ATLAS Online L1 Calorimeter Trigger Monitoring Framework

User – Developer Guide

version 1.0

07/13/04

Adrian Mirea

CCLRC, Rutherford Appleton Laboratory

Table of Contents

Acronims Index

API	Advanced Programming Interface
CDS	Complex Data Structures
DAQ	Data Acquisition
DO	Data Objects
EMS	Event Monitoring Stream
GUI	Graphical User Interface
HD	Histogram Display
HO	Histogram Object
IS	Information Service
MF	Monitoring Framework
MISS	Monitoring Information Service Server
MP	Monitoring Provider
OH	Online Histogramming
OV	Options Vector
RHO	Raw Histogram Object
SAO-API	Standard ATLAS Online API
UCM	User Callable Methods

1 Design Principles

1.1 The Monitoring Framework

The monitoring of a complex DAQ system like ATLAS L1 Calorimeter Trigger [1] is the main task of a separate system, named **Monitoring Framework (MF)**. The MF has been designed and developed in full compliance with the **standard ATLAS online API (SAO-API)**[2] and its principle is depicted in Figure 1. It consists of three parts: the **Monitoring Provider (MP)**, the **Monitoring IS Server (MISS)**[3] and the **Histogram Display (HD)**.

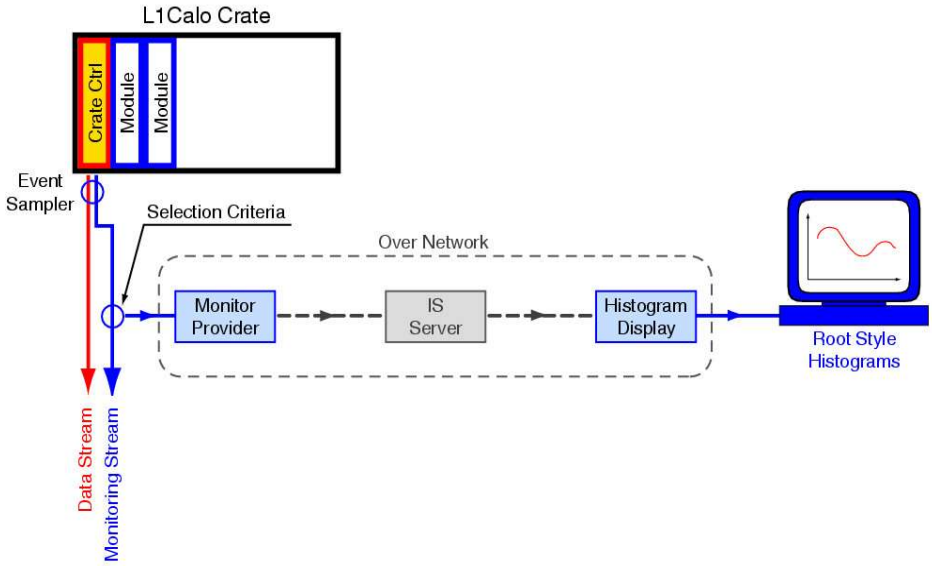

Figure 1. The ATLAS L1 Calo Trigger Monitoring Framework.

The MP is the server side application of the Monitoring Framework and one can connect it with any SAO-API **event monitoring stream (EMS)**[4]. The main purpose of the MP is to select events from the EMS according to a set of selection criteria, process them through a user algorithm, produce a set of histograms and to publish them on a dedicated MISS.[1]

1 One of the future developments of the MP would be to be able receive, process, publish complex user data structures which involves the definition of certain mechanisms of encoding and serialization that have to be compatible with the SAO-API. This will allow to build in a fairly straightforward manner additional monitoring streams that deal with data not embedded in the event stream (e.g. global statistics variables, error counters, etc.).

The MISS is in fact a pure SAO-API IS server and its purpose is to receive, hold and serve on request formatted byte streams. In the MF case, the byte streams are preformatted as ROOT histograms[5] or **complex data structures (CDS)**.

