Published by IOP Publishing for Sissa Medialab
Received: *July 30, 2019*
Accepted: *October 14, 2019*
Published: *October 23, 2019*

EUDAQ2 — A flexible data acquisition software framework for common test beams

Y. Liu,[a,1] M.S. Amjad,[b] P. Baesso,[c] D. Cussans,[c] J. Dreyling-Eschweiler,[a] R. Ete,[a] I. Gregor,[a] L. Huth,[a] A. Irles,[d] H. Jansen,[a] K. Krueger,[a] J. Kvasnicka,[e,a] R. Peschke,[a,f] E. Rossi,[a] A. Rummler,[g] F. Sefkow,[a] M. Stanitzki,[a] M. Wing[b,a] and M. Wu[a]

[a]*Deutsches Elektronen-Synchrotron, Notkestr. 85, 22607 Hamburg, Germany*
[b]*Department of Physics and Astronomy, University College London, Gower Street, London WC1E 6BT, United Kingdom*
[c]*University of Bristol, Tyndall Avenue, Bristol BS8 1TL,United Kingdom*
[d]*Laboratoire de l'Accélerateur Linéaire (LAL), CNRS/IN2P3 et Université de Paris-Sud XI, Centre Scientifique d'Orsay, Bâtiment 200, BP 34, F-91898 Orsay, CEDEX, France*
[e]*Institute of Physics of the Czech Academy of Sciences, Na Slovance 2, 18221 Prague 8, Czech Republic*
[f]*Department of Physics and Astronomy, University of Hawai'i at Mānoa, Watanabe 416, 2505 Correa Road, Honolulu, HI 96822, U.S.A.*
[g]*European Organization for Nuclear Research (CERN), 1211 Geneva 23, Switzerland*

E-mail: yi.liu@desy.de

Abstract: The data acquisition software framework, EUDAQ, was originally developed to read out data from the EUDET-type pixel telescopes. This was successfully used in many test beam campaigns in which an external position and time reference were required. The software has recently undergone a significant upgrade, EUDAQ2, which is a generic, modern and modular system for use by many different detector types, ranging from tracking detectors to calorimeters. EUDAQ2 is suited as an overarching software that links individual detector readout systems and simplifies the integration of multiple detectors. The framework itself supports several triggering and event building modes. This flexibility makes test beams with multiple detectors significantly easier and more efficient, as EUDAQ2 can adapt to the characteristics of each detector prototype during testing. The system has been thoroughly tested during multiple test beams involving different detector prototypes. EUDAQ2 has now been released and is freely available under an open-source license.

Keywords: Control and monitor systems online; Data acquisition concepts; Detector control systems (detector and experiment monitoring and slow-control systems, architecture, hardware, algorithms, databases); Software architectures (event data models, frameworks and databases)

ArXiv ePrint: 1907.10600

[1]Corresponding author.

Contents

1 Introduction

The next generation of particle physics experiments requires detectors with an outstanding performance to be designed, built and tested. The challenges involve spatial resolutions to the micron level, picosecond timing resolution and more on-detector intelligence. At the same time, the material budget needs to be further reduced compared to present systems, which requires novel solutions for the readout, powering and cooling of the detectors. As a part of any successful R&D program a set of test beams for each new detector are required to demonstrate its capabilities and performance.

In order to facilitate these detector test beams, high resolution pixel beam telescopes, the so-called EUDET-type pixel beam telescopes [1] (section 4.1) have been developed as a common infrastructure available to any R&D group. The accompanying data acquisition software, EUDAQ [2], was originally developed to read out the data from the EUDET-type pixel telescopes and was therefore closely connected to its data acquisition (DAQ) architecture. During a decade of usage many user groups have integrated their DAQ system into EUDAQ and have successfully combined their data with the telescope data, based on common trigger IDs. Together with the EUTelescope software package, a common pixel telescope data analysis framework, the EUDET-type pixel beam telescopes offer the whole infrastructure for detector development from the initial measurements to the final results. Furthermore the availability of these telescopes at the test beam facilities at CERN, DESY, ELSA(Bonn) and SLAC provided users the possibility to move their setup between test beams and use the same interface to the telescopes.

The recent development of EUDAQ2 provides a more flexible DAQ software framework for operating detector prototypes at test beams worldwide. The emerging need for running several detectors together with a telescope as a so-called system test is extending the use case way beyond the original conception of EUDAQ. Being a major upgrade to EUDAQ, EUDAQ2 has been completely rewritten using modern C++ and was designed to become even more versatile and usable for an extended range of detectors. The modular and cross-platform data acquisition framework serves as a flexible and simple-to-use data taking software for the EUDET-type pixel beam telescopes while allowing for the easy integration of many other detectors. EUDAQ2 is freely available [3] and distributed under the LGPLv3 [4] open-source license.

2 EUDAQ2 architecture

EUDAQ2, like its predecessor EUDAQ, is implemented in C++, taking advantage of many of the powerful features provided by the C++11 standard [5]. Compatibility across operating systems and compilers is one of the EUDAQ2 design principles, hence it only uses standard C++ language features and POSIX [6] system routines. Therefore, EUDAQ2 can run natively on Linux, MacOS and Windows. To build an EUDAQ2 system, different compilers such as GCC, LLVM/Clang and MS Visual C++ are supported. The build and installation processes are configured using CMake [7]. EUDAQ2 is a distributed DAQ framework with its communication protocol running on a custom TCP/IP stack.

Figure 1 shows a schematic overview on the EUDAQ2 framework: each component of EUDAQ2 can run anywhere on the network on separate machines, entirely operating-system independent, and connect to each other using the EUDAQ2 data taking setup at run-time. Within the EUDAQ2 framework, each detector hardware is being controlled and read out by an individual EUDAQ2 instance. At the same time, the data streams from different detectors can be merged using well-defined synchronization mechanisms and stored to disk.

