EUDAQ

AIDA Final Meeting

Ulf Behrens, Alan Campbell, Francesco Crescioli, David Cussans,
Hendrik Jansen, Moritz Kiehn, Hanno Perrey, Richard Peschke,
Igor Rubinskiy, Simon Spannagel

HELMHOLTZ
ASSOCIATION

AIDA

DESY

Integration of user DAQs to EUDAQ

Why?

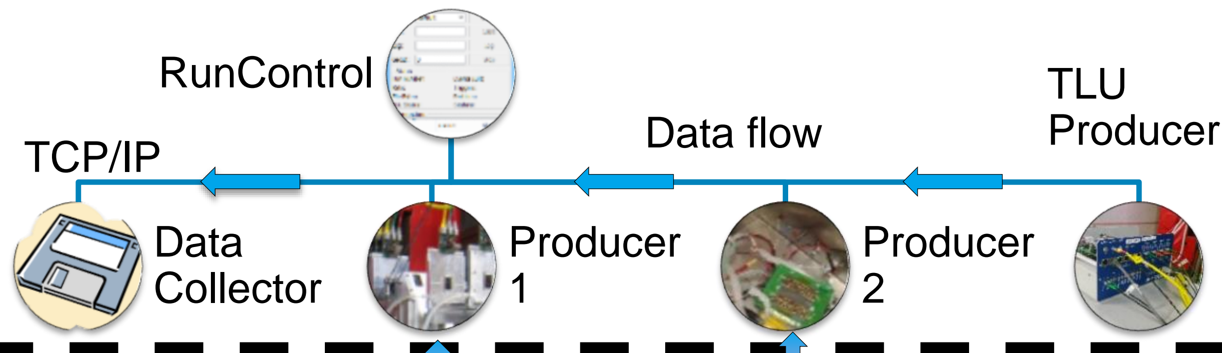
- Centralized RunControl
- Sanity checks
  - Check for event number mismatch
  - Missing timestamps
- Online monitoring
  - Correlation plots

How?

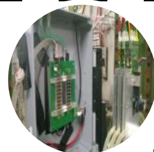
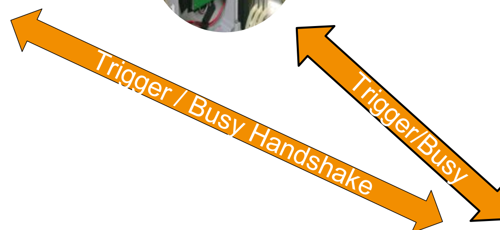
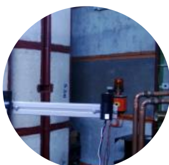
- Clear interface between user code and EUDAQ through "Producer" and "Data Converter Plugins"
- The "Producer" is a C++ class which handles the communication with EUDAQ
- Interfaces to other languages
  - Python
  - ROOT
  - LabView interface planed

EUDAQ

Software Layout



Read Out

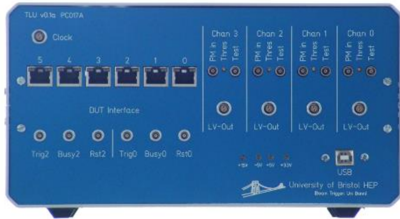




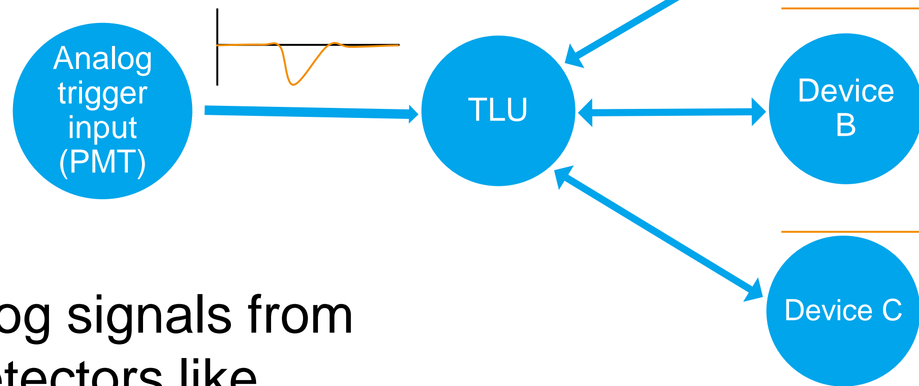
Hardware Layout

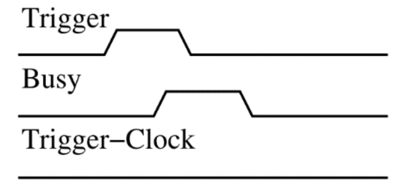
- Distributed DAQ system
  - Consists of completely independent parts / programs such as:
    - Run Control
    - Data Collector
    - Producers
- Synchronous read out
  - All devices get a common trigger
  - Every device raises a busy signal
- DAQ for MIMOSA telescope

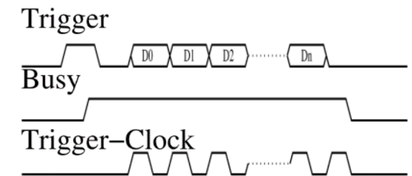
DESY

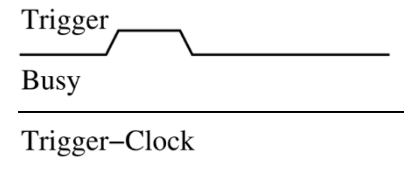
Trigger Logic Unit (TLU)











- Receives analog signals from fast particle detectors like scintillators with PMTs
- Converts the analog signal to a digital trigger
- Sends trigger to the devices
- Waits for the devices to finish their data taking

- Three types of handshakes
  - **Trigger/Busy Handshake:** Device sends a busy signal back. No new triggers can be issued during this time
  - **Trigger/Busy/Trigger-Data Handshake:** The TLU can send the TLU Trigger ID to the devices
  - **No handshake mode:** TLU just sends trigger without waiting for a response from the device