At the end of the MF chain stands the HD which acts as a client for the MISS. The purpose of the MF is to browse the MISS for any ROOT histogram objects or CDSes and to display them on the screen[2].

The communication protocol between MP and MISS and between MISS and HD is completely defined by the SAO-API.

1.2 The Monitoring Provider

The MP is a multithreaded application[6] that can be implemented as a `main()` routine or embedded in an already existing program. The main thread implements the object **publishing loop** while the analyse thread implements the **event loop** – mainly data processing and object updating (see Figure 2).

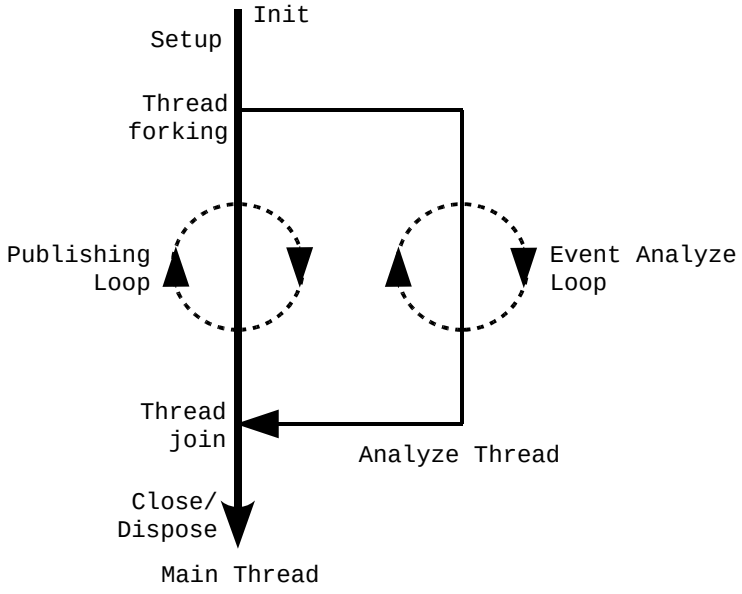

Figure 2. The ATLAS L1 Calo Trigger Monitoring Provider process flow.

Taking into account that the event loop has a period of the order of $\sim 10^2$ ms and

2 A future development of the MF would be to associate with every complex user data structures certain methods that define them as “displayable”, by building on-the-fly on the client side histogram objects for the suitable members of such structures.

a reasonable publishing loop period might be ~1 sec, the purpose of this asynchronous decoupling is to allow for a better optimisation in the communication with MISS, together with a sustained capability of event processing [7],[8],[9].
The chosen design for the data flow is presented schematically in Figure 3. In order to allow data access outside this environment and the exchange between the threads, all published objects are created in a global namespace named `UserData`.

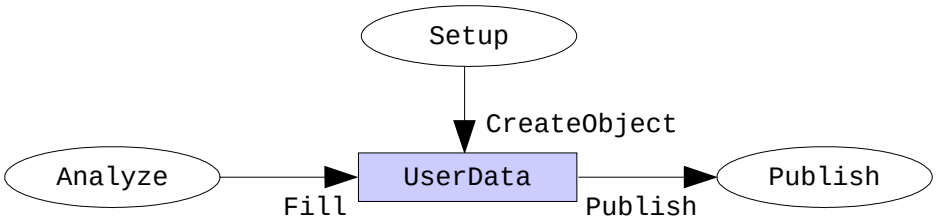

Figure 3. The ATLAS L1 Calo Trigger Monitoring Provider data flow.