2.1 Distributed run-time roles

A typical EUDAQ2 setup is split into several run-time instances with different roles, each communicating via the network. Table 1 gives an overview of the roles of each EUDAQ2 instance.

The `Run Control` is at the core of each EUDAQ2 system. Every EUDAQ2 system requires exactly one running instance of the `Run Control`. All other EUDAQ2 participating instances must be made aware of the network location of the `Run Control` and announce themselves to the `Run Control` at startup. The `Run Control` reads and parses the initialization/configuration files and distributes the corresponding settings to each connected EUDAQ2 instance according to the instance's run-time name. It also serves as user interface via an optional graphical interface. The `Run Control` is responsible for the start and stop the data taking.

A `Producer` controls underlying detector hardware which participates in the EUDAQ2 data taking. Each detector hardware needs to have a dedicated `Producer`. Normally, there are several

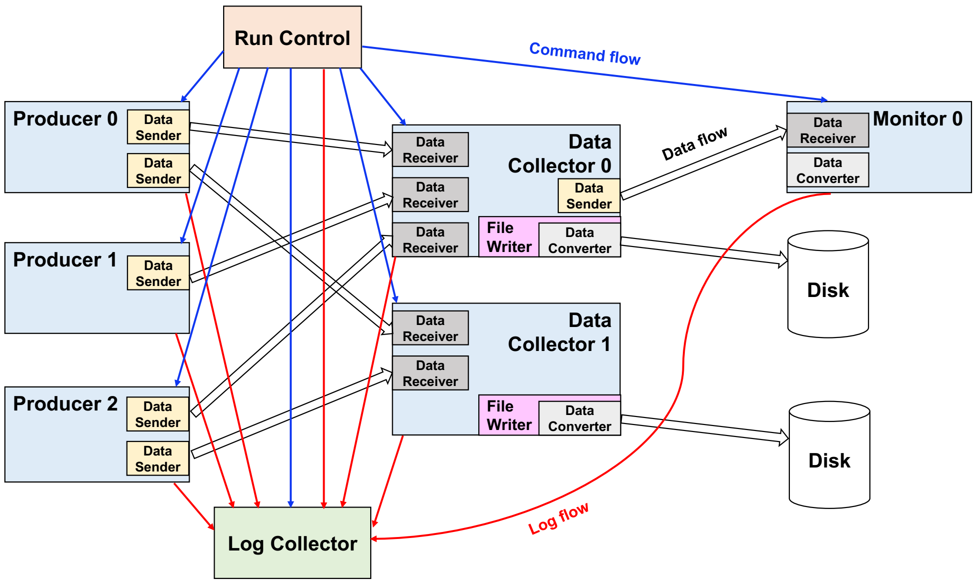


Figure 1. A schematic view of the EUDAQ2 architecture.

Table 1. EUDAQ2 roles.

EUDAQ2 Role	Description
`Run Control`	Central controller for full EUDAQ2 system
`Producer`	Controls an individual detector, and sends detector data to EUDAQ2
`Data Collector`	Collects and merges data from individual `Producers`, then stores the data to disk file
`LogCollector`	Collects log messages from the entire EUDAQ2 system
`Monitor`	Online monitoring of data quality
`Offline Tools`	Fast offline data conversion and data analysis

`Producers` running in parallel, as the data taking during a test beam consists of several detectors. The `Producers` feed the detector data into the distributed EUDAQ2 framework. A `Producer` also responds to `Run Control` commands and manages the detector hardware accordingly. The `Producer` is the only part where the user is required to make a dedicated and hardware-specific implementation, technically by employing the C++ polymorphism mechanism. A `Producer` may support both an "internal" and "external" loop for interacting with the hardware and retrieving data, when it is available. For the "internal" loop, the `Producer` provides a loop/thread internally, manages the *START* and *STOP* commands and handles the error exceptions. This is the simplest

2019 JINST 14 P10033

way for users to integrate their hardware into EUDAQ2. This might not be suitable for certain DAQ systems, which provide their own read-out loop and state-machine. Therefore another mode, the "external" loop mode, where EUDAQ2 is not managing the readout but merely receives data, whenever it is made available, is provided.

The `Data Collectors` collects the data from the individual detectors via their `Producers` and write data to disk. A `Producer` can be configured to send data to one or several `Data Collector` instances, depending on the user needs during data taking. A `Data Collector` can be configured to receive data from one or several `Producers`, hence supporting very flexible ways of event-building when using different detectors. The synchronization of the data from different `Producers` can be done either using straight-to-disk, skipping event validation and synchronization, sync-with-arrival-order, sync-with-trigger and sync-with-timestamp, all of them fully supported by EUDAQ2. Basic sanity checks such as testing the consistency of event numbers or TriggerIDs are implemented by default. The API of the `Data Collector` allows straightforward inclusion of additional synchronization methods if required by a user, see section 3.2.

The `LogCollector` gathers logging information from all EUDAQ2 instances and displays them centrally in one unified logging window. A single log file is stored along with the corresponding data file for later reference. This simplifies the tracking of problems encountered during data-taking. Only a single instance of the `LogCollector` is allowed in any EUDAQ2 setup. If a setup does not provide a `LogCollector` instance during the run, the log messages just go to the local screen where an EUDAQ2 instance runs.

The `Monitor` analyses the incoming data stream and generates a set of histograms, ensuring data quality online. Hardware failures of setup issues can be tracked down efficiently utilizing the `Monitor`. A legacy `OnlineMonitor` from EUDAQ is shipped to maintain compatibility and to simplify migration to EUDAQ2. The `Monitor` is able to run offline with disk data as well, which allows for quick data quality verification.

Data decoding can be performed either online or offline. Independently, the `DataConverter` is the only point where specific decoding routines need to be implemented. The `DataConverter` to be called for a specific event is derived from the event type. A corresponding `DataConverter` can be retrieved and called by the `Monitor` and other offline analysis tools.

