControl::Configure();
43 }
44
45 void Ex0RunControl::Exec(){
46   StartRunControl();
47   while(IsActiveRunControl()){
48     if(m_flag_running && m_stop_second){
49       auto tp_now = std::chrono::steady_clock::now();
50       std::chrono::nanoseconds du_ts(tp_now - m_tp_start_run);
51       if(du_ts.count()/1000000000>m_stop_second)
52   StopRun();
53     }
54     std::this_thread::sleep_for(std::chrono::milliseconds(500));
55   }
56 }
```

4 virtual methods are override in Ex0Monitor: `Configure()`, `StartRun()`, `StopRun()` and `Exec()`. It registers itself to the correlated `eudaq::factory` by the hash number from the name string `Ex0RunControl`.

10.3. RunControl Mode

There are two options when making the integration of the EUDAQ RunControl and other DAQ systems which do not adapt the eudaq::Producer approach and can not talk with base eudaq::RunControl. The Ex0RunControl example is working the first case.

10.3.1. EUDAQ Working as Master DAQ

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

10.3.2. EUDAQ Working as Slaver DAQ

In this case, the base eudaq::RunControl can be left as is without modification or derivation. However, the standard GUI RunControl launcher is not usable. The master DAQ should instance an object of eudaq::RunControl and call the correlated method whenever it wants to issue the command to the EUDAQ subsystem and get the status report.

11. Support User Defined Event Type

In the EUDAQ framework, the definition `data` usually means an 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 three eudaq::Event derivations in the default installation of EUDAQ. They are `RawDataEvent` (subsubsection 8.1.1), `StandardEvent` (subsubsection 8.1.2) and `LCEvent`. The eudaq::Event derivations are serializable. But if there are additional member variables introduced to derived Events, the new variable should be serialized along with the other variables defined in eudaq::Event. If this is not done, the unserialized data will be lost. It is recommend to reuse the raw data blocks `m_blocks` of eudaq::Event. Then the base eudaq::Event will 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. The user may prefer to write Event data in other format which is human readable or widely used by other software. In order to define a new file format, the user can define a class derived from eudaq::FileWriter.

11.2.1. FileWriter Prototype

```
30 class DLLEXPORT FileWriter {
31 public:
32   FileWriter();
33   virtual ~FileWriter() {}
34   void SetConfiguration(ConfigurationSPC c) {m_conf = c;};
35   ConfigurationSPC GetConfiguration() const {return m_conf;};
36   virtual void WriteEvent(EventSPC ) {};
37   virtual uint64_t FileBytes() const {return 0;};
38   static FileWriterSP Make(std::string type, std::string path);
39 private:
40   ConfigurationSPC m_conf;
41 };
```

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 write to disk in binary format (aka native) which can only be read by EUDAQ.

```
#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;
  }
```

```
42   if(!m_ser)
43     EUDAQ_THROW("NativeFileWriter: Attempt to write unopened file");
44   m_ser->write(*(ev.get())); //TODO: Serializer accepts EventSPC
45   m_ser->Flush();
46 }
47
48 uint64_t NativeFileWriter::FileBytes() const {
49   return m_ser ?m_ser->FileBytes() :0;
50 }
```

11.3. FileReader

To reconstruct the Event from a disk file in native format or user-defined format, the 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

```
26   class DLLEXPORT FileReader{
27   public:
28     FileReader();
29     virtual ~FileReader();
30     void SetConfiguration(ConfigurationSPC c) {m_conf = c;};
31     ConfigurationSPC GetConfiguration() const {return m_conf;};
32     virtual EventSPC GetNextEvent() {return nullptr;};
33     static FileReaderSP Make(std::string type, std::string path);
34   private:
35     ConfigurationSPC m_conf;
36   };
```

11.3.2. Example Code: NativeFileReader

```
1  #include "eudaq/FileDeserializer.hh"
2  #include "eudaq/FileReader.hh"
3
4  class NativeFileReader : public eudaq::FileReader {
5  public:
6    NativeFileReader(const std::string& filename);
7    eudaq::EventSPC GetNextEvent()override;
8  private:
9    std::unique_ptr<eudaq::FileDeserializer> m_des;
10   std::string m_filename;
11 };
12
```