The data access on the `UserData` namespace is regulated by means of read, write and update bitmasks that are also members of the same namespace[3]. These bitmasks are arrays of 32 bit integers where the size is the round-up to the next multiple of 32 of the number of histogram objects.
During the initialization, the bitmasks with the appropriate size for the estimated number of histogram objects are created and initialized in the `UserData` namespace. At setup, the histogram objects are created and initialized in the same namespace. In the event loop the histogram objects are updated and in the publishing loop the objects are red-out and published on the MISS. For every read operation of an object the associated read-mask bit is turned on and checked if the corresponding write-mask bit is off. At the end of the read operation, the read-mask bit is turned off. The same kind of logic protects the write operation.
The SAO-API implements two types of histogram objects: **raw histograms** and

3 In general the number of histogram objects might be of the order of hundreds or thousands. Therefore is not advisable to use the standard UNIX – IPC's (i.e. semaphores) or the standard thread mutexes as these are limited system resources (usually 1024) and their fastidious handling involves the kernel and in hence, for large numbers it might affect the overall system performance. Using these bitmasks the handling is transferred completely in the user address space [7],[9].

ROOT histograms[10]. The present implementation has opted for the raw histograms as they are quit general and allow for an easy overloading.

The phylosophy of the raw histograms is the following: for every histogram there are created two objects: a **data object (DO)** that basically is the memory support for the content of the histogram and the **raw histogram object (RHO)** itself that in principle implements the publishing methods on the MISS[4]. All histogram manipulation comes to some methods that act only on DOs.

The present package has defined the following templated classes `RAWData` - base class, `RAW1DData` - for one-dimensional and `RAW2DData` - for two-dimensional histograms that implement the DO (see Figure 4) and `RAWHistogram1D` and `RAWHsitogram2D` that implement the RHO. The DO support classes create some arrays to contain the data and errors and implement various histogram manipulation methods that act on these arrays. One can enumerate `PushBack` and `PushFront` – for 1d and `PushBackX` – for 2d that allow for time histograms (or history of a variable or an array of variables), `AddEntry` that allows for explicit manipulation of every member of the data and error arrays and `Fill` that allows for weighted distribution-like updating.

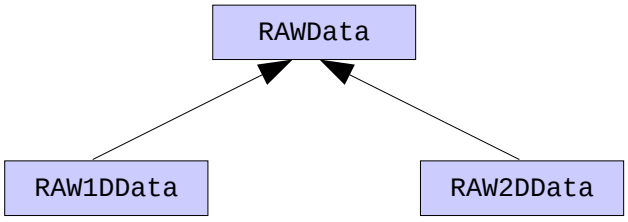

Figure 4. The inheritance diagram of the DO support classes.

The base class `ProviderUserClass` implements naturally the process flow design of Figure 2 and its extended class `RawProviderUserClass` implements the data flow of Figure 3. Wrapper methods for DO and RHO incorporate all optimisations and thread-safe data access mechanisms described previously. A `CreateObject` set of methods deal simultaneously with DO's

4 The SAO-API internally converts and casts the raw histograms in ROOT histograms during publishing so they are seen as ROOT histograms on the MISS.

and RHO's of every histogram object. Another significant optimisation feature is handled through the update-mask. If the DO's are updated with the wrapper methods of the `RawProviderUserClass`, then the associated bit of that histogram object in the update-mask is turned on. The `RawProviderUserClass::Publish` method will scan this mask and will publish on the MISS only those histogram objects for which the update-mask bits are on. After publishing, these update-mask bits are turned off. This proves to be a significant optimisation feature in the case of large number of histogram objects and in order to be effective, the users should take into account in their event loop to update only those histogram objects that have effectively changed.

The `RawProviderUserClass::CreateObject` adds automatically to the object names the strings "`:1d`", "`:2d`" or "`:3d`" to signal beforehand the histogram type and avoid dynamical casting tests on the HD side. These extensions are visible on MISS but are parsed and stripped off on the HD.

The developers are expected to extend the `RawProviderUserClass` and to overwrite those methods that are declared as `virtual` (`Init`, `Setup`, `Analyze` and `Dispose`). The users are free to add new members or (overloaded) methods to their extended class.