2.2 State & command model

Any EUDAQ2 instance has to maintain a run-time state. To change between the individual EUDAQ2 states, a set of EUDAQ2 commands is available, which is shown in table 2. The finite state machine including allowed transitions is shown in figure 2.

Each EUDAQ2 instance reports the state *UNINIT* to the `Run Control` at startup. An issued *INIT* command by the `Run Control` triggers the initialization of an instance, which changes the state to *UNCONF*, if no errors occur. A subsequent successful execution of the *CONFIG* command leads to a reported *CONF* state. If all instances are in a *CONF* state, the `Run Control` is able to *START* the readout, changing the EUDAQ2 state to *RUNNING*. The *STOP* command stops the data taking and the EUDAQ2 instance keeps waiting for a new run as soon as another *START* command arrives. In case a new configuration file needs to be used and distributed to the `Producers`, a *CONFIG* command needs to be executed between the *STOP* and the next *START*. In case of an unexpected error, the state *ERROR* is reported. The only way to recover from an *ERROR* state is to

Table 2. The EUDAQ2 commands to trigger a change between EUDAQ2 States.

Command	State before command	State after command	Command Description
INIT	*UNINIT*	*UNCONF*	Initialize using the initialization file
CONFIG	*UNCONF*/*CONF*	*CONF*	Configure using the configuration file
START	*CONF*	*RUNNING*	Start up a new run
STOP	*RUNNING*	*CONF*	Stop the current run
RESET	*ERROR*/*CONF*/*UNCONF*	*UNINIT*	Reset all running components
TERM	all except *RUNNING*		Terminate EUDAQ2

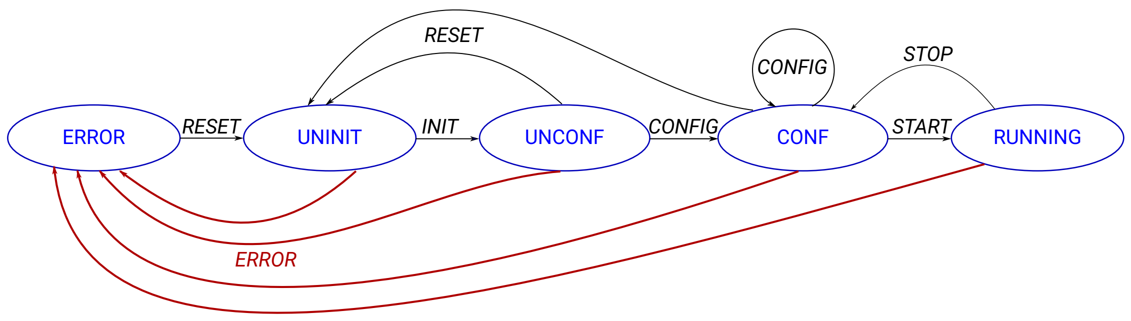


Figure 2. Finite State Machine of EUDAQ2. States are displayed in blue, *commands* in black and error handling in red. For the successful initialization and configuration of EUDAQ2, valid initialization and configuration files need to be provided.

execute the *RESET* command which can reset the system to *UNINIT* state. Errors are handled by an exception mechanism. The *TERM* command terminates the complete system, closing all EUDAQ2 instances. It is worth noting that the *INIT* command can only be executed once while the *CONFIG* command can be issued several times.

It is the task of the `Run Control` instance to issue the commands to alter the states of all participating instances and check/validate the return codes of all instances. If an error occurs during the processing of a command by a `Producer`, e.g. during the configuration of the underlying detector hardware, an exception can be issued, which is caught and automatically converted to an error state. The error state is part of the return information collected by `Run Control`. The `LogCollector` receives a detailed error report automatically. Commands pushed from `Run Control` to each `Producer` are processed in the sequence they were received. a request to retrieve the current state from all the connected EUDAQ2 instances is sent periodically and the state is updated accordingly, hence providing a heartbeat of the entire system.

The configuration of the EUDAQ2 data acquisition framework is performed via two global configuration files. One of them is used by the *INIT* command, and the second for the *CONFIG* command. Both configuration files are stored as plain text and are divided into named sections for the individual run-time instance. Each configuration section, named by combination of role and run time name of the running destination instance, contains a list of parameter-value pairs which can be either mandatory or optional.

Table 3. EUDAQ2 data model.

Variable	Provided by User	Set by EUDAQ2 automatically
EventType	no	yes
RunNumber	no	yes
EventNumber	no	yes
Timestamp	optional	no
TriggerNumber	optional	no
RawDataBlock	yes	no
Tags	optional	no
SubEvents	optional	no

2.3 Data model

Data objects are sent between various EUDAQ2 instances in the same way as State and Commands. The data objects therefore need the capability to be serialized. When a data object is serialized, all the crucial data of this data object is fed to a serialized memory section which then can be sent as a plain binary data stream to another application and reconstructed as a copy of the original data object. Technically, the EUDAQ2 native data file format is a collection of serialized binary data streams from data objects

In EUDAQ2, the *Serializable* class is implemented by the *Event* class. The *Event* class stores data from a detector. The basic EUDAQ2 *Event* implementation provides a few general variables as the *Event* object is not required to be aware of all the details about each individual detector. But it is mandatory for each *Event* Object to have a few parameters set properly. Table 3 lists all member variables of a *Serializable Event* object and the members which EUDAQ2 sets automatically.