David Cussans

The Producer

Producer Started

Connecting to RunControl

Unconfigured

RunControl sends configuration

Configured

Re-Configuring

Start run

Stop run

Running

Readout loop

DataCollector

Readout

Trigger

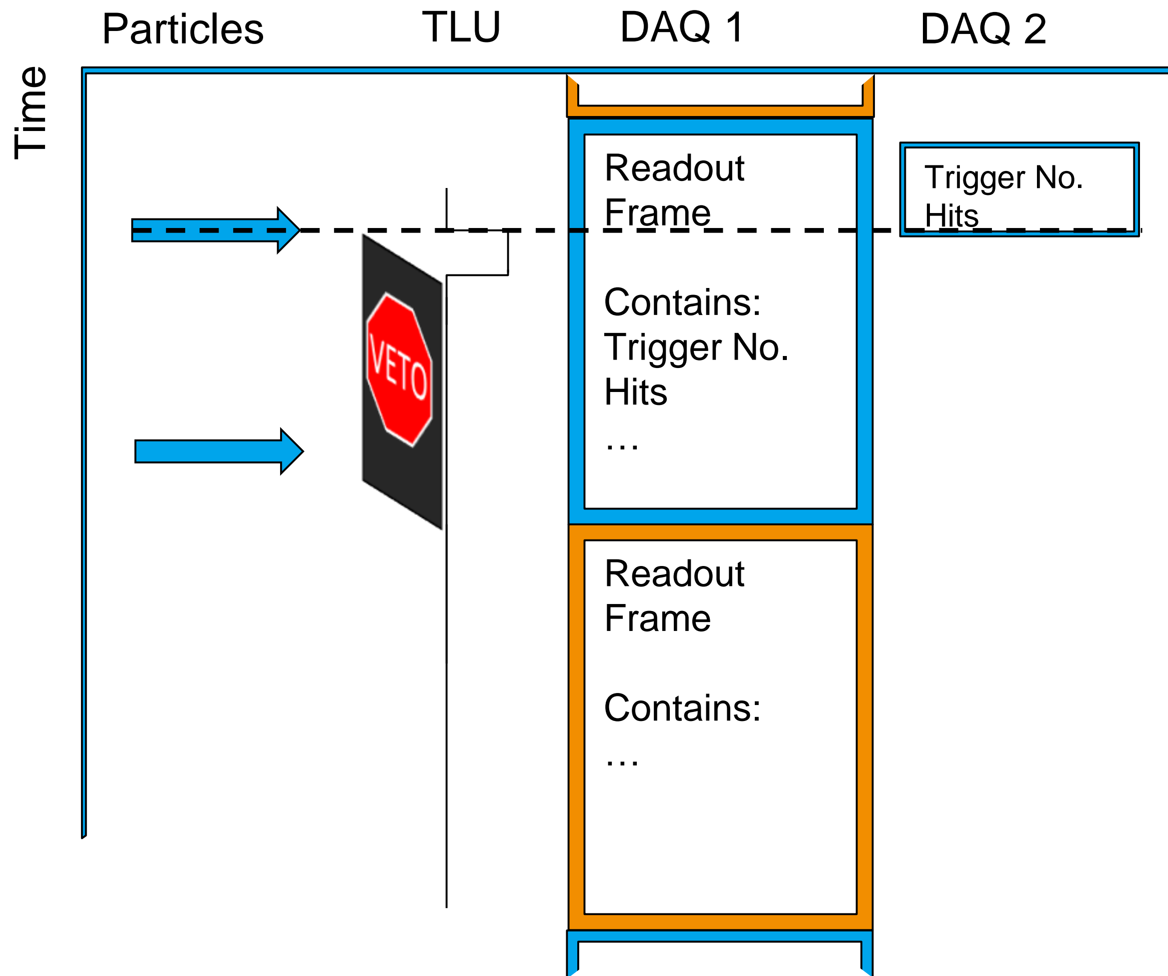
TLU

Device

The Producer as a State Machine

- 3 States
  - Unconfigured
  - Configured
  - Running
- 4 Transitions
  - Configure
  - Start run
  - Stop run
  - Terminate

EUDAQ 1.x: A Trigger Based DAQ System



- One trigger per read out frame
- Prevents the issuing of triggers for the whole time of the read out
- All but the first particle are ignored
- Online event building
- Slowest device limits the Event rate

