Introduction to coding in diana

Stefano Pozzi

Data production bootcamp, Nov 16th 2020

Sequences, modules, events, objects

- Sequence: list of modules to be ran
- Module: performs a specific action on each event in the input files (e.g. calculates the amplitude of the pulse)
- Event: data related to a single triggered pulse/noise/..., progressively expanded by each module
- (Q)Object: each quantity associated to an event is written in a QObject

A module directly interacts with the QObjects stored in the input files and can add new QObjects if needed

Some examples of QObjects

- Basic info found in the raw file
  - `QPulse` (grants access to the sampled waveform)
  - `QPulseInfo` (channel number, trigger type - signal, noise, pulser…)
  - `QHeader` (run number, time)
- Containers and math tools
  - `QVector`
  - `QMatrix`
- Redefinition of base types to be compatible with diana trees
  - `QBool`
  - `QBaseType<T>` (T = int/float/double/…)

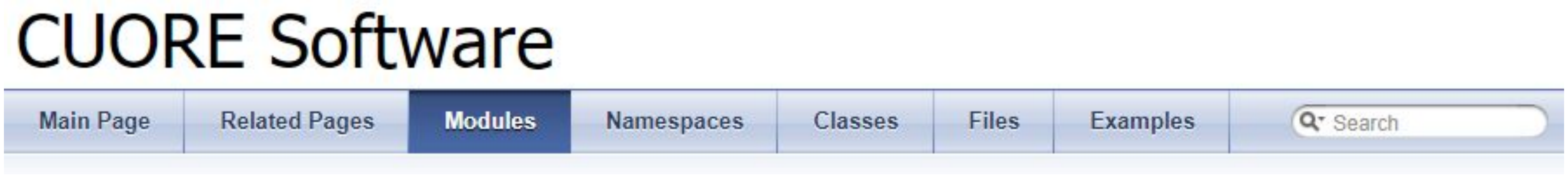
Diana documentation

The full diana documentation can be found here

https://cuore-collab.lngs.infn.it/swdoc/master/

It contains a description of what each module/filter/object/... does, the variables it contains and the methods to access them

Use the modules tab to navigate a full list with brief descriptions, or the search function if you're looking for a specific module



Diana manual

These slides are based on the diana manual, which can be found here

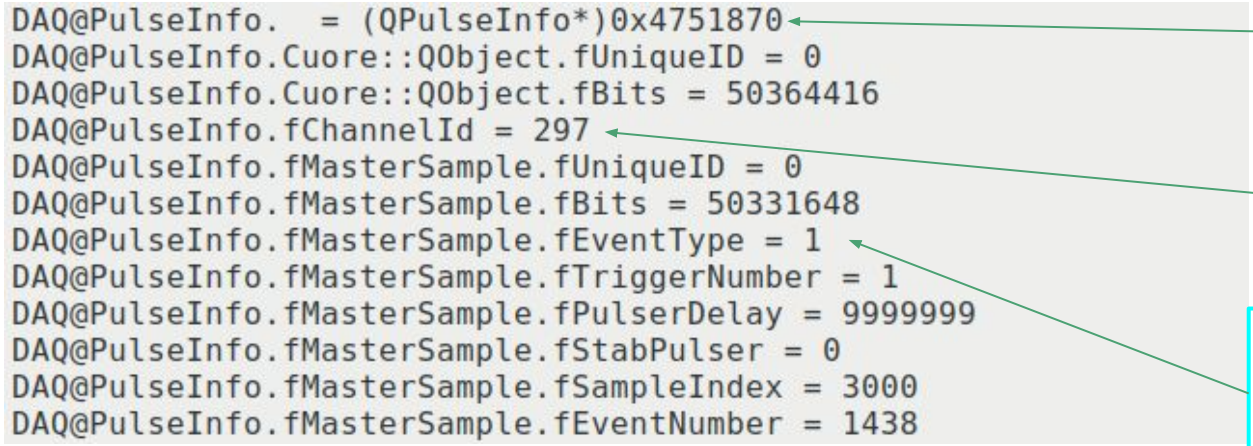
https://cuore-collab.lngs.infn.it/swdoc/master/DianaManual/DianaManual.pdf

The manual is linked from the home page of the SW documentation as well

These slides are just the basics, the manual is much more complete!