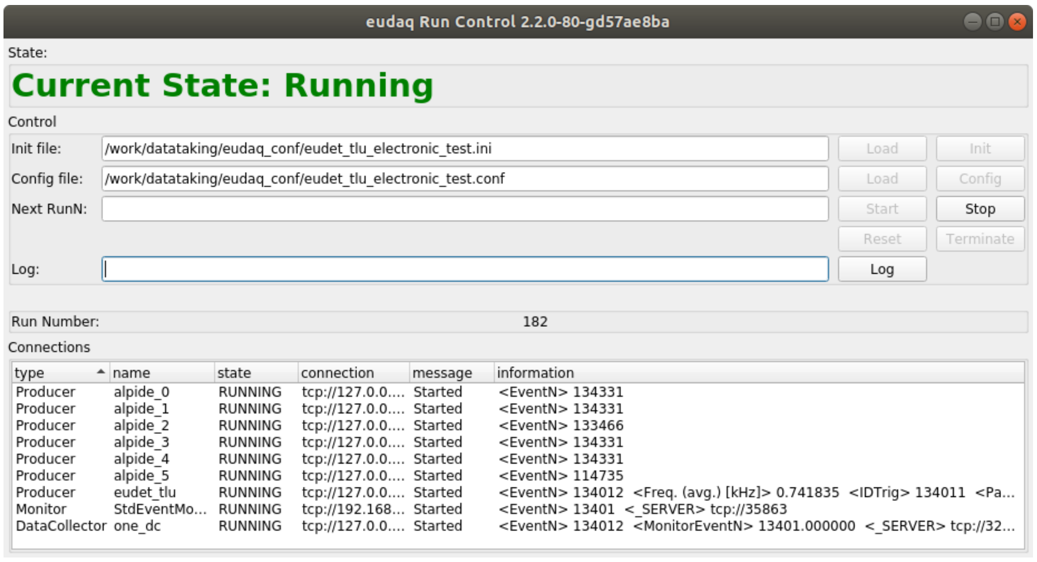
The *RawDataBlock* contains the corresponding detector data stored in a detector-specific format, encoded as an array of `uint8t`. It is the responsibility of each user to provide the necessary data decoders using a `DataConverter`. A pair of timestamps defines the start and end time of the *RawDataBlock* and a trigger number identifies the trigger sequence defined by the hardware setup. The timestamps and trigger number are per se optional, but are necessary to be set if they are to be used to synchronize data from multiple data streams. In this case, the raw measurement data has to be partly processed and decoded since the timestamps and trigger numbers are part of the detector raw data. However, to minimize the CPU consumption due to data processing by the `Producer`, full decoding of the detector raw data can be postponed to the online monitoring stage or the offline analysis. It is possible to encapsulate several events inside an *Event* object, the so-called *SubEvent* objects. This can be the case if the detector hardware controls multiple subsystems or has a cascaded data structure. The *SubEvent* objects are stored in the parent *Event* object and are therefore also *Serializable*.

2.4 Network communication

The distributed communication is running using only TCP/IP with a custom protocol, based on the developments for EUDAQ. Replacing the custom protocol with a more modern, performance optimized network protocol, allowing for higher data throughput and restoring lost packages has been investigated, but not yet implemented. This can be achieved with the existing abstraction layer and can be implemented as an extension library, without influencing any user code, see section 3.2. We foresee this for one of the future EUDAQ2 releases depending on user needs.

2.5 The graphical user interfaces

EUDAQ2 provides Graphical user interfaces (GUIs) based on the Qt framework [8], which is a free and open-source widget toolkit to create GUIs for cross-platform applications. The Qt-based versions of the EUDAQ2 `Run Control` (see figure 3), `LogCollector` and a `Monitor` are available as part of the EUDAQ2 package together with a command line version.



State:
Current State: Running
Control
Init file: /work/datataking/eudaq_conf/eudet_tlu_electronic_test.ini
Config file: /work/datataking/eudaq_conf/eudet_tlu_electronic_test.conf
Next RunN:
Log:
Run Number: 182
Connections

type	name	state	connection	message	information
Producer	alpide_0	RUNNING	tcp://127.0.0....	Started	<EventN> 134331
Producer	alpide_1	RUNNING	tcp://127.0.0....	Started	<EventN> 134331
Producer	alpide_2	RUNNING	tcp://127.0.0....	Started	<EventN> 133466
Producer	alpide_3	RUNNING	tcp://127.0.0....	Started	<EventN> 134331
Producer	alpide_4	RUNNING	tcp://127.0.0....	Started	<EventN> 134331
Producer	alpide_5	RUNNING	tcp://127.0.0....	Started	<EventN> 114735
Producer	eudet_tlu	RUNNING	tcp://127.0.0....	Started	<EventN> 134012 <Freq. (avg.) [kHz]> 0.741835 <IDTrig> 134011 <Pa...
Monitor	StdEventMo...	RUNNING	tcp://192.168...	Started	<EventN> 13401 <_SERVER> tcp://35863
DataCollector	one_dc	RUNNING	tcp://127.0.0....	Started	<EventN> 134012 <MonitorEventN> 13401.000000 <_SERVER> tcp://32...

Figure 3. The Graphical user interface of the EUDAQ2 `Run Control`.

3 Integration with user code

To keep the pure C++ EUDAQ2 core clearly separated from all user code, which may depend on non-standard libraries, the EUDAQ2 binary library is split into a core library and optional module/extension libraries. Figure 4 shows this relation libraries with respect to the EUDAQ2 core library. A CMake configure file to support user code integration against the EUDAQ2 core is supplied.

EUDAQ2 is not compatible with EUDAQ, as the API has been changed and several interface methods have been deprecated. However, the migration to EUDAQ2 is straightforward and requires only minor changes in the user code, as the exception and error handling has changed. To avoid accidental misuse of EUDAQ user code in EUDAQ2, all interface functions have been renamed.