Marked for write to Disk
Not Marked

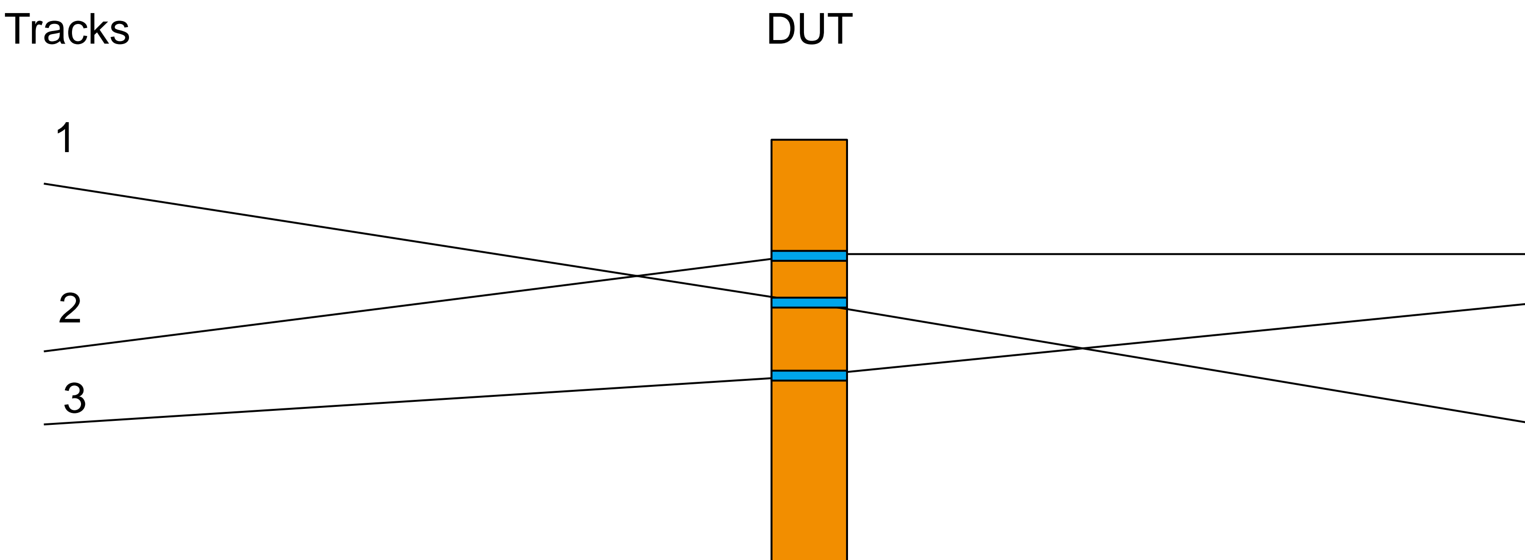
DESY

EUDAQ 2.0

- More flexibility in the data format
  - Possibility to store multiple readout frames in one “packet”
- Increasing of the track rate by more than 2 orders of magnitude
- Easier combination of different kind of devices
  - FE-I4 / Mimosa / Timepix(3) / Slow Control
- Resolved scalability issues
  - Decentralized data taking
  - Data can be stored locally without network overhead
- Stay backward compatible to old EUDAQ 1.x Producer and analyze readout chain (EUTelescope)
  - Only recompiling is needed
  - Changes are only need to benefit from some new features
- Cross platform

Definition: Read Out Frame (ROF)

- Read Out Frame is the finest granularity one gets from a device
- It does not mean the granularity with which one reads out the Device.
  - As an Example: With every read out from the TLU one gets the information from multiple triggers. Since one can disentangle the information from the individual trigger the Read Out Frame is the individual Trigger!
  - For the FE-I4 one ROF is one LHC bunch crossing which covers a time of 25 ns
  - For the Mimosa one ROF is one frame which covers a time of 115 µs

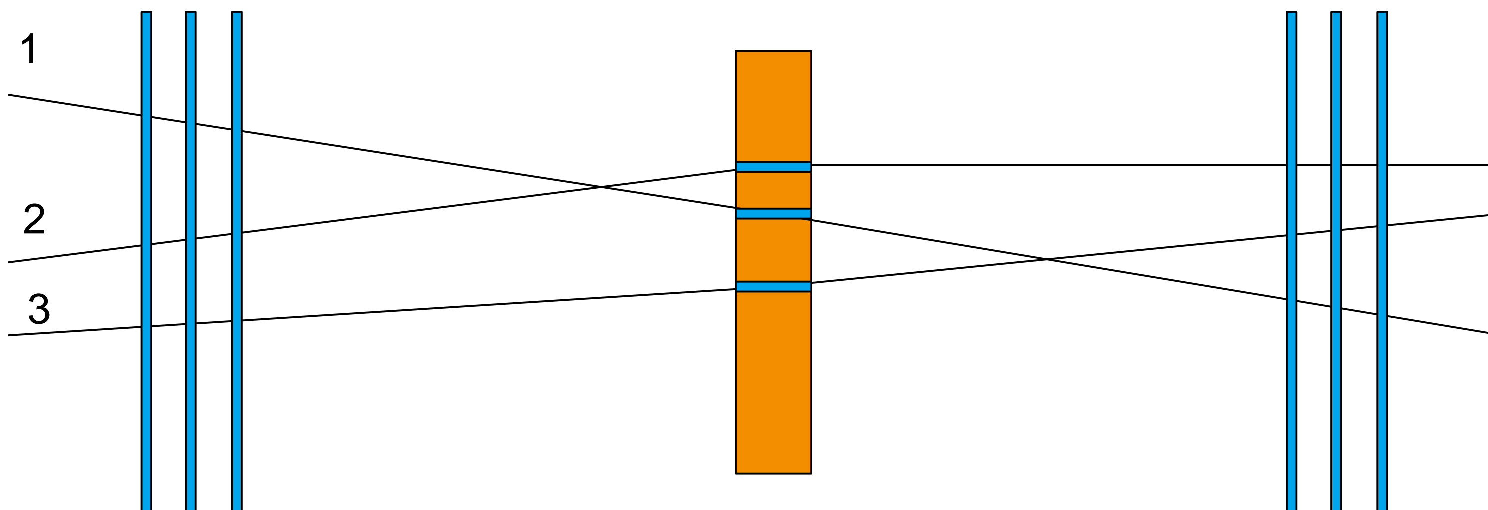


Information Needed:

- Space and time information at the position of the DUT
  - Space resolution < 3 µs
  - Separation time* < 30 ns
  - Time Resolution < 4 ns

*Tracks can only be separated if their time difference is larger than the separation time