Diana documentation - example

We need to recover the Channel number associated to an event and to check whether it’s a signal or noise/pulser.
This is contained in the `DAQ@PulseInfo` object, whose type is `QPulseInfo`



```
DAQ@PulseInfo.  = (QPulseInfo*)0x4751870
DAQ@PulseInfo.Cuore::QObject.fUniqueID = 0
DAQ@PulseInfo.Cuore::QObject.fBits = 50364416
DAQ@PulseInfo.fChannelId = 297
DAQ@PulseInfo.fMasterSample.fUniqueID = 0
DAQ@PulseInfo.fMasterSample.fBits = 50331648
DAQ@PulseInfo.fMasterSample.fEventType = 1
DAQ@PulseInfo.fMasterSample.fTriggerNumber = 1
DAQ@PulseInfo.fMasterSample.fPulserDelay = 9999999
DAQ@PulseInfo.fMasterSample.fStabPulser = 0
DAQ@PulseInfo.fMasterSample.fSampleIndex = 3000
DAQ@PulseInfo.fMasterSample.fEventNumber = 1438
```

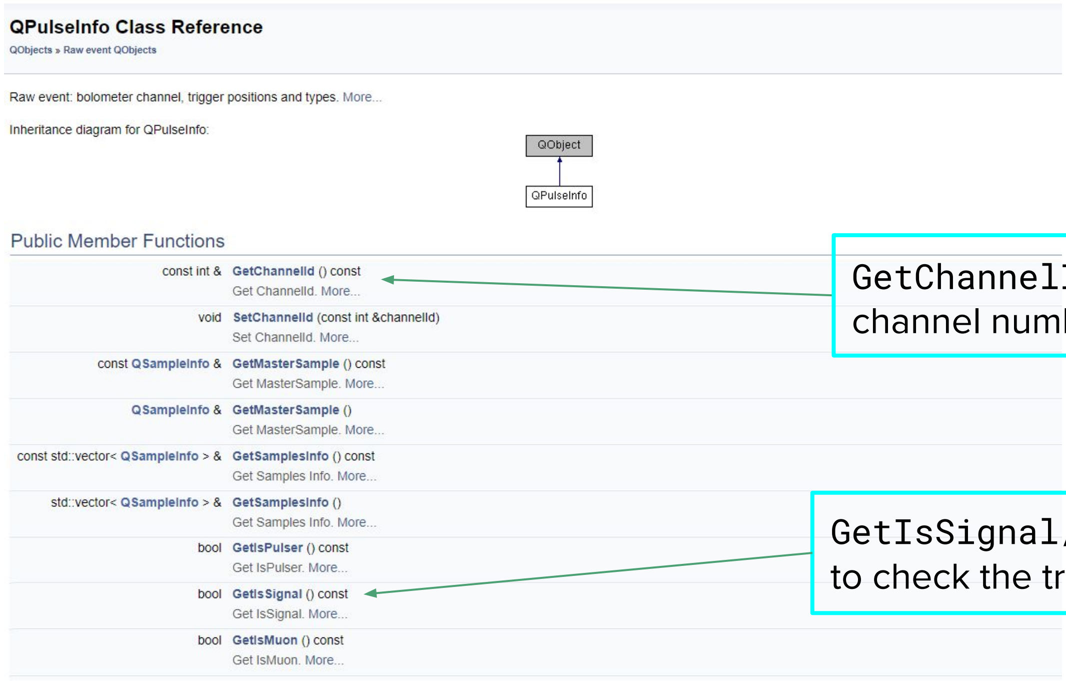
`DAQ@PulseInfo` is a `QPulseInfo` object

The variable `fChannelId` contains the Channel number

`fMasterSample.fEventType` contains the event type (signal, noise, pulser, ...)

Diana documentation - example

Searching for the QPulseInfo class yields its documentation



QPulseInfo Class Reference

QObjects » Raw event QObjects

Raw event: bolometer channel, trigger positions and types. More...

Inheritance diagram for QPulseInfo:

Public Member Functions

const int &	**GetChannelId** () const Get ChannelId. More...
void	**SetChannelId** (const int &channelId) Set ChannelId. More...
const **QSampleInfo** &	**GetMasterSample** () const Get MasterSample. More...
QSampleInfo &	**GetMasterSample** () Get MasterSample. More...
const std::vector< **QSampleInfo** > &	**GetSamplesInfo** () const Get Samples Info. More...
std::vector< **QSampleInfo** > &	**GetSamplesInfo** () Get Samples Info. More...
bool	**GetIsPulser** () const Get IsPulser. More...
bool	**GetIsSignal** () const Get IsSignal. More...
bool	**GetIsMuon** () const Get IsMuon. More...