3.1 Modular plugins

Modular plugins are used to interact with the user hardware, the DAQ systems, or to adjust the EUDAQ2 core functionalities to meet certain user requirements. A typical EUDAQ2 modular

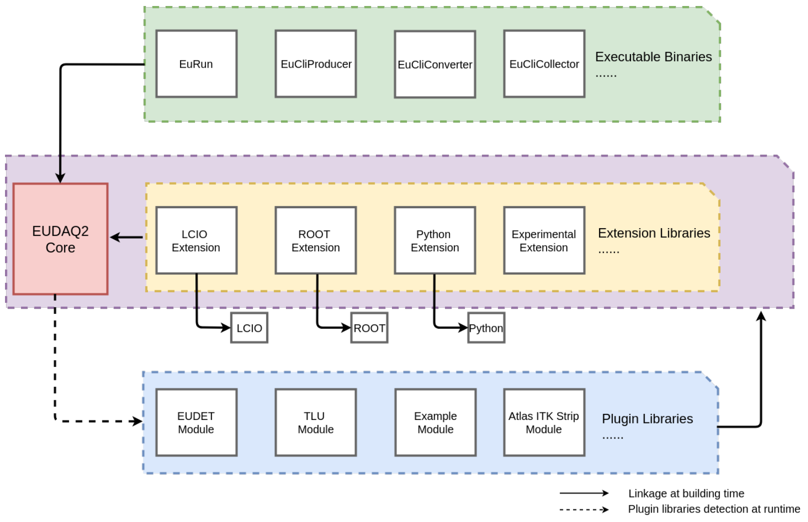


Figure 4. Layout of EUDAQ2, showing the relation between the EUDAQ2 core libraries and the Executables, Extensions and Plugins.

plugin contains a `Producer`, a `Data Collector` and specific hardware code provided by the user. Optionally a modified `Run Control` can also be defined as a module. External libraries or specific dependencies of a module have to be handled by the user by adjusting the module's CMake file. A shared library is created for each modular plugin at compile time. This library is loaded at the startup of EUDAQ2 run-time instances.

EUDAQ2 takes advantage of the Object Oriented Programming (OOP) [9] capabilities of C++ to create objects like `Producers` and `Data Collectors`, which inherit from the base classes by method dispatching. Using factories instead of constructors allows the usage of polymorphism for the object creation.

There are a few generic EUDAQ2 executables, so-called portals, which provide a generic way to create a user instance, e.g. a `Producer`. Examples for such portals are the `euCliProducer` and `euCliConverter` (see figure 4), which are generic utilities. After a EUDAQ2 portal executable has created an EUDAQ2 instance, the portal executable only interacts with the EUDAQ2 core library, while all user-specific functionality required by each instance is implemented in modular plugins. The EUDAQ2 core then takes care of searching and loading all available user plugin modules located in the specified directories. The communication between the EUDAQ2 core and its plugins is based on derived OOP classes. Therefore, the EUDAQ2 core can steer any user plugin binary without having knowledge of the hardware/plugin specific dependencies, both during build and during runtime. This encapsulation enables simple changes in the prototypes and class hierarchies.

3.2 Extension library

Supporting EUDAQ2 on multiple operation systems requires a compact core package without any external dependencies. However, this limits the available functionality, which can be an issue in certain use cases. To increase the flexibility, the core library is very modular with a minimal coupling between the major components. This allows for a simple replacement of individual components using extension libraries. An extension library provides additional optional core features, which might then require external dependencies. These extensions provide a convenient way to extend the functionality of EUDAQ2, without changing the core libraries themselves.

The data for example can only be stored as a raw and uncompressed serial-event object stream within the standard EUDAQ2 core. Extensions can be used to write out data using different forms: the LCIO [10, 11] extension library for example allows for writing data as LCIO-formatted data, which is widely used in the Linear Collider community. Developments of an extension to support data storage in ROOT `TTrees` [12, 13] is currently work in progress.

3.3 Python interface

Python has become a very popular programming language in the particle physics community. Therefore, a Python wrapper based on `pybind11` [14], which is a lightweight, header-only library that exposes C++ types to Python, is provided. Python and C++ have significant differences in the treatment of variables and memory: while Python passes variables as a reference and does automatic garbage collection, C++ requires manual memory management by default. With the introduction of C++11 [5] smart pointers, which provide memory management, have become available. Using smart pointers to store all the EUDAQ2 objects instead of raw pointers bridges the gap between Python and C++. However, a small performance penalty has to be paid using Python due the use of an interpreter and an additional layer, which encapsulates the C++ methods and converts the interfaces to Python. The Python interface enables a safe transfer of the *Event* and *Status* objects from C++ into Python or vice versa and a Python example is provided as a part of the EUDAQ2 package.

4 User examples

In most user cases, providing the dedicated and detector-specific user code and building them into EUDAQ2 as a modular plugin is sufficient. A modular plugin usually involves following parts:

1. Implementation of a set of `Producers` for each new detector.
2. Implementation of a `DataConverter` to convert each *RawEvent* to a *StdEvent* and to the LCIO format, if used for later analysis.
3. Optionally, implementation of a `Data Collector`, if a specific synchronization, other than trigger-ID or timestamp based methods, is required.

The EUDAQ2 manual [15] provides detailed technical information as well as a set of code examples.

EUDAQ2 contains many new features from long-standing requests of the user community that has used EUDAQ for nearly a decade. Several user groups have become early adopters already during the development phase of EUDAQ2 and provided very valuable feedback throughout and are now using EUDAQ2 for their test beam campaigns.

Already today, EUDAQ2 supports many test beam campaigns of detector prototypes at CERN and DESY. EUDAQ2 is becoming more popular and many users migrate their EUDAQ implementations to EUDAQ2. Some examples for the usage of EUDAQ2 are given below, showcasing the wide range of applications, ranging from the EUDET-type and Lycoris beam telescopes to test beams for HL-LHC and CALICE.

4.1 EUDET-type beam telescopes

As the origin of the EUDAQ framework, measurements based on EUDET-type beam telescopes are the most extensive application of EUDAQ2/EUDAQ software. Together with the EUDET-TLU [16] they compose a common Trigger-DAQ infrastructure provided by several facilities. Over the last decades, many different user setups have been integrated into EUDAQ in order to use one of the seven copies of EUDET-type beam telescopes located on five different beam lines [2]. In parallel, developments to achieve higher trigger rates in the Trigger-DAQ system of common beam telescopes [17] have started.