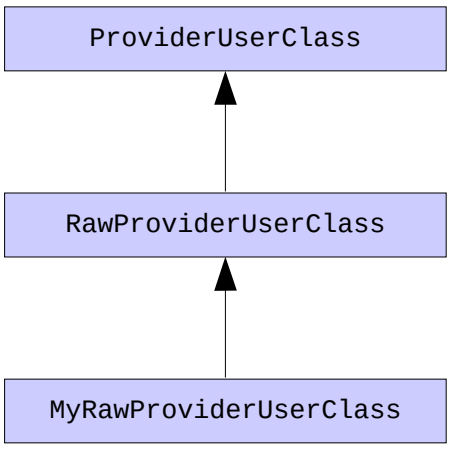

Figure 5. The inheritance diagram of the raw provider classes. `MyRawProviderUserClass` *is the one that the users should employ directly.*

At the end of this section we should mention that absolutely all SAO-API techniques and methods can be employed but in this case the user is fully responsible to implement himself the thread safety and useful optimisations[5].

1.3 The Histogram Display

As the histogram objects[6] appear in the MISS as ROOT histograms it becomes natural to design and develop the HD as a ROOT application. The HD is essentially a graphic interface that allows to browse the histogram content of a MISS and display it on the screen[7][8]. The GUI and the *look and feel* of the HD are implemented using the ROOT package. The `HDisplay` extends the `TApplication` class of ROOT and the execution happens in few steps: *instantiation*, *MISS setup*, *update timer setup* and *running*. The `HDisplay::Start` method calls in fact the `TApplication::Run` that spawns a new *event driven thread* that takes the control and *never returns*. The return methods currently used are the *ROOT signal and event handlers* and *exceptions*.

The HD consists of three main components implemented as separate classes:

- the **`HDisplay` class** – is the steering class mainly responsible for displaying the histograms and for the management of the **options vectors (OV)** for the canvas and every individual histogram;
- the **`HFolder` class** – spawned by the `HDisplay` and which is mainly a histogram object browser. A histogram object naming scheme similar to

5 A future development of the MF is to extend the `ProviderUserClass` to a new class named `RootProviderUserClass` that creates and manipulates true ROOT histogram objects in the same way as RawProviderUserClass deals with RHO's. The advantage would be the direct usage into the MP off all built-in ROOT histogram methods (i.e. fitting, profiling, slicing, integration, etc.) very useful in the event analysis.

6 From now on all histogram-like objects resident on or retrieved from an MISS will be collectively named **histogram objects (HO)**.

7 A future development of the HD would be to allow **User Callable Methods (UCM)** to run on selected histogram objects at runtime. As the HD is built as a batch ROOT application, these methods should be compiled in a user shared library and eventually configurable at runtime through a configuration file. Such methods might implement very easy algorithms like fitting, pattern recognition, reference histogram likelihood fits, warnings and alarms, etc.

8 Another future development of the HD would be to allow certain forms of default display mechanisms of complex data structures (CDS) exported into the MISS.

the UNIX filesystem pathnames has been implemented in such a way that for a name like "`a/b/c/myhist`" the `HFolder` is able to parse and display it like a virtual filesystem hierarchical tree with `a`, `b` and `c` shown as subdirectories and `myhist` as an object casted with the appropriate type (`TH1`, `TH2` or `TH3` ROOT histogram). All the objects that share virtual subpaths are collected and displayed automatically in the object browser as such. One have to mention that such naming scheme has absolutely no meaning for the MISS. In this way it is very simple to organize and display hundreds or thousands of histograms[9].

- the **`DynamicOptionsPanel` class** – spawned by the HFolder class upon certain mouse actions over selected histogram objects in the object browser. This is the GUI that allows on an *object per object* basis to display and modify the various options. The new options are individually stored for every object during the lifetime of the application and can be saved in a configuration file and retrieved at the next start-up.