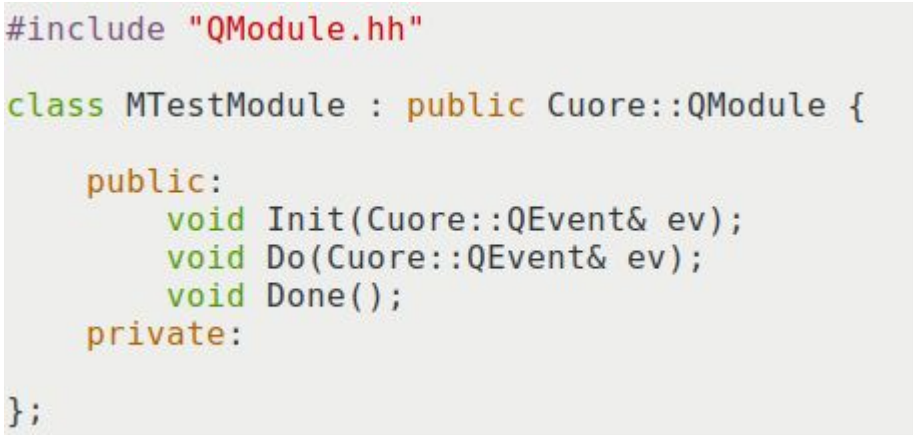
`GetChannelID()` gives us the channel number

`GetIsSignal/Pulser/Noise()` to check the trigger type

Structure of a diana module

Structure of a diana module

A diana module is a class which inherits from Cuore::QModule



```
#include "QModule.hh"

class MTestModule : public Cuore::QModule {

    public:
        void Init(Cuore::QEvent& ev);
        void Do(Cuore::QEvent& ev);
        void Done();
    private:

};
```

At least these three methods must be implemented:

```
void Init(Cuore::QEvent& ev);

void Do(Cuore::QEvent& ev);

void Done();
```

The Init method

The Init method is called once at the beginning of the execution. Uses:

- Get parameters from the cfg file containing the current sequence
- Check if the required input variables exist
- Setup the output variables produced by the module
- Load external files (e.g. txt) to be used later
- Read/write global data

Init: get parameters from cfg

There's a few Get methods that can be used to read options from the cfg and convert them to the required format:

`GetDouble, GetInt, GetBool, GetString, GetVectorDouble, GetVectorInt, GetVectorBool, GetVectorString`

https://cuore-collab.lngs.infn.it/swdoc/master/classQBaseModule.html

Init: get parameters from cfg

All the Get methods share the same basic structure

```
double GetDouble(const std::string &parName, double defVal, bool warnCfg=true) const;
```

- parName: name of the parameter in the cfg file (type: std::string)
- defVal: default value if not found in cfg (type: same as return type)
- warnCfg: print a warning if not found in cfg (type: bool)

Init: get parameters from cfg

Cfg file:

```
module TestModule
enable = true
maxEnergy = 500
inputFile = testFile.txt
steps = 10,20,30,40
endmod
```

MTestModule.hh :

```
class MTestModule : public Cuore::QModule {
    private:
        double fMaxEnergy;
        std::string fInputFile;
        std::vector<int> fSteps;
```

MTestModule.cc :

```
void MTestModule::Init(QEvent& ev)
{

    fMaxEnergy = GetDouble("maxEnergy", 100);
    fInputFile = GetString("inputFile", "", true);

    std::vector<int> defaultSteps { 50, 100 };
    fSteps = GetVectorInt("steps", defaultSteps);
```

Init: check input variables

Before moving on to the bulk of the processing, the existence of the required input variables must be verified.

This is done by the `Require` method:

```
void MTestModule::Init(QEvent& ev) {

    ev.Require("owner", "name");
    ev.RequireByLabel("owner@name");
}
```

These are equivalent, only one should be used

Init: check input variables

In a previous example we saw that the channel number for each event is contained in the `DAQ@PulseInfo` object.