```
13 namespace{
14   auto dummy0 = eudaq::Factory<eudaq::FileReader>::
15     Register<NativeFileReader, std::string&>(eudaq::cstr2hash("native"));
16   auto dummy1 = eudaq::Factory<eudaq::FileReader>::
17     Register<NativeFileReader, std::string&&>(eudaq::cstr2hash("native"));
18 }
19
20 NativeFileReader::NativeFileReader(const std::string& filename)
21   :m_filename(filename){
22 }
23
24 eudaq::EventSPC NativeFileReader::GetNextEvent(){
25   if(!m_des){
26     m_des.reset(new eudaq::FileDeserializer(m_filename));
27     if(!m_des)
28       EUDAQ_THROW("unable to open file: " + m_filename);
29   }
30   eudaq::EventUP ev;
31   uint32_t id;
32
33   if(m_des->HasData()){
34     m_des->PreRead(id);
35     ev = eudaq::Factory<eudaq::Event>::
36       Create<eudaq::Deserializer&>(id, *m_des);
37     return std::move(ev);
38   }  else  return nullptr;
39
40 }
```

12. Contributing to EUDAQ

12.1. Reporting Issues

The GitHub server, on which EUDAQ is hosted, provides a system for reporting bugs and for requesting new features. It is accessible at the following address: https://github.com/eudaq/eudaq/issues.
Here you may submit new reports (you are required to register first to do this), or follow the status of existing bugs and feature requests. This is recommended over (or at least, as well as) sending an email to the developers, as it ensures a record of the issue is available, and others may follow the progress.

12.2. Regression Testing

to be re-activated for EUDAQ2.

12.3. Commiting Code to the Main Repository

If you would like to contribute your code back into the main repository, please follow the "fork & pull request" strategy:

- Create a user account on github, log in
- "Fork" the (main) project on github (using the button on the page of the main repo)
- *Either*: clone from the newly forked project and add 'upstream' repository to local clone (change user names in URLs accordingly):

```
git clone https://github.com/your_account_name/eudaq eudaq
cd eudaq
git remote add upstream https://github.com/eudaq/eudaq.git
```

- *or* if edits were made to a previous checkout of upstream: rename origin to upstream, add fork as new origin:

```
cd eudaq
git remote rename origin upstream
git remote add origin https://github.com/your_account_name/eudaq
git remote -v show
```

- Optional: edit away on your local clone! But keep in sync with the development in the upstream repository by running

```
git fetch upstream        # download named heads or tags
git pull upstream master  # merge changes into your branch
```

on a regular basis. Replace `master` by the appropriate branch if you work on a separate one. Don't forget that you can refer to issues in the main repository anytime by using the string `eudaq/eudaq#XX` in your commit messages, where `XX` stands for the issue number, e.g.

```
[...]. this addresses issue eudaq/eudaq#1
```

- Push the edits to origin (our fork)

```
git push origin
```

(defaults to `git push origin master` where origin is the repo and master the branch)

- Verify that your changes made it to your github fork and then click there on the "compare & pull request" button
- Summarize your changes and click on "send"
- Thank you!

Working together on a branch: If you have a copy installed, and want to update it to the latest version, you do not need to clone the repository again, just change to the `eudaq` directory use the command:

```
git pull
```

to update your local copy with all changes commited to the central repository.

A. Platform Dependent Issues/ Solutions

A.1. Linux

A.1.1. GCC Compiler

Normally, CMake takes the default GCC Compiler shipped by Linux distrubtion, even if alternative GCC tool suits exist and have prior in binary search paths. To use an alternative GCC, it is necessary to set some CMake parameters which explicitly locate the paths of GCC binaries. Those parameters are CMAKE_CXX_COMPILER, CMAKE_C_COMPILER.

A.1.2. LLVM/Clang

LLVM/Clang Compiler has not been tested on Linux platform.

A.2. MacOS

A.2.1. Qt

Install Qt5 or later, e.g. by using MacPorts (http://www.macports.org/):

```
sudo port install qt5-mac-devel
```

A.3. Windows

A.3.1. Release/Debug Version

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
```

A.3.2. Qt

In case CMake can not find Qt5 installation at config time, set CMake option Qt5Widgets_DIR to the folder which contaion file Qt5WidgetsConfig.cmake .

