Writing a producer for EUDAQ Software

# Disclaimer

I am writing this documentary in parallel do writing my first producer therefore it can be that the ways of doing it is not the most advanced one.

# Starting point

For this tutorial it is expected that the reader has already red the “EUDAQ Software User Manual” from Emlyn Corrin.

Step 1: get the latest EUDAQ Version from Github

<https://github.com/eudaq/eudaq>

step 2: get

# The Example

## Different Approaches

It is expected that one has already s class that is representing the hardware and it has at least some primitive interface to interact with the data. In particular that means you have some dedicated methods to configure the Hardware. Some other function to start/stop runs. It is quite common that one doesn’t have any method to access individual events. It is more common that the event are just stored to files or processed otherwise. If you want to use the EUDAQ software it is crucial that you have a possibility to access the individual events. It is very likely that you have some “Main Loop” which gets the events at some point and processes them. Now you have in principle at least two possibilities:

* You can temporally store the data (or a pointer to the data or some internal reference number) somewhere. One possibility is that one stores the data in a queue that you can access from your “get Current event” method. The reason why you would do it this way is because you don’t to make too much changes to the readout class.
* On the other hand you could make your Hardware class a producer itself which can send the data directly from where they are produced. If you start a new project where the primary goal is to work with EUDAQ you should do this.

Since my hardware class is rather old and not very transparent I will go with the first approach.

## The Example Hardware

This example expect that you have already a class that is representing your hardware. Then you just add the following methods to it:

ExampleHardware.hh:

class ExampleHardware {

public:

ExampleHardware();

void Setup(int);

void PrepareForRun();

bool EventsPending() const;

unsigned NumSensors() const;

std::vector<unsigned char> ReadSensor(int sensorid);

void CompletedEvent();

private:

unsigned short m\_numsensors, m\_width, m\_height, m\_triggerid;

eudaq::Timer m\_timer;

double m\_nextevent;

};

1. void Setup(int);  
   in this function you can put everything that needs to be done before you can use your sensor the first time. Maybe some Startup script. Obviously you can change the signature from “int” to whatever you want.
2. Void PrepareForRun();   
   This function is kind of the Reset function. It is not called from the Example Producer but the Idea is that is it gets called before a new run is starting. Most likely your Hardware class has a method that starts the data taking process. This can either be Synchronous or asynchronous. If it is synchronous you have to call the data taking method asynchronous.
3. bool EventsPending() const;  
   this function indicates if there is an event in the queue or not. During run the Producer would just wait until a new event is produced. On the end of the run when the TLU is not anymore giving any signals it is expected that your Hardware can run until it has processed the last event. So the producer will run until there is no more event pending.
4. unsigned NumSensors() const;  
   gives the number of sensors …
5. std::vector<unsigned char> ReadSensor(int sensorid);  
   This function is the trickiest one because it read the information from the sensor and in the same time it converts them to a format that can be serialized and send over the network. The entire EUDAQ framework is depending on sending data as a vector of unsigned char over network. If you have some other format you need to convert same. The reconverting is done in your “Converter plugin” therefore it is very important that if you change one you have to change the other too.
6. void CompletedEvent();  
   after you are finished with reading you can give your hardware class some feedback

## The Example Producer

This is the definition of the Example Producer. It is about the easiest thing that can communicate with the run control. All the important things are already done in the producer base class so one can concentrate totally on implementing functionality.

ExampleProducer.cxx

// 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 & name, const std::string & runcontrol)

: eudaq::Producer(name, 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();

// This is just an example, adapt it to your hardware

void ReadoutLoop();

private:

// This is just a dummy class representing the hardware

// It here basically that the example code will compile

// but it also generates example raw data to help illustrate the decoder

eudaq::ExampleHardware hardware;

unsigned m\_run, m\_ev, m\_exampleparam;

bool stopping, done;

};

Let’s start with the constructor it is getting two parameters the first one is its own name the second one is the address of the run control. You can for example specify it as “tcp://localhost:44000” if it is the local host. You can also use some command line input to specify it, there is a good helper class already implemented that is very good in reading command line input: “eudaq::OptionParser”. However if you don’t have access to the command line input for whatever reason you can just hardcode it and it will run.

Next is the “OnConfiguration” method. It is called when the Run Control configures the producers. As an example we can see how it is working for the TLU. In the Configuration file is an extra section for the TLU starting with [Producer.TLU]. This shows the run control to send the Configuration to the TLU Producer. The TLU producer receives then only the information that are important for it.

[Producer.TLU]

AndMask = 0xf

OrMask = 0

VetoMask = 0

DutMask = 20

TriggerInterval = 0

TrigRollover = 0

In the “onConfiguration” method one gets this class “const eudaq::Configuration & param”. The class has a comfortable interface to access the configurations. The most important method is Get(key, default). The first argument “key” gives the name of the parameter, for example “AndMask”. The second one gives the default value of the parameter. The type of the default argument gives the type of the return value. You can also use the function with 3 argument:

std::string Get(key, fallback,default)

This function gives you the possibility to specify a second key. The first one is the one that is preferred and if this does not exist the fallback key is taken. If both do not exist the default value is taken.