If we need to use the channel number in our main event loop, we should require that the input file contains `DAQ@PulseInfo`

```
void MTestModule::Init(QEvent& ev) {
    ev.Require("DAQ", "PulseInfo");
    ev.RequireByLabel("DAQ@PulseInfo"); // Same as above
}
```

If the object doesn't exist, the execution is halted immediately and an error is printed

Init: adding new QObjects to the output file

If the module is adding new QObjects to the output file, this should be done here

```
void MTestModule::Init(QEvent& ev) {
    ev.Add <QObjectType> ("name");
}
```

This will add an object of type QObjectType (QDouble, QInt, QVector, ...) named 'name' to the output file. The owner of this QObject will be the current module (`MTestModule@name`)

Here we're just creating a new output variable, it will be filled in the main event loop

The Do method

The Do method is called for each event. Uses:

- Perform actions on each event. Often, bulk of the processing
- Read QObjects from the input file
- Write new QObjects to the output file
- Read/write global data

Do: read QObjects from the input file

The contents of a QObject (after being `Required` in Init) can be recovered with Get, specifying the QObject type and name

```
void MTestModule::Do(QEvent& ev) {
    QPulseInfo& pi = ev.Get<QPulseInfo>("DAQ@PulseInfo");
    QPulseInfo& pi = ev.Get<QPulseInfo>("DAQ", "PulseInfo"); // Same as above

    int channel = pi->GetChannelId();
…
}
```

Do: write QObjects to the output file

After Adding a QObject to the output file in Init, it is filled in Do:

```
void MTestModule::Init(QEvent& ev) {
    ev.Add<QObjectType>("name"); // Add MTestModule@name (type: QObjectType)
}

void MTestModule::Do(QEvent& ev) {
    // Recover MTestModule@name and fill it
    QObjectType& obj = ev.Get<QObjectType>("name");
    obj = …;
}
```

The Done method

Done is called once at the end of execution. Uses:

- Operate on quantities collected during processing (e.g. fill spectra in Do, find calibration coefficients in Done, when all data is collected)
- Operate on sequence execution (set it to run another time instead of once)
- Cleanup (clear vectors, maps, plots...)

Done: operate on sequence execution

Modules can ask for a reiteration of a sequence, useful for iterative algorithms

- `SetRunAgain()`: make the sequence run again
- `GetIteration()`: get number of sequence iterations
- `GetRunAgain()`: check if the sequence will be ran again

All these functions can be used in Init, Do and Done alike; it is more common to see them in Done

Global data

Global data contains quantities that do not vary event by event, but are common to all of them (i.e. DAQ parameters for the current run)

They can be accessed (read/write) in Init, Do or Done, depending on the application

Global data can be accessed from:

- Cache (previosly loaded during the sequence)
- External file containing (a) QObject(s)
- DB

Reading global data: handles

You can interact with global data with *handles*, which take care of the I/O

All QObjects can be interacted with via a `GlobalHandle.` This works for QObjects stored/to be stored in an output file

When the objects have to be loaded from the DB the generic `GlobalHandle` is no longer sufficient; a specific handle, which can communicate with the required DB table(s), must be used/created

Reading global data

Global objects are accessed via handles and read from `GlobalData()`

```
GlobalHandle<QObjectType> gHandle("name");
// If needed, additional info can be specified (e.g. channel)
gHandle.SetChannel(3);
// Get from cache
GlobalData().Get("owner", &gHandle, "");
// Get from external file
GlobalData().Get("owner", &gHandle, "filename.ext");
// Get from DB, only if not using a GlobalHandle
GlobalData().Get("owner", &gHandle, "DB");

// Get the QObject
const QObjectType& gObject = gHandle.Get();
```

Reading global data - QRunData

A commonly used global object is QRunData, which contains details on the current run (run number, start/stop date, list of channels…)

https://cuore-collab.lngs.infn.it/swdoc/master/classQRunData.html

As it can be read from the DB it has a specific handle, which takes the run number as a parameter:

```
QRunDataHandle (const int run, const std::string &name="RunData")
```

Reading global data - QRunData

As it is used very often, essentially every sequence contains an instance of `MRunDataLoader`, which loads the current QRunData (cached)

```
int runNumber = 300123; // Run number is contained in DAQ@Header (QHeader)
QRunDataHandle rHandle(runNumber);

GlobalData().Get("DAQ",&rHandle,""); // Object is cached
const QRunData& runData = rHandle.Get();

time_t startDate = runData.fStartDate;
```

Writing global data

Global QObjects are written via `GlobalHandles`, with the same caveats for DB access

```
QObjectType gObject;
GlobalHandle<QObjectType> outGHandle("name");
outGHandle.SetChannel(3);
outGHandle.SetRun(300123);
outGHandle.Set(gObject);

// Write CurrentModule@name (type: QObjectType) to filename.root
GlobalData().Set( outGHandle, "filename.root" );
```