B. ROOT TTree Converter

This Annexe[4] provides details of a working example of a converter from `raw` format to `ROOT TTree` format mentioned in 8.1.3. Any issues encountered during implementation of the instructions in this document should be reported at the dedicated platform.[5]

B.1. Event Structure in EUDAQ

In the EUDAQ data acquisition framework,`eudaq::Event` is the most important object in terms of data handling. It stores the data generated by the device under test (DUT). It is created by the `Producer` component of EUDAQ. The Producer then feeds physics data to fill it. It is the base class in EUDAQ and is serializable. Any object of type `Event` for any linked processes including Data Collector and Data Converter are to be derived from this class. The member variables of `eudaq::Event` are listed in table 9.

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 9: Variables of eudaq::Event.

Each variable describes a certain parameter/characteristic in the `Event` data. Some parameters are common and could be termed as *basic* parameters such as `m_run_n, m_ev_n, m_tg_n,` etc which represent Run number, Event number and Trigger number respectively. As is evident from the naming scheme, the `m_run_n` represents the number of run which could be same for multiple events while the `m_ev_n` parameter should vary from event to event. All of these mentioned parameters are of type `unint_32` i.e. unsigned integer with 32 bit. Other parameters which contain the hardware specific data can vary from device to device in type and size. The `m_block` variable in the above list holds the physics data from hardware. The content and type of these data is known to the user

[4] Authored by Sohail Amjad(s.amjad@ucl.ac.uk)
[5] https://github.com/eudaq/eudaq/issues

only. As is evident from their types, these can be more complex objects such as vectors and maps.
An `Event` may also contain sub-structure depending on the circumstances. These can vary in number and size significantly. The variable `m_extend` is used to identify and point to such sub-events. A complete and adequate handling of the `Event` needs taking care of these sub-blocks.

B.2. TTree Converter

The latest addition to the EUDAQ converters list is the TTree Converter. It works in a similar way as the other converters except that the data is converted to ROOT format which does not have a pre-defined Event class like LCIO to hold the objects of `eudaq::Event`. In this converter therefore, the member variables are to be stored in the branches of a `ROOT TTree`. There are two ways in which branches can be filled during the conversion process depending on the nature of parameters which are to be stored i.e. *basic* parameters and data *blocks*.

B.2.1. Converter Flow

The source code for the Converter can be found in `main/lib/ttree`. The dependencies are set during the compilation by providing path to the ROOT software. The Converter consists of a few components described in more detail below.

FileWriter This processor prepares the output file, write the filled `TTree` to it, and books the pre-defined branches for *basic* variables of the Event. The first part of the code given below registers the TTreeFileWriter to the core of EUDAQ. This essentially tells the EUDAQ which converter to execute given the extension of output file in the execution command.

```
namespace{auto dummy01 = Factory<FileWriter>::Register<TTreeFileWriter, ↩
    std::string&>(cstr2hash("root"));
```

The constructor function `TTreeFileWriter()` prepares the output file and also declared branches for fundamental variables. This declaration needs special care for the type of variables in the event. The file is written in the deconstructor function.

EventConverter This processor is called by the FileWriter. It is responsible for obtaining the *basic* parameters from the raw Event and filling them to the tree as branches. As the number and type of these parameters is known already, they are stored to the pre-defined branches declared in the FileWriter. It also looks for sub-events. In case a sub-event is found, it calls the adequate process to deal with it.

RawEvent2TTreeConverter This is the process to take care of the sub-events inside an event. It will identify the type of the sub-event with a given tag, and will call the relevant converter to deal with it. There can be multiple types and each can be converted using the most suitable scheme. For example if the subtype of the event in `raw` data is `"Ex0Raw"`, it will call the `Ex0RawEvent2TTreeEventConverter`.

B.2.2. Ex0RawEvent2TTreeEventConverter

The above mentioned three components are sufficient for converting the basic parameters. For conversion of the data blocks a dedicated conversion process needs to be defined. Since the type and detail of this data is only known to the user, this part is mainly to be developed by the users to meet their needs. A working example is provided with the name of `Ex0RawEvent2TTreeEventConverter`. The source code can be found in `user/example/module/src/`. This converts the variable `m_block` containing `block_id` and respective blocks of data. It declares the type of event it is designed for via

```
static const uint32_t m_id_factory = eudaq::cstr2hash("Ex0Raw");
```