Next method is onStartRun it gets as input parameter the run number that just started. It is important that you use this number otherwise the data collector will complain about wrong run number. Also you have reset the event counter. Then you have to send a BORE (begin of run event) as the first event. There is a factory class that produces these kind of events. And finally you have to set the producer status to running. You can take the code from the example producer as a frame for your producer:

// This gets called whenever a new run is started

// It receives the new run number as a parameter

virtual void OnStartRun(unsigned param) {

m\_run = param;

m\_ev = 0;

std::cout << "Start Run: " << m\_run << std::endl;

// It must send a BORE to the Data Collector

eudaq::RawDataEvent bore(eudaq::RawDataEvent::BORE(EVENT\_TYPE, m\_run));

// You can set tags on the BORE that will be saved in the data file

// and can be used later to help decoding

bore.SetTag("EXAMPLE", eudaq::to\_string(m\_exampleparam));

// Send the event to the Data Collector

SendEvent(bore);

// At the end, set the status that will be displayed in the Run Control.

SetStatus(eudaq::Status::LVL\_OK, "Running");

}

Next is the onStopRun. You can just copy it from the example producer. It is important that you set the some flag that indicates that you want to stop the run. Then you have to wait until the main loop has processed the last event. After this you need to send an EORE (end of run event).

// This gets called whenever a run is stopped

virtual void OnStopRun() {

std::cout << "Stopping Run" << std::endl;

// Set a flag to signal to the polling loop that the run is over

stopping = true;

// wait until all events have been read out from the hardware

while (stopping) {

eudaq::mSleep(20);

}

// 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("Test", m\_run, ++m\_ev));

}

What is left is the read out function. This function should continuously run as long as the producer is alive. In the example producer the loop is designed that it relies on the hardware to detect if it has an event or not. It is continuously reading out the sensor and in the moment the sensor has an event it will sent the event to the EUDAQ. It will continue running until there are no more events left and the “stopping” flag is set to true.

// This is just an example, adapt it to your hardware

void ReadoutLoop() {

// Loop until Run Control tells us to terminate

while (!done) {

if (!hardware.EventsPending()) {

// No events are pending, so check if the run is stopping

if (stopping) {

// if so, signal that there are no events left

stopping = false;

}

// Now sleep for a bit, to prevent chewing up all the CPU

eudaq::mSleep(20);

// Then restart the loop

continue;

}

// If we get here, there must be data to read out

// Create a RawDataEvent to contain the event data to be sent

eudaq::RawDataEvent ev(EVENT\_TYPE, m\_run, m\_ev);

for (unsigned plane = 0; plane < hardware.NumSensors(); ++plane) {

// Read out a block of raw data from the hardware

std::vector<unsigned char> buffer = hardware.ReadSensor(plane);

// Each data block has an ID that is used for ordering the planes later

// If there are multiple sensors, they should be numbered incrementally

// Add the block of raw data to the event

ev.AddBlock(plane, buffer);

}

hardware.CompletedEvent();

// Send the event to the Data Collector

SendEvent(ev);

// Now increment the event number

m\_ev++;

}

}

## The Example Converter Plugin

To run your producer in the EUDAQ you need to have a Converter Plugin. Otherwise the data collector will complain that he doesn’t understand the data type. As a first approach you can take the Example Converter Plugin and change the EVENT\_TYPE string to the name you gave your producer. Then it will run and collect the data. If you write a new plugin you don’t need to make it known to any other part of the program this is done automatically. The magic of it is, that the base class constructor (DataConverterPlugin) calls registers itself to the plugin manager. To achieve this, one has to create an instant of this class in the source file. This is done by having a static member of the converter plugin and instantiating it later in the code.

namespace eudaq {

class ExampleConverterPlugin : public DataConverterPlugin {

…

// The single instance of this converter plugin

static ExampleConverterPlugin m\_instance;

}; // class ExampleConverterPlugin

// Instantiate the converter plugin instance

ExampleConverterPlugin ExampleConverterPlugin::m\_instance;

} // namespace eudaq

Next function to implement is “unsigned GetTriggerID(Event & ev)” in principle the event should know witch trigger was corresponding therefore you can just return “ev->GetEventNumber()” it will return the event number from the event object. To go back to the Producer it will return the value you gave in the constructor of the Event “RawDataEvent ev(EVENT\_TYPE, m\_run, m\_ev);”.

Next function you need to implement is “bool GetStandardSubEvent(StandardEvent & sev, const Event & ev)”. Therefore it is good to make a small excurse. A standard event is a container and contains information from all telescope planes and the DUT. Each detector is just added as a new (sensor) plane to the Standard Event. The task of the GetStandardSubEvent function is now to convert the data from the event format to a “plane” class and add it to the “StandartEvent”. In principle you should never need to create the standard event yourself, because it should be created by some other part of the program. Also you should not change any settings of the standartEvent.

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);

// This should return the trigger ID (as provided by the TLU)

// if it was read out, otherwise it can either return (unsigned)-1,

// or be left undefined as there is already a default version.

virtual unsigned GetTriggerID(const Event & ev) const ;

// 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 ;

#if USE\_LCIO

// This is where the conversion to LCIO is done

virtual lcio::LCEvent \* GetLCIOEvent(const Event \* /\*ev\*/) const;

#endif

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();

// Information extracted in Initialize() can be stored here:

unsigned m\_exampleparam;

// The single instance of this converter plugin

static ExampleConverterPlugin m\_instance;

}; // class ExampleConverterPlugin

One thing that is missing is converting the data to LCIO files. It while be handle later.

For now the producer converter chain should work and you should be able to see your data.