On-screen output

In order to print messages on screen and in the log file, avoid using the standard c++ method (cout).

```
// Send a debug message (used to debug the module)
void Debug (const char *descr,...)

// Send an info message (information)
void Info (const char *descr,...)

//Send a warning message (an error that can be recovered)
void Warn (const char *descr,...)

// Send an error message (an error that cannot be recovered)
void Error (const char *descr,...)

// Send a panic message (stops diana now)
void Panic (const char *descr,...)
```

QObject validity

Some quantities are not computed on all events (e.g. some runs/channels may be filtered out by the sequence)

When this happens, modules still write QObjects for these events but they are filled with default values and set as not valid

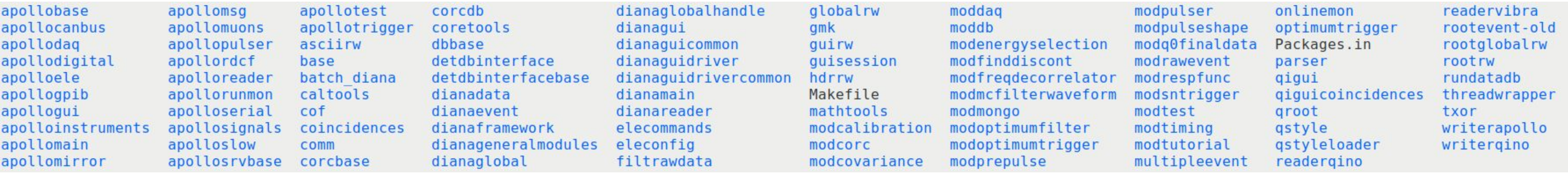
To avoid uncontrolled behaviour, you can check for object validity before using them

```
QObjectType& obj = ev.Get<QObjectType>("owner", "name");
if( !obj.IsValid() ) {
   Panic("Invalid object");
}
```

Where and how to create a new module

Diana packages

Diana components are split in packages, located in the `cuoresw/pkg/` folder



```
apollobase         apollomsg      apollotest     corcdb               dianaglobalhandle     globalrw        moddaq               modpulser       onlinemon          readervibra
apollocanbus       apollomuons    apollotrigger  coretools            dianagui              gmk             moddb                modpulseshape   optimumtrigger     rootevent-old
apollodaq          apollopulser   asciirw        dbbase               dianaguicommon        guirw           modenergyselection   modq0finaldata  Packages.in        rootglobalrw
apollodigital      apollordcf     base           detdbinterface       dianaguidriver        guisession      modfinddiscont       modrawevent     parser             rootrw
apolloele          apolloreader   batch_diana    detdbinterfacebase   dianaguidrivercommon  hdrrw           modfreqdecorrelator  modrespfunc     qigui              rundatadb
apollogpib         apollorunmon   caltools       dianadata            dianamain             Makefile        modmcfilterwaveform  modsntrigger    qiguicoincidences  threadwrapper
apollogui          apolloserial   cof            dianaevent           dianareader           mathtools       modmongo             modtest         qroot              txor
apolloinstruments  apollosignals  coincidences   dianaframework       elecommands           modcalibration  modoptimumfilter     modtiming       qstyle             writeapollo
apollomain         apolloslow     comm           dianageneralmodules  eleconfig             modcorc         modoptimumtrigger    modtutorial     qstyleloader       writerqino
apollomirror       apollosrvbase  corcbase       dianaglobal          filtrawdata           modcovariance   modprepulse          multipleevent   readerqino
```

Each package is related to a specific function, new related objects should go in the correct folder

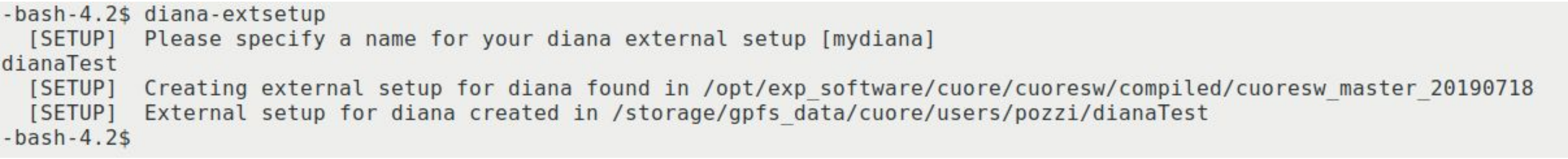
If an entirely new package is created, its folder name must be added to `Packages.in` ; this is required in order for the new package to be compiled