So if an event or sub-event announces itself with the tag `"Ex0Raw"`, it will be sent to this converter for adequate conversion.

```
1  #include "eudaq/TTreeEventConverter.hh"
2  #include "eudaq/RawEvent.hh"
3
4  class Ex0RawEvent2TTreeEventConverter: public eudaq::TTreeEventConverter{
5  public:
6    bool Converting(eudaq::EventSPC d1, eudaq::TTreeEventSP d2, ↩
       eudaq::ConfigSPC conf) const override;
7    static const uint32_t m_id_factory = eudaq::cstr2hash("Ex0Raw");
8  };
9
10 namespace{
11   auto dummy0 = eudaq::Factory<eudaq::TTreeEventConverter>::
12     Register<Ex0RawEvent2TTreeEventConverter>(Ex0RawEvent2TTreeEventConverter::m_id_factory
13 }
14
15 bool Ex0RawEvent2TTreeEventConverter::Converting(eudaq::EventSPC d1, ↩
     eudaq::TTreeEventSP d2, eudaq::ConfigSPC conf) const{
16   auto ev = std::dynamic_pointer_cast<const eudaq::RawEvent>(d1);
17   uint32_t id = ev->GetExtendWord();
18   std::cout << "ID " << id << std::endl;
19   size_t nblocks= ev->NumBlocks();
20   auto block_n_list = ev->GetBlockNumList();
21   std:: cout << " blocks " << nblocks << std::endl;
22   for(auto &block_n: block_n_list){
23     std::vector<uint8_t> block = ev->GetBlock(block_n);
24     if(block.size() < 2)
```

```
25        EUDAQ_THROW("Unknown data");
26      uint8_t x_pixel = block[0];
27      uint8_t y_pixel = block[1];
28      std::vector<uint8_t> hit(block.begin()+2, block.end());
29      std::vector<uint8_t> hitxv;
30      if(hit.size() != x_pixel*y_pixel)
31        EUDAQ_THROW("Unknown data");
32      TString temp = "block";
33      if (d2->GetListOfBranches()->FindObject(temp)) ↩
            d2->SetBranchAddress(temp,&x_pixel);
34      else
35        d2->Branch(temp,&x_pixel,"x_pixel/b:y_pixel/b");
36    }
37    return true;
38  }
```

In this example, the content of data blocks is a vector of integers while `block_id`s are also integers so they are simply stored to respective branches without any further treatment. One of the important things to consider at this point is that the number of blocks cannot be known beforehand and they may very event to event. Therefore it will not be possible to book the `TTree` branches in advance. Hence these branches are created on the go as the blocks are identified in the `raw` event. The converter checks if the branch is already there for a specific `block_id` and fills it. If the branch is not there, it creates it. For more complex type of data blocks, an optimized approach can be adopted to store the member variables in a list of branches, as leaves of a branch etc, according to suitability for the hardware data.

B.2.3. Execution and Storage

The converter can be executed using the command line as for the other converters. The command to be executed is following:

```
./euCliConverter -i input.raw -o output.root
```

The output file name extension tells the EUDAQ to call the adequate conversion process. The converted files are stored to the disk. This behaviour can be modified to route the converter output data towards other linked processes such as monitoring.

B.2.4. An Application

As a demonstration of the Converter at work, data from AHCAL beam test has been taken for test. The data consists of *basic* parameters as described above, as well as sub-events tagged as `CaliceObject` and `DesyTableRaw`. Therefore two dedicated converters are written for these. The converter announces itself to the eudaq core in the following way, declaraing the type of sub-event it is designed to handle.

```
static const uint32_t m_id_factory = eudaq::cstr2hash("CaliceObject");
```

The sub-events will then be dealt with using the conversion rules in this converter. The converter can be modified to handle the specific structure and type of the data. Different numbers of data blocks can be obtained and stored accordingly. The `CaliceObject` here, for example, consists mostly of 8 blocks though there can occasionally be up to 10 due to multiple ASIC triggers in the same event. These blocks are of varying size and contain different types of data. The converter can take the contents of block and store them in a most suitable way for the user. The blocks of zero size are not converted. The larger blocks containing the ASIC data will need the specific decoder to decipher the data, before storing in one of the allowed formats for a `TTRee` Branch.

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 Union's 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 6$^{\text{th}}$ 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] S. Spannagel, "Test Beam Measurements for the Upgrade of the CMS Pixel Detector and Measurement of the Top Quark Mass from Differential Cross Sections", Ph.D. thesis, U. Hamburg, Dept. Phys., Hamburg (2016).
URL http://bib-pubdb1.desy.de/search?cc=Publication+Database&of=hd&p=reportnumber:DESY-THESIS-2016-010

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

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

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

[9] D. Shirokova, "Software Development for a common DAQ at test", *DESY Summer-studetnts*.