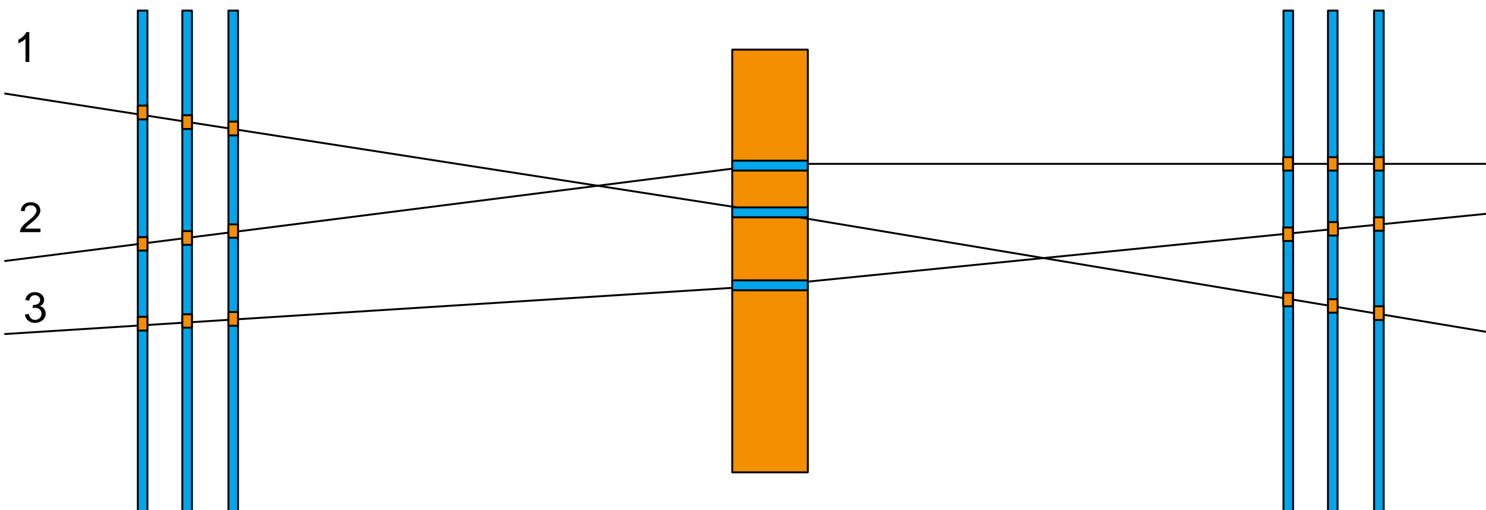
DESY

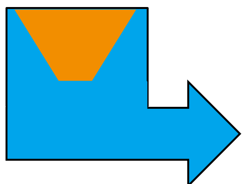


Information Needed:

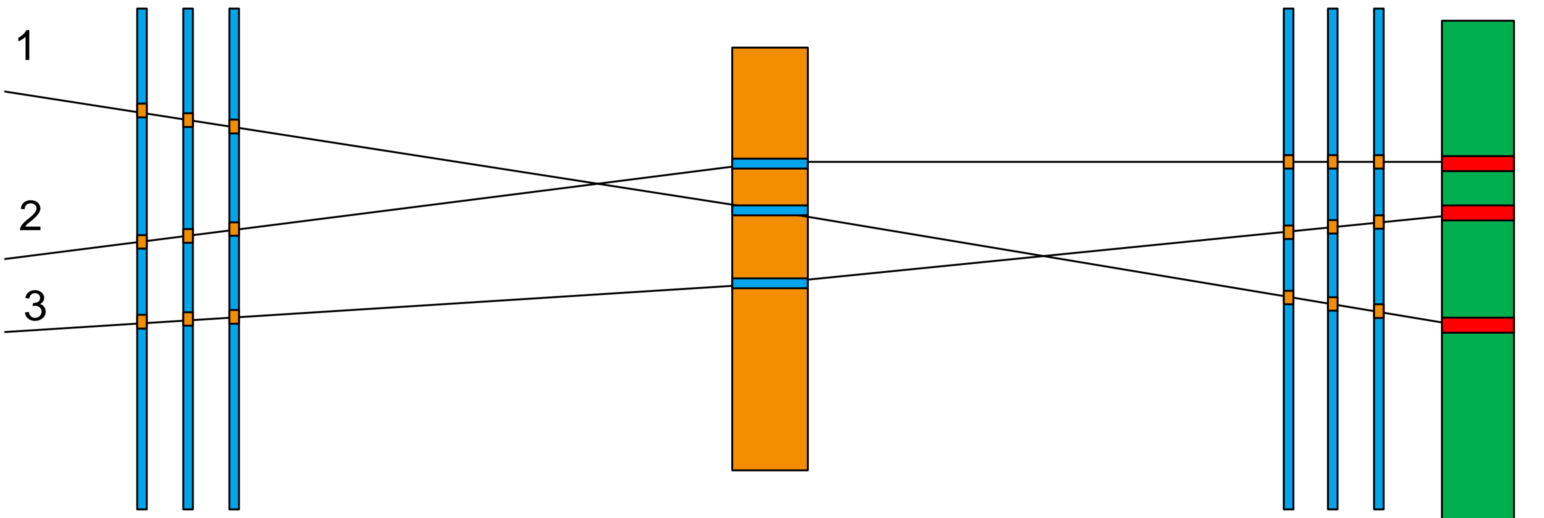
- Space and time information at the position of the DUT
  - Space resolution < 3 µs
  - Separation time* < 30 ns
  - Time Resolution < 4 ns

*Tracks can only be separated if their time difference is larger than the separation time





+ Intrinsic resolution of ~3.5 µm precession
- Very poor time resolution (115.2 µs)



+ Intrinsic resolution of ~3.5 µm precession
- Very poor time resolution (115.2 µs)

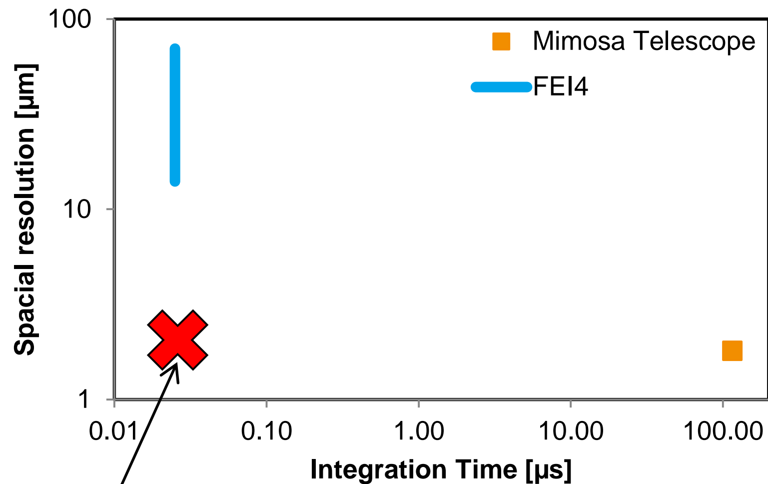
Solution adding faster device for time stamping the tracks

+ time resolution 25 ns

- Intrinsic resolution 14 x 70 µm

Asynchronous data streams

Combining of two types of sensors



- By adding the FE-I4 Detector the time resolution is increased by more than 3 orders of magnitude
- Gives the possibility to timestamp the individual Tracks

DESY

From Trigger to Timestamps

Particles | TLU | DAQ 1 | DAQ 2

Time

Readout Frame

Contains:
Timestamp 1

Hits
…

Timestamp 1
Hits

Readout Frame

Contains:
…

- Using all particles
  - Possible rate increase of two orders of magnitude
- No online event building
- Offline merging