Working on a package: external setup

In order to work on a single package (e.g. developing a new module) it is not necessary to get the whole cuoresw repo and compile diana from scracth

First, setup an external installation:

- Select a working diana version by running its `setup.sh` script
- Run `diana-extsetup` and select a name for the new folder (e.g. dianaTest)



```
-bash-4.2$ diana-extsetup
  [SETUP]  Please specify a name for your diana external setup [mydiana]
dianaTest
  [SETUP]  Creating external setup for diana found in /opt/exp_software/cuore/cuoresw/compiled/cuoresw_master_20190718
  [SETUP]  External setup for diana created in /storage/gpfs_data/cuore/users/pozzi/dianaTest
-bash-4.2$
```

- This will create a dianaTest subfolder from where you ran `diana-extsetup`

Checking out a single package

This gives you a folder to work in which, by running the setup.sh script within, references and uses the “main” diana installation

You can now get a single package from the cuoresw repository by running

```
diana-checkout-pkg modprepulse
```

The (previously empty) pkg folder now contains the ‘modprepulse’ package

This also makes the ext. setup folder a working git directory, so any changes to ‘modprepulse’ can be committed to the main repository

Working on a single package

Diana doesn’t need to be recompiled/linked in order to work with the updated code

- Checkout a single package, modify it and compile it
  - Only the updated package is compiled, the original installation is unchanged
  - This creates a shared library in your external install
- Run diana
  - All packages are taken from the original diana installation..
  - .. with the exception of the one in the ext. setup

Creating a new diana module

The easiest way to create a new module is to run

```
diana-createmodule TestModule
```

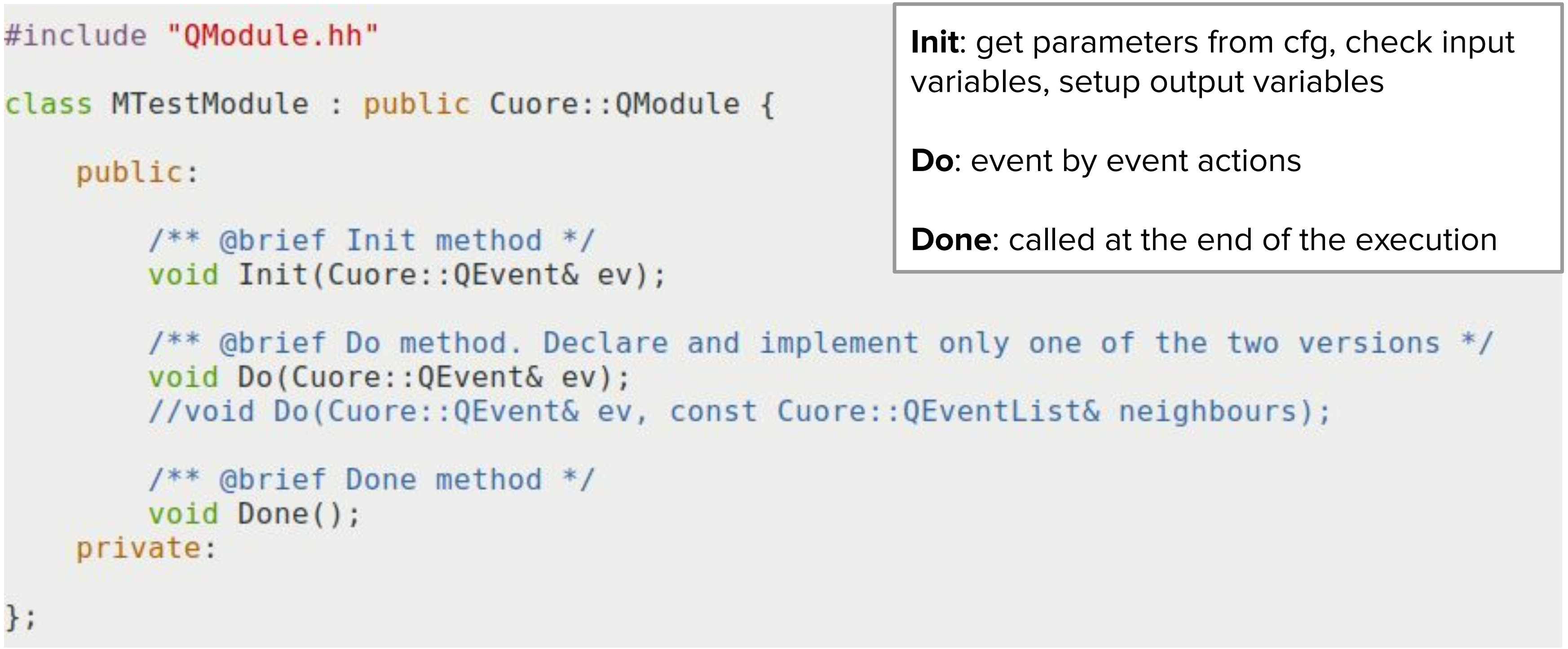
from the `cuoresw/pkg/modname/` folder. This will create two files:

```
MTestModule.cc    MTestModule.hh
```