EUDET-type telescopes are Mimosa26-based [18, 19], which operate in a rolling shutter mode with a period of 115.2 µs. A single FPGA receives the data of six individual telescope planes and creates trigger based double sensors frames. Trigger and rolling shutter are not correlated. Thus, the next frame also needs to be read out resulting in a busy time of the telescope between 115.2 and 230.4 µs. A telescope event integrates all possible particle hits in the corresponding double frame read-out, increasing the potential track multiplicity. Using a faster pixel sensor, as for example a FE-I4 [20] based sensor with a 25 ns read-out time, allows identification of the track invoking the trigger by correlating the telescope track and the FE-I4 hit. Together with the global busy logic of the EUDET-TLU, the maximum rate for tracks with a high time resolution is about 4 kHz.

The development of EUDAQ2 and the implementation of the AIDA-TLU [21] allows for a new data taking mode which overcomes this trigger rate limit. With the EUDET-TLU, the system trigger rate has been limited by the slowest device. This issue is overcome with the AIDA mode, that provides more flexibility by enabling individual configuration for each connected device. By configuring an individual busy, faster devices can receive triggers while slower devices are still busy: a FE-I4 DAQ can potentially receive several trigger signal during the read-out time of the Mimosa26-DAQ.

The time information can be used to assign the multiple trigger information to potential multiple tracks in a Mimosa26 frame. Applying this to the telescope system a trigger rate of 115 kHz is possible, or a factor of ~ 28 of improvement, limited by the time required to read out the trigger ID. The trigger ID is used to synchronize the event streams in this data taking mode, by assigning unique numbers to triggers. The AIDA-TLU provides the same data-trigger-busy communication protocol as the EUDET-TLU to be compatible to existing device-under-test integration setups.

The EUDAQ components of the telescope were upgraded and complemented within the EUDAQ2 framework. These are `Producers` for the Mimosa26 DAQ and for both TLU types which define also the choice of the `Data Collector`. Operating with the EUDET-TLU, the `EudetTluProducer` and the `EventIDSyncDataCollector` are used to synchronize the event streams by event number knowing that devices are operated in a trigger global busy scheme. Operating with the AIDA-TLU, the `AidaTluProducer` and the `TriggerIDSyncDataCollector` are used to synchronize event streams by trigger number which allows the new data taking mode as described above. Furthermore a generic `DirectSaveDataCollector` is provided which can be called multiple times, to store data from each connected `Producer`. This is an example of distributed and decentralized data taking for which the event synchronization happens offline. Therefore, exemplary executables are provided for synchronizing multiple data streams by event number or trigger ID offline. Finally, `DataConverter` modules can convert the TLU, FE-I4 and

2019 JINST 14 P10033

Mimosa26 *RawEvent* to *StdEvent* blocks for providing interpretable data to the `StdEventMonitor` which is the exported EUDAQ `OnlineMonitor`. Corresponding `DataConverter` modules are available to convert events to the LCIO format [11] in order to perform track reconstruction with the EUTelescope framework [22] as part of the EUDET-type telescope infrastructure.

4.2 ATLAS ITk strip

The ATLAS Inner Tracker (ITk) is a planned silicon tracker which is foreseen to start operation in 2026. It comprises an inner section consisting of pixel sensors, and an outer section consisting of strip sensors. The latter will cover an active area of approximately 165 m^2 with 17888 modules. Each module is composed by a silicon sensor, front-end electronics and a power board.

Since 2017, five successful test beam campaigns with EUDAQ2 have been conducted. Two fully irradiated prototype end-cap module and the first double-sided prototype end-cap module were tested, among other devices [23]. The migration from EUDAQ to EUDAQ2 did not require extensive modifications to the previous setup.

As in EUDAQ, two separate `Producers` are used for the ATLAS ITk DAQ: a `Producer` transmitting data from the front-end chips, as the hit information, and a Producer providing TTC (Timing, Trigger and Control) information from the readout FPGA. A dedicated converter to the LCIO data format is implemented for each stream in order to perform track reconstruction and data analysis using the EUTelescope reconstruction framework.

4.3 KPiX strip telescope

The KPiX readout system is used by the Lycoris strip telescope [24] at the DESY II Test Beam Facility. Its native DAQ system consists of an ASIC called KPiX [25], and a FPGA DAQ board. The KPiX digitizes and serializes the collected data, the FPGA reads out the data from all the connected KPiX and transmits to the PC. The KPiX chip is designed to be power cycled in between data acquisition periods, which requires an external acquisition start signal. The chip has an internal calibration and trigger module, but it can work with a forced external trigger. Both the acquisition start signal and the external trigger can be sent from the DAQ software.

KPiX has its own DAQ software, which is the prototype version for the ROGUE [26] framework developed by SLAC. Given that KPiX DAQ has been designed with multiple threads to control data taking and software command transmission, the final integration to the EUDAQ2 was implemented with an intermediate FIFO queue. The KPiX user module consists of one `Producer` connecting KPiX via shared KPiX DAQ libraries, one `Data Collector` linked with KPiX binary data format libraries, two `DataConverter` units to interpret a *RawEvent* to the *StdEvent* and the LCIO format, and one `Run Control` to ensure Stop and Reset functioning for all run modes with KPiX.

Moreover, the KPiX user module contains a customized GUI inherited from the core GUI for printing more detailed information like the Run Rate, and one analysis executable called `lycorisCliConverter` for producing a set of straightforward plots like the ADC distribution of each channel to a ROOT file from an EUDAQ2 *RawEvent*.

This user module has been used for data taking in multiple test beam campaigns. Figure 5 shows the good spatial correlation between two Lycoris strip sensor modules taken with this EUDAQ2 module.