Marked for write to Disk
Not Marked

From Trigger to Timestamps

Particles | TLU | DAQ 1 | DAQ 2

Time

Readout Frame

Contains:
Timestamp 1
Timestamp 2
Hits
…

Timestamp 1
Hits

Timestamp 2
Hits

Readout Frame

Contains:
…

- Using all particles
  - Possible rate increase of two orders of magnitude
- No online event building
- Offline merging

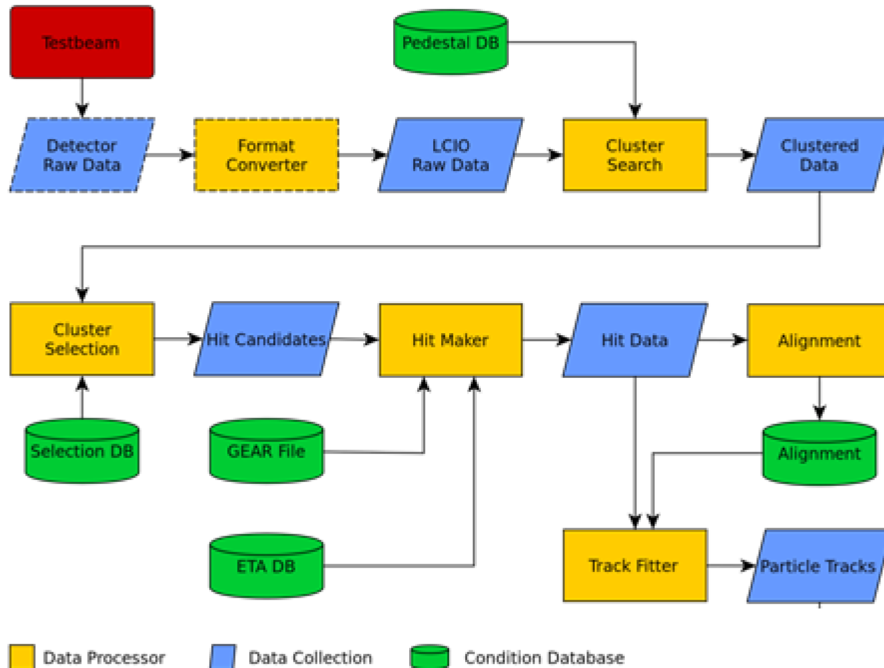
Marked for write to Disk
Not Marked

Merging

- Every Data Stream is stored separately
- The TLU is the central authority for merging
- Every event gets compared to the TLU events to find the corresponding TLU event
- The Compare algorithm can easily be modified by the users.
  It is part of the converter plugin
- Adding the TLU timestamp to the Event
- Processing events from different producers Individually until the tracks are extracted
- Merging happens on the level of tracks and not events

Merging

Current workflow

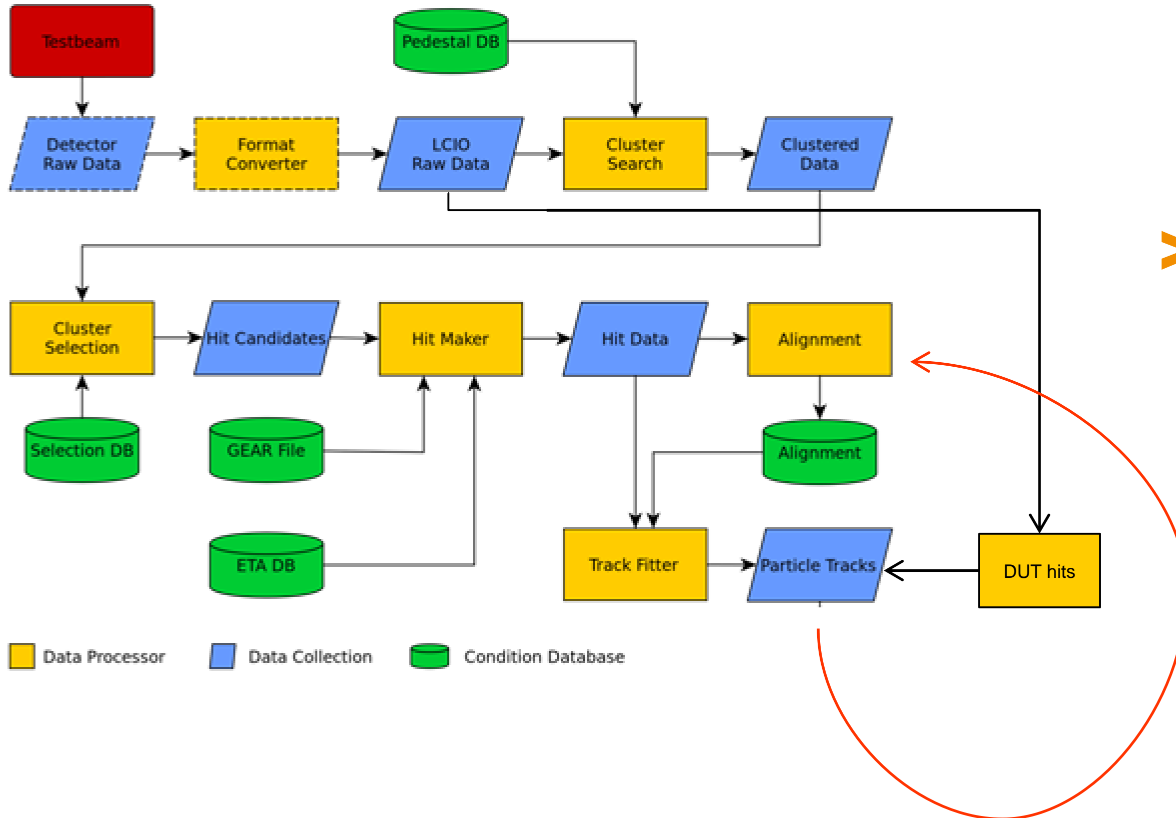


- The merging of the tracks with the DUT hits goes at the end.
- Reprocessing of alignment step after merging

DESY

Merging

Current workflow



- The merging of the tracks with the DUT hits goes at the end.
- Reprocessing of alignment step after merging

DESY

Packets

EUDAQ 1.x

- Every Producer sends one Event per readout frame
- For some devices this leads to a lot of overhead
- With low rate (~4 kHz) the overhead is not limiting the data rate