These contain a skeleton for a new module

The source files for **all modules** must start with a capital M

MTestModule.hh



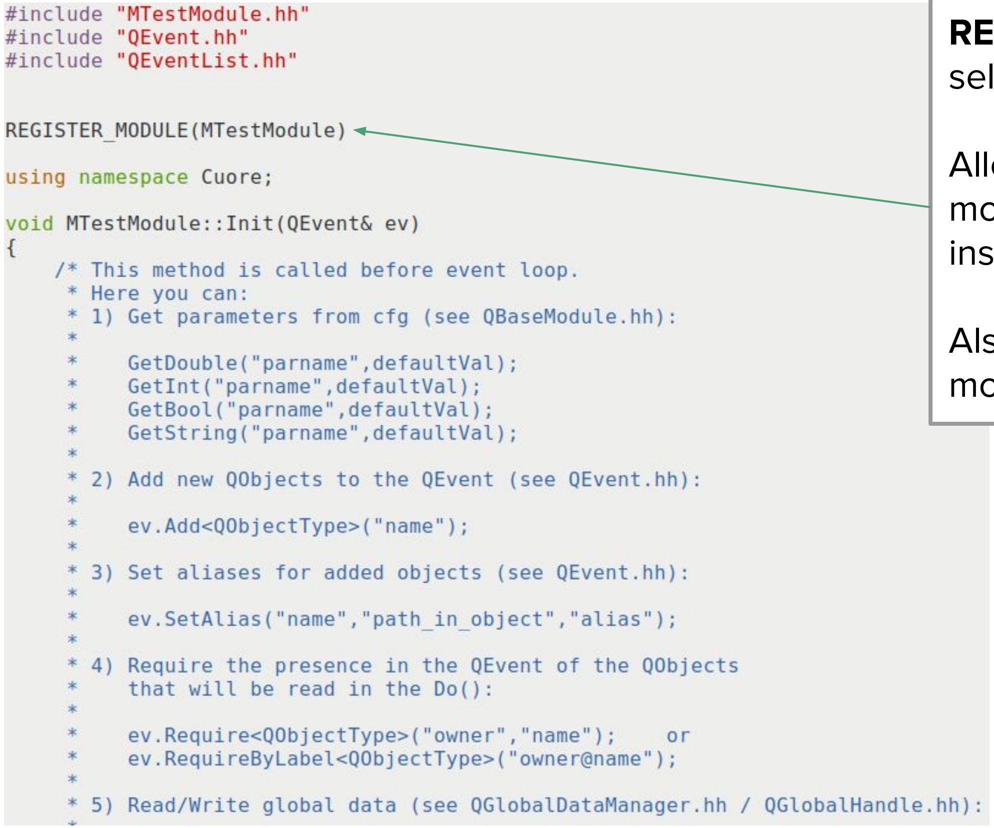
```cpp
#include "QModule.hh"

class MTestModule : public Cuore::QModule {

    public:

        /** @brief Init method */
        void Init(Cuore::QEvent& ev);

        /** @brief Do method. Declare and implement only one of the two versions */
        void Do(Cuore::QEvent& ev);
        //void Do(Cuore::QEvent& ev, const Cuore::QEventList& neighbours);

        /** @brief Done method */
        void Done();
    private:

};
```