2019 JINST 14 P10033

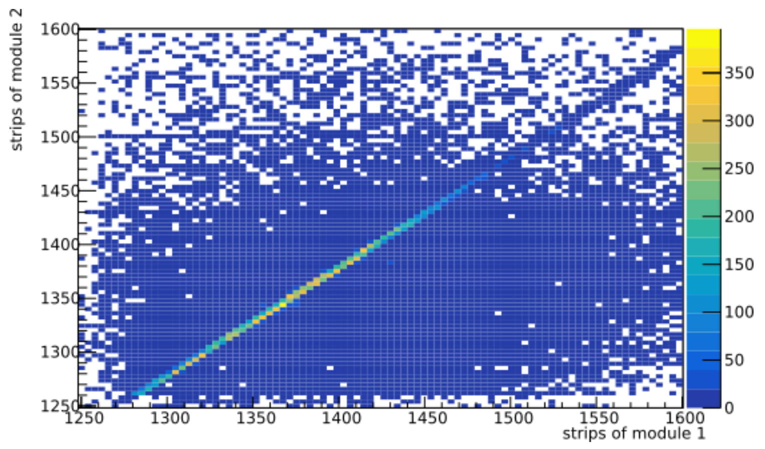


Figure 5. Spatial correlations of two Lycoris strip sensor modules from example data taken with the EUDAQ2 KPiX user module at DESY II Test Beam Facility.

4.4 CALICE AHCAL

The CALICE AHCAL is a prototype of a highly granular hadron calorimeter [27] optimized for the particle flow algorithm [28] at a future e^+e^-collider [29]. It uses $30 \times 30 \times 3\,\mathrm{mm}^3$ scintillator tiles, which are individually read out by silicon-photomultiplier photodetectors. The embedded, low-power readout electronics is based on the SPIROC chip [30] with 36 input channels, specifically designed for a power-pulsing operation in sparse spills with an active duty cycle of less than 1 %. The chip is self-triggered and stores for each channel a charge, a hit time and a bunch-crossing ID (BXID).

The AHCAL EUDAQ2 `Producer` includes an configurable event building method, producing events in various modes in order to be able to synchronize with other detectors:

Timestamp is recorded in AHCAL DAQ hardware with 25 ns resolution with the possibility to receive (or provide) the clock from (to) other systems. The timestamp is 48 bits wide and overflows only after 81 days.

Trigger number can be counted and timestamped from external source in the AHCAL DAQ hardware via a separate path. In the producer's event building stage, the trigger timestamp is then used to pair the trigger with the corresponding self-triggered hits from the ASIC data. The trigger ID is then used as an event number.

Acquisition cycle and BXID are unique identifiers, that can be used for synchronization with other acquisition-cycle oriented detectors, especially with front-end chips from the same family. The event can be build on a cycle level, containing all hits within the acquisition cycle (spill), or split into separate events by BXID.

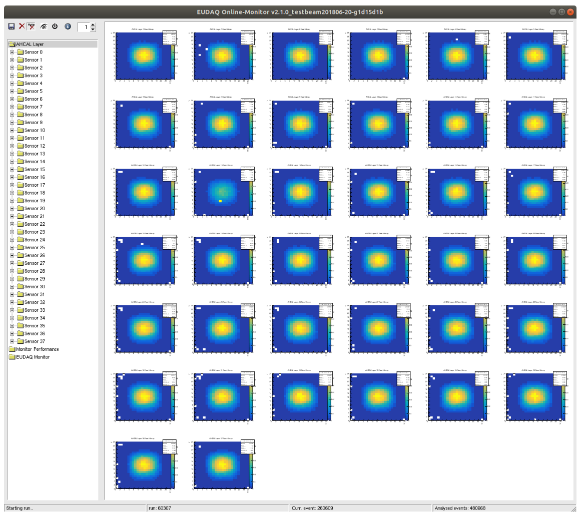
EUDAQ2 provides the possibility to run multiple data collectors. This feature was used extensively during the synchronization studies with the Mimosa beam telescope, where several instances of a data collector with different event number offsets ran together. The event number offset was found by observing the spatial correlation in generated files. The events are merged in EUDAQ by the event (trigger) number.

A data converter from AHCAL EUDAQ2 *RawEvent* to a *StdEvent* provides a possibility to display the AHCAL plane ($72 \times 72\,\mathrm{cm}^2$) as a pixel plane in the EUDAQ2 `OnlineMonitor`, giving

2019 JINST 14 P10033

an immediate beam footprint feedback, as shown at figure 6. The converter was used to check and monitor the spatial correlations with the Mimosa telescope at the DESY II Test Beam Facility using the `OnlineMonitor`.

The CALICE AHCAL has successfully used EUDAQ2 in many commissioning and test beam campaigns. A specific AHCAL `Run Control` automatically loading a new configuration before restart of a new run was used to perform internal calibration or automated position scans on the moving platform at the DESY II Test Beam Facility. Delay wire chambers were used at the CERN test beam for beam particle trajectory reference. Online event number synchronization (based on the trigger number) was not reliable and timestamps were not using the same time base. The synchronization was therefore achieved offline. The AHCAL used EUDAQ2 together with the CMS-HGCAL [31] at the CERN SPS test beam in 2018. The event synchronization was done offline, based on the trigger numbers and validated with the timestamps.



Starting run.. | run: 60307 | Curr. event: 260609 | Analysed events: 480668

Figure 6. Hit maps of the AHCAL with 38 layers in the 40 GeV muon beam at the CERN SPS, May 2018.

5 Conclusion

EUDAQ2 is a modular, modern and versatile data acquisition framework that has been developed as a successor of EUDAQ. A core library is utilized to manage the readout, data collection and steering. The core only depends on standard C++ functionalities, allowing for platform independent developments. Hardware specific code, a `Producer`, is linked to the core and can utilize more specific external libraries.

EUDAQ2 is shipped together with `Producers` for a trigger logic unit and the EUDET-type telescopes, which are provided as common hardware on test beam facilities at DESY, ELSA and CERN. The EUDAQ2 software, together with a detailed operation manual is available online under the LGPLv3 open-source licence. Already in the development phase, EUDAQ2 has been used successfully by several test beam groups. `Producers` to integrate pixel and strip sensors as well a highly granular calorimeter prototypes have been implemented. EUDAQ2 is in a unique position to repeat the success story of the first EUDAQ.