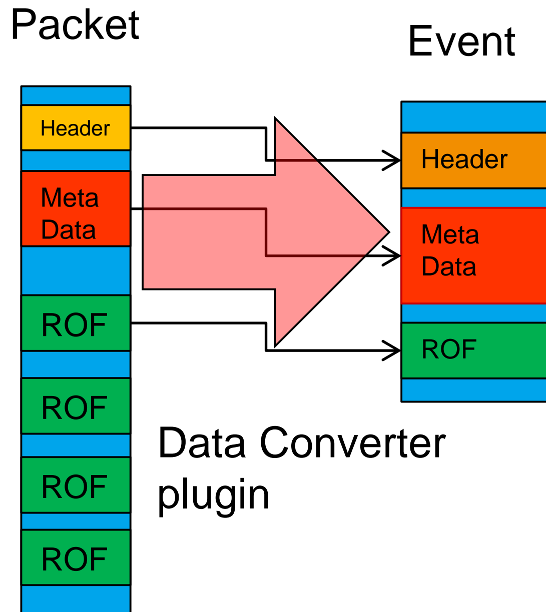
EUDAQ 2.0

- Desired rates:
  100 kHz -1 MHz
- The overhead from packing every ROF into one Event can Limit the data rate
- Allowing the use of Packet
- Packets can contain multiple ROFs.

→ No overhead for the individual ROFs

- ROFs are extracted Offline

How to extract Events from Packets



- Uses the well known mechanism of data converter plugins
- Complete flexibility how the users store their data

Summary

Accomplished

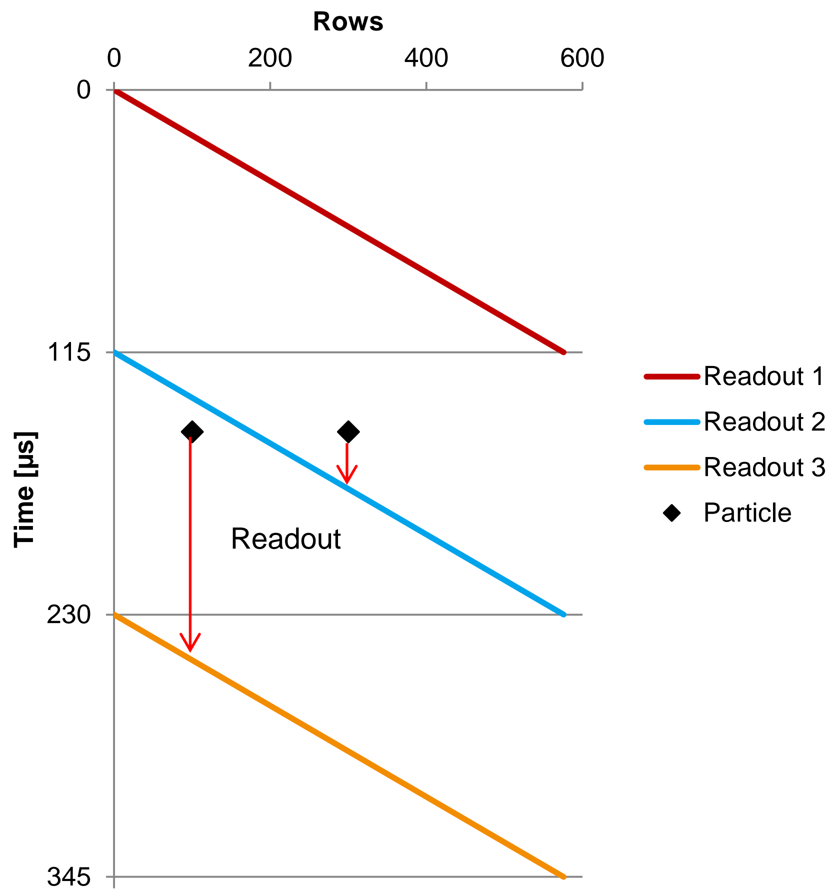
- New Data Format for Multiple Readout frames
- Possibility to Store Data locally to reduce the network overhead
- Merging for Online Monitor

Open Task

- Track merging in EUTelescope
- Extensive beam tests
- Users need to update their producer converter to take full advantage from EUDAQ 2

End of slide show, click to exit.

Mimosa Readout

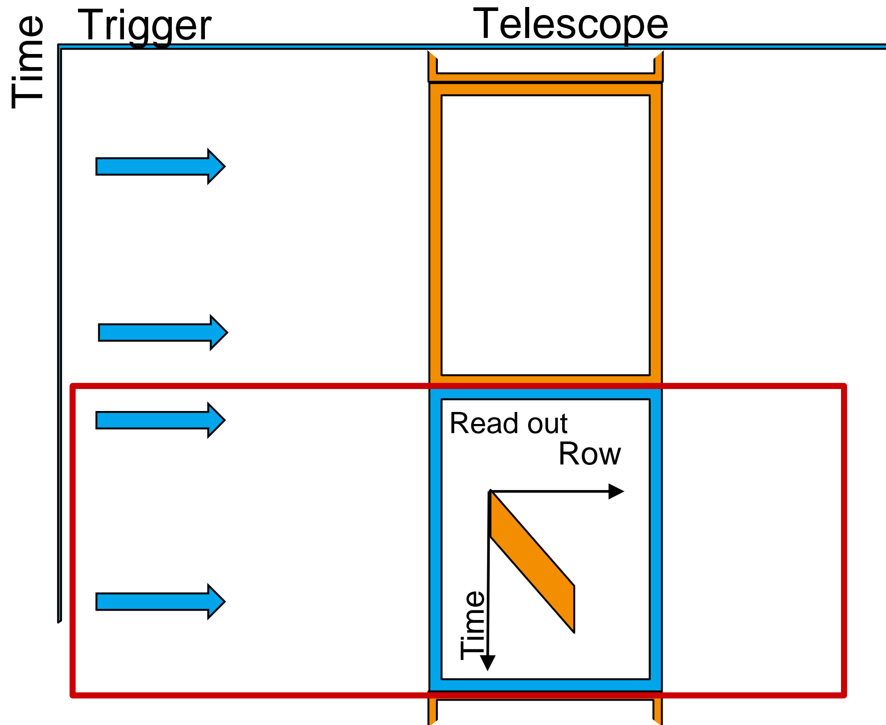


- Rolling shutter readout
- Readout row by row
  - The readout takes 115 µs
- Hits that that appear at the exact same time can be in two different readout frames