Init: get parameters from cfg, check input variables, setup output variables

Do: event by event actions

Done: called at the end of the execution

MTestModule.cc



```
#include "MTestModule.hh"
#include "QEvent.hh"
#include "QEventList.hh"


REGISTER_MODULE(MTestModule)

using namespace Cuore;

void MTestModule::Init(QEvent& ev)
{
    /* This method is called before event loop.
     * Here you can:
     * 1) Get parameters from cfg (see QBaseModule.hh):
     *
     *    GetDouble("parname",defaultVal);
     *    GetInt("parname",defaultVal);
     *    GetBool("parname",defaultVal);
     *    GetString("parname",defaultVal);
     *
     * 2) Add new QObjects to the QEvent (see QEvent.hh):
     *
     *    ev.Add<QObjectType>("name");
     *
     * 3) Set aliases for added objects (see QEvent.hh):
     *
     *    ev.SetAlias("name","path_in_object","alias");
     *
     * 4) Require the presence in the QEvent of the QObjects
     *    that will be read in the Do():
     *
     *    ev.Require<QObjectType>("owner","name");    or
     *    ev.RequireByLabel<QObjectType>("owner@name");
     *
     * 5) Read/Write global data (see QGlobalDataManager.hh / QGlobalHandle.hh):
```

REGISTER_MODULE: mandatory, self-registration of the module

Allows the conversion of a string with the module name (read by the cfg) to an actual instance of MTestModule

Also makes diana “know” about this module without recompilation/linking

A simple module: calculate the baseline

```
#include "QModule.hh"
#include <string>

class MTestModule : public Cuore::QModule {
    public:
        void Init(Cuore::QEvent& ev);
        void Do(Cuore::QEvent& ev);
        void Done();

    private:
        /** @brief Average value of baseline */
        double fBaseline;
        /** @brief Number of points for calculation */
        int  fNumberOfPoints;
        /** @brief Label of the QPulse object */
        std::string fPulseLabel;
};
```

MTestModule.hh

`Init`, `Do` and `Done` are mandatory

`fBaseline` is the output we need

`fNumberOfPoints` and `fPulseLabel` are parameters from the cfg file

MTestModule - Init

```
REGISTER_MODULE(MTestModule)

using namespace Cuore;

void MTestModule::Init(QEvent& ev) {
    // Get number of points from cfg - default 1000
    fNumberOfPoints = GetInt("NumPoints", 1000, false);
    // Label of the QPulse object - default DAQ@Pulse
    fPulseLabel = GetString("PulseLabel", "DAQ@Pulse", false);

    // Required objects
    ev.RequireByLabel<QPulse> (fPulseLabel);
    ev.Require<QHeader> ("DAQ", "Header");
    ev.Require<QPulseInfo> ("DAQ", "PulseInfo");

    // Add the value of the baseline to output
    ev.Add<QDouble>("Baseline");
}
```

MTestModule.cc - Init

Get parameters from the cfg file

Require that objects that will be used in Do exist in the input file

Add a new QObject to the output file

MTestModule - Do (1)

```
void MTestModule::Do(QEvent& ev) {
    // Recover the pulse, header and pulseInfo
    const QPulse& pulse = ev.GetByLabel<QPulse>(fPulseLabel);
    const QHeader& header = ev.Get<QHeader>("DAQ", "Header");
    const QPulseInfo& pulseInfo = ev.Get<QPulseInfo>("DAQ", "PulseInfo");

    // Get the pulse samples
    const QVector& samples = pulse.GetSamples();

    // Get QRunData from GlobalData
    int runNumber = header.GetRun();
    QRunDataHandle rHandle( runNumber );
    GlobalData().Get("", &rHandle, "");
    const QRunData& runData = rHandle.Get();

    // ADC2mV is in QChannelRunData
    int channel = pulseInfo.GetChannelId();
    const QChannelRunData& chanRunData = runData.GetChannelRunData(channel);
    const double ADC2mV = chanRunData.fADC2mV;
...
```

MTestModule.cc - Do

Get input QObjects from the event

Get the pulse samples from QPulse

Get ADC2mV conversion from global data

MTestModule - Do (2), Done

```
...
    // Recover the output variable
    fBaseline = ev.Get<QDouble>("Baseline");
    // Calculate the average baseline
    fBaseline = samples.Sum( fNumberOfPoints,  0 );
    fBaseline /= fNumberOfPoints;
    fBaseline *= ADC2mV;
}

void MTestModule::Do(QEvent& ev) {
    // Nothing to do here
}
```

MTestModule.cc - Do (2), Done

Recover the output variable (added in Init) and fill it with the average baseline

Done is empty, in this case we don't need to operate on collected quantities or clear vectors/maps/...

The module is ready to be compiled; this can be done from within the package folder in your ext. setup directly