Acknowledgments

This project has received funding from the European Union's Horizon 2020 Research and Innovation programme under Grant Agreement no. 654168. This work was in part supported by the Science and Technologies Facilities Council, U.K.. Measurements leading to these results have been performed at the Test Beam Facility at DESY Hamburg (Germany), a member of the Helmholtz Association (HGF).

References

[1] H. Jansen et al., *Performance of the EUDET-type beam telescopes*, *Eur. Phys. J. Tech. Instr.* **3** (2016) 7.

[2] P. Ahlburg et al., *EUDAQ – A Data Acquisition Software Framework for Common Beam Telescopes*, arXiv:1909.13725.

[3] *EUDAQ2 github repository*, accessed: June 2019 [http://github.com/eudaq/eudaq].

[4] Free Software Foundation, *GNU Lesser General Public License*, version 3, 29 June 2007, accessed: 13 June 2019 [https://www.gnu.org/licenses/lgpl-3.0.html].

[5] ISO, *ISO/IEC 14882:2011 Information technology — Programming languages — C++*, International Organization for Standardization, Geneva, Switzerland (2012) [https://www.iso.org/standard/50372.html].

[6] *IEEE Standard for Information Technology-Portable Operating System Interface (POSIX(R)) Base Specifications, Issue 7*, *IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008)* (2018) 1.

[7] Kitware, Inc., *CMake*, accessed: April 2019 [http://cmake.org].

[8] The Qt Group, *Qt*, accessed: May 2019 [http://www.qt.io].

[9] E. Kindler and I. Krivy, *Object-oriented simulation of systems with sophisticated control*, *Int J. General Systems* **40** (2011) 313.

[10] F. Gaede, T. Behnke, N. Graf and T. Johnson, *LCIO: A Persistency framework for linear collider simulation studies*, *eConf* **C 0303241** (2003) TUKT001 [physics/0306114].

[11] S. Aplin et al., *LCIO: A persistency Framework and Event Data Model for HEP*, *IEEE Nucl. Sci. Symp. Med. Imag. Conf. Rec.* (2012) 2075.

[12] R. Brun and F. Rademakers, *ROOT: An object oriented data analysis framework*, *Nucl. Instrum. Meth.* **A 389** (1997) 81.

[13] I. Antcheva et al., *ROOT: A C++ framework for petabyte data storage, statistical analysis and visualization*, *Comput. Phys. Commun.* **180** (2009) 2499 [arXiv:1508.07749].

[14] W. Jakob and J. Rhinelander and D. Moldovan, *pybin11, a lightweight header library that exposes C++ types in Python and vice versa*, accessed: 28 June 2019 [https://pybind11.readthedocs.io/en/master].

[15] Y. Liu, *EUDAQ2 User Manual*, AIDA-2020-NOTE-2018-001.

[16] D. Cussans, *Description of the JRA1 Trigger Logic Unit (TLU), v0.2c*, EUDET-Memo-2009-04 [https://www.eudet.org/e26/e28/e42441/e57298/EUDET-MEMO-2009-04.pdf].

[17] U. Behrens et al., *Evolving the DAQ and Analysis Software of the AIDA Telescope: Toward high rates and one-trigger-per-particle Operation*, AIDA-NOTE-2015-010.

[18] J. Baudot et al., *First test results of MIMOSA-26, a fast CMOS sensor with integrated zero suppression and digitized output*, *IEEE Nucl. Sci. Symp. Conf. Rec.* (2009) 1169.

[19] C. Hu-Guo et al., *First reticule size MAPS with digital output and integrated zero suppression for the EUDET-JRA1 beam telescope*, *Nucl. Instrum. Meth.* **A 623** (2010) 480.

[20] M. Garcia-Sciveres et al., *The FE-I4 pixel readout integrated circuit*, *Nucl. Instrum. Meth.* **A 636** (2011) S155.

[21] P. Baesso, D. Cussans and J. Goldstein, *The AIDA-2020 TLU: a flexible trigger logic unit for test beam facilities*, 2019 *JINST* **14** P09019.

[22] T. Bisanz et al., *EUTelescope: A modular reconstruction framework for beam telescope data*, prepared for submission to *JINST*.

[23] L. Wiik-Fuchs et al., *First double-sided end-cap strip module for the atlas high-luminosity upgrade*, PoS(TWEPP2018)015 (2019).

[24] M. Wu, *Silicon strip reference tracker at DESY*, AIDA-2020-D15.2.

[25] J. Brau et al., *KPiX — A 1,024 channel readout ASIC for the ILC*, *IEEE Nucl. Sci. Symp. Med. Imag. Conf. Rec.* (2012) 1857.

[26] ROGUE Software Developers, *ROGUE documentation*, accessed: April 2019 [http://slaclab.github.io/rogue/].

[27] CALICE collaboration, *A highly granular SiPM-on-tile calorimeter prototype*, *J. Phys. Conf. Ser.* **1162** (2019) 012012 [arXiv:1808.09281].

[28] M.A. Thomson, *Particle Flow Calorimetry and the PandoraPFA Algorithm*, *Nucl. Instrum. Meth.* **A 611** (2009) 25 [arXiv:0907.3577].

[29] T. Behnke et al., *The International Linear Collider Technical Design Report — Volume 1: Executive Summary*, arXiv:1306.6327.

[30] S. Conforti Di Lorenzo et al., *SPIROC: Design and performances of a dedicated very front-end electronics for an ILC Analog Hadronic CALorimeter (AHCAL) prototype with SiPM read-out*, 2013 *JINST* **8** C01027.

[31] CMS collaboration, *The Phase-2 Upgrade of the CMS Endcap Calorimeter*, CERN-LHCC-2017-023.