→ It is needed two associate two readout frames two one trigger

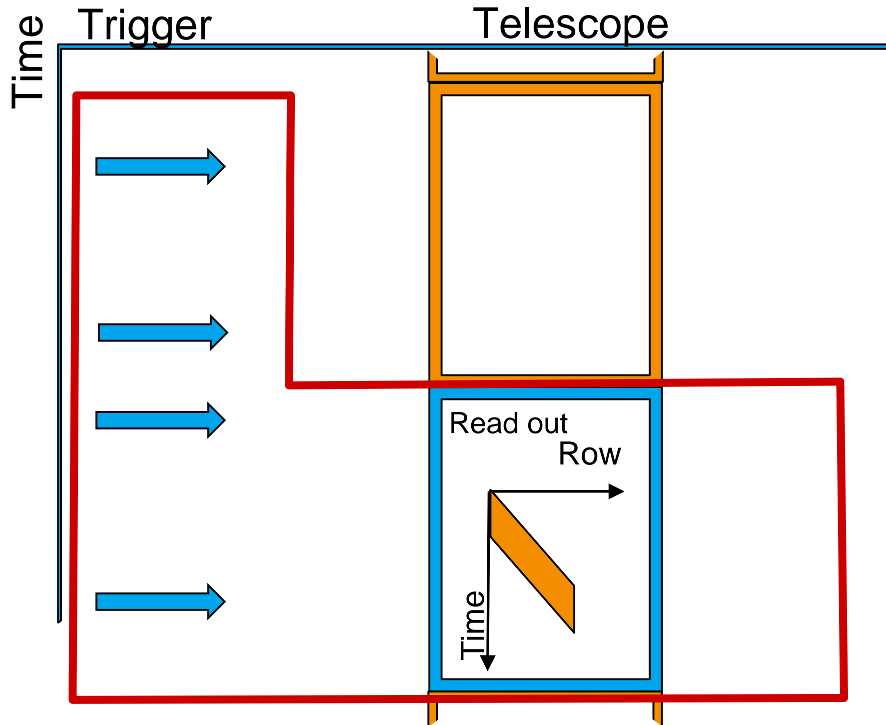
DESY

Telescope Type devices



- The information for one Trigger is split up on two ROF
- Limitation in the analyze framework prevents the use of references to previous events

Telescope Type devices



- The information for one Trigger is split up on two ROF
- Limitation in the analyze framework prevents the use of references to previous events
- Instead of associating only the trigger that happened during the ROF we also Associate the one from the previous ROF to this Event
- Trigger get associated to multiple ROF

DESY

Writing a Custom Producer

```
// Declare a new class that inherits from eudaq::Producer
class ExampleProducer : public eudaq::Producer {
  public:

    // The constructor must call the eudaq::Producer constructor with the name
    // and the runcontrol connection string, and initialize any member variables.
    ExampleProducer(const std::string & runcontrol)
      : eudaq::Producer("ExampleProducer", runcontrol) {}

    // This gets called whenever the DAQ is configured
    virtual void OnConfigure(const eudaq::Configuration & config);

    // This gets called whenever a new run is started
    // It receives the new run number as a parameter
    virtual void OnStartRun(unsigned RunNumber);

    // This gets called whenever a run is stopped
    virtual void OnStopRun();

    // This gets called when the Run Control is terminating,
    // we should also exit.
    virtual void OnTerminate();

    void ReadoutLoop();

  private:
    unsigned   m_runNumber;  //Variable to store the RunNumber
                             //Received from Runcontrol

    unsigned m_eventNumber;  //Counter to enumerate the Events
                             //starting from zero
};
```

Producer Name:

- Is displayed in RunControl
- Must be the same name as in the configuration file

- Users have to overload these functions to fit their needs.
- Functions get called asynchronously from RunControl

Readout loop for communication with hardware

eudaq\main\exe\src\ExampleProducer.cxx

Configuring a Producer

Configuration File

```
[Producer.TLU]
OrMask = 0
VetoMask = 0
AndMask = 15
DutMask = 1
TriggerInterval = 0
TrigRollover = 0
```



```
void ExampleProducer::OnConfigure(const eudaq::Configuration & config)
{
  // ...
  auto ExampleParameter = config.Get(ExampleTag, defaultValue);
  if (ConfigureHardware(ExampleParameter)==successful_configured)
  {
    SetStatus(eudaq::Status::LVL_OK, "Configured (" + config.Name() + ")");
  }
  else
  {
    SetStatus(eudaq::Status::LVL_ERROR, "Unable to configure hardware");
  }
}
```

The Configuration file

- Is sent by the RunControl
- Has sections for every Producer
- Contains tag-value pairs

The OnConfigure function

- Receives the producer specific section of the configuration file
- Extracts the configuration
- Has a possibility to report errors to the user

Starting and Stopping A Producer

```cpp
void ExampleProducer::OnStartRun(unsigned RunNumber)
{
  m_runNumber = RunNumber;
  // It must send a BORE to the Data Collector
  eudaq::RawDataEvent bore(eudaq::RawDataEvent::BORE(EVENT_TYPE, RunNumber));
  //..
  // Send the event to the Data Collector
  SendEvent(bore);

  startHardwareReadout();
  // At the end, set the status that will be displayed in the Run Control.
  SetStatus(eudaq::Status::LVL_OK, "Running");
}

void ExampleProducer::OnStopRun()
{
  // Set a flag to signal to the polling loop that the run is over
  stopHardwareReadout();
  // wait until all events have been read out from the hardware
  waitForPendingEvents();

  // Send an EORE after all the real events have been sent
  // You can also set tags on it (as with the BORE) if necessary
  SendEvent(eudaq::RawDataEvent::EORE(EVENT_TYPE, m_runNumber, ++m_eventNumber));
  SetStatus(eudaq::Status::LVL_OK, "Stopped");
}
```

The string only contains information additional information for the user

The status level is important for the mechanism.

- "onStartRun"
  - Receives the current run number
  - Has to send a Begin Of Run Event (BORE)
  - Starts the hardware readout
  - Sets the Producer status
- "onStopRun"
  - Stops the readout
  - Waits for the hardware readout to finish
  - Sends an End Of Run Event (EORE)
- After every call to one of the virtual functions the Producer status gets sent back to RunControl

Hardware interaction

Event Type

```
eudaq::RawDataEvent ev("ExampleProducer",
                       RunNumber,
                       EventNumber);
ev.SetTag("exampleTag", 123);
ev.setTimeStamp(Now);
ev.AddBlock(Identifier, VectorOfBinaryData);
SendEvent(ev);
```

A number to identify the data later in the data converter plugin

User Data in a serialized form

- The user has to check if the producer is in the correct state
- The user has to pack every readout frame in one event
- Events contain event Type, timestamps, tags and binary data
  - The event type is used to determine the correct data converter plugin
  - The event type must be the same as in the BORE and EORE
- The Producer has a public member function “SendEvent” to send events to the data collector

Python Producer

Producer Started

Connecting to RunControl

Unconfigured

RunControl sends configuration

Configuring

Re-Configuring

User has to set the status to configured

configured

Start run

Starting

User has to set the status to running

Stopping

Running

Stop run

- In opposite to the C++ producer the Python Producer doesn't have any virtual function to overload
- The transition is done by adding intermediate states
- The user has to check actively for the status of the producer

States changes from EUDAQ

States changes from the user

Python Producer

```
from PyEUDAQWrapper import * # load the ctypes wrapper
pp = PyProducer("testproducer","tcp://localhost:44000")
# wait for configure cmd from RunControl
while i<maxwait and not pp.Configuring:
    sleep(waittime)
    print "Waiting for configure for ",i*waittime," seconds"
    i+=1
# check if configuration received
if pp.Configuring:
    print "Ready to configure, received config string 'Parameter'=" \
                                        ,pp.GetConfigParameter("Parameter")
    # .... do your config stuff here ...
    sleep(5)
    pp.Configuring = True
# check for start of run cmd from RunControl
while i<maxwait and not pp.StartingRun:
    sleep(waittime)
    print "Waiting for run start for ",i*waittime," seconds"
    i+=1
# check if we are starting:
if pp.StartingRun:
    print "Ready to run!"
    # ... prepare your system for the immanent run start
    sleep(5)
    pp.StartingRun = True # set status and send BORE
# starting to run
while not pp.Error and not pp.StoppingRun and not pp.Terminating:
    # prepare an array of dim (1,3) for data storage
    data = numpy.ndarray(shape=[1,3], dtype=numpy.uint8)
    data[0] = ([123,456,999]) # add some (dummy) data
    pp.SendEvent(data)        # send event off
    sleep(2)                  # wait for a little while
# check if the run is stopping regularly
if pp.StoppingRun:
    pp.StoppingRun=True # set status and send EORE
```

The only access to the event
(one vector of c_unit8)

Limited Access to the Event Class

No Access to:

- Timestamps
- Tags
- Flags

Work around

> Using an external serializer
  - Protobuf

→ Introducing more external dependencies
  - not easy to maintain in a cross platform cross language environment

→ Eventually we have to provide full access to the events

Data Converter Plugin

User code for Reconstruction

```
// Declare a new class that inherits from DataConverterPlugin
class ExampleConverterPlugin : public DataConverterPlugin {

public:

  // This is called once at the beginning of each run.
  // You may extract information from the BORE and/or configuration
  // and store it in member variables to use during the decoding later.
  virtual void Initialize(const Event & bore,
    const Configuration & cnf);

  // Here, the data from the RawDataEvent is extracted into a StandardEvent.
  // The return value indicates whether the conversion was successful.
  // Again, this is just an example, adapted it for the actual data layout.
  virtual bool GetStandardSubEvent(StandardEvent & sev,
    const Event & ev) const;

private:

  // The constructor can be private, only one static instance is created
  // The DataConverterPlugin constructor must be passed the event type
  // in order to register this converter for the corresponding conversions
  // Member variables should also be initialized to default values here.
  ExampleConverterPlugin() : DataConverterPlugin(EVENT_TYPE){}

  // The single instance of this converter plugin
  static ExampleConverterPlugin m_instance;
}; // class ExampleConverterPlugin

// Instantiate the converter plugin instance
ExampleConverterPlugin ExampleConverterPlugin::m_instance;
```

This function is called at the beginning with the BORE event

Converts the raw event into an plane which contains pixel information

Singleton class. Gets created once at the program start

Event Type must be the same as on the Producer side

Example converter

```cpp
bool ExampleConverterPlugin::GetStandardSubEvent(StandardEvent & sev,
                                                  const Event & ev) const
{
  // If the event type is used for different sensors
  // they can be differentiated here
  std::string sensortype = "example";
  // Create a StandardPlane representing one sensor plane
  int id = 0;
  StandardPlane plane(id, EVENT_TYPE, sensortype);
  // Set the number of pixels
  int width = 100, height = 50;
  plane.SetSizeRaw(width, height);
  // Set the trigger ID
  plane.SetTLUEvent(GetTriggerID(ev));

  for (size_t i = 0; i < extract_size(ev);++i)
  {
    plane.PushPixel(
      extract_x_pos(ev, i),
      extract_y_pos(ev, i),
      extract_pixel(ev, i)
      );
  }

  // Add the plane to the StandardEvent
  sev.AddPlane(plane);
  // Indicate that data was successfully converted
  return true;
}
```

- In the data converter plugin user have the full access to the raw data
- Merge the pixel information into one common event which consist of individual plans for each detector
- Every plane represents one detector

Upcoming for EUDAQ 2.0

Asynchronous data Streams

- Every producer can send the data with its own speed
- Useful to combine devices with different integration time
  - MIMOSA / FE-I4 / Slow Control
- Recombination algorithm is part of the user code
  - User has to provide a function to compare there own events with the TLU events

Packets

- Contain the information of multiple Readout Frames (ROFs)
- Reduces network traffic
- Extraction of the ROFs is part of the user code (data converter plugin)