The main interprocess communication mechanisms used by HD components are the ROOT signals and event handlers, callback methods, pointers to shared objects and exceptions.

Lets have a closer look at two of the most important data flows of the HD: the *histogram object handling*, and the *objects options handling*.

1.3.1 The histogram object (HO) handling

The MISS connection and the memory allocation for every HO is handled entirely by the `HDisplay`. To locate unambiguously the MISS, the SAO-API requires a partition, an IS server and a provider name to be given. In the current implementation these are start-up parameters[10]. The `HDisplay::SetServer` called after instantiation and before `HDisplay::Start`, calls indirectly `HDisplay::GetObjectsList` and `HDisplay::CreateOHIt-`

9 Another future development of the HD is to implement inner or UCM's that act collectively on groups of objects that share a given virtual pathname.

10 Another future development of the HD would be to be able to switch between different IS servers at runtime, without restarting the application and eventually display them simultaneously in the object browser.

erator. The result is an SAO-API OHHistogramIterator. Running all the possible iterations, the whole content of the MISS is browsed and the names of the HO's are stored in a variable of type std::vector<std::string> returned by HDisplay::GetObjectsList and stored in the variable HDisplay::ObjectList. This is the only complete iteration of the MISS necessary to build the object browser by a call to the method HDisplay::ObjectBrowser(ObjectList) (see Figure 6). All object names are parsed in HFolder for virtual pathnames and type and the object browser is built. HFolder installs a mouse handler[11] (HFolder::OnClick) and for every left button mouse click a callback into the parent HDisplay::BrowserHandler method is made with the true[12] name of the selected object and the "signal" SIG_DISPLAY_OBJECT. The MISS is iterated again in HDisplay::GetObject until the object with that name is found. The associated display options of that object (if any) are recalled (HDisplay::GetDynamicOptions) and the object is displayed on the screen (HDisplay::Draw) (see Figure 7).

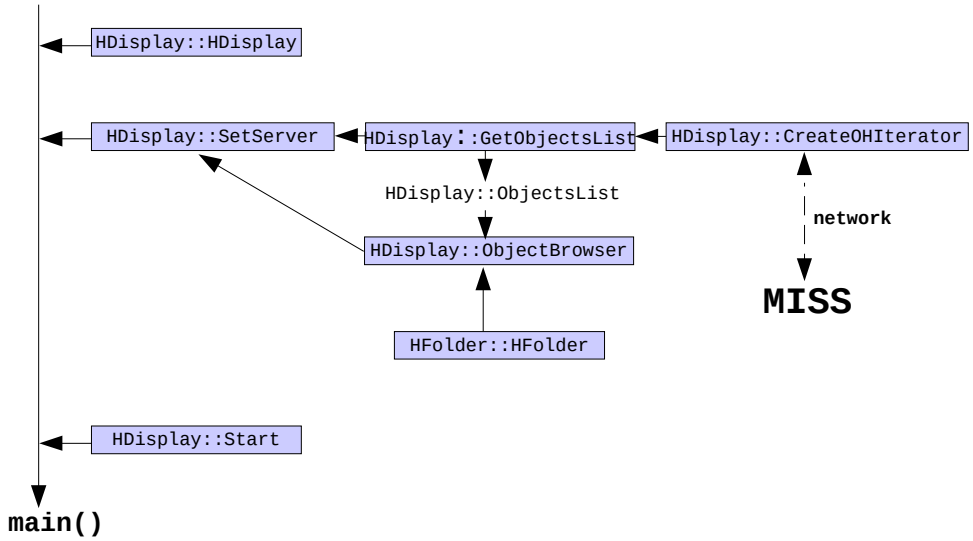

Figure 6. The process flow of HD start-up including the MISS browsing and the object browser (HFolder) thread spawning.

11 See [5].

12 A true name is something like "a/b/c/myhist:1d" and the displayed name would be only "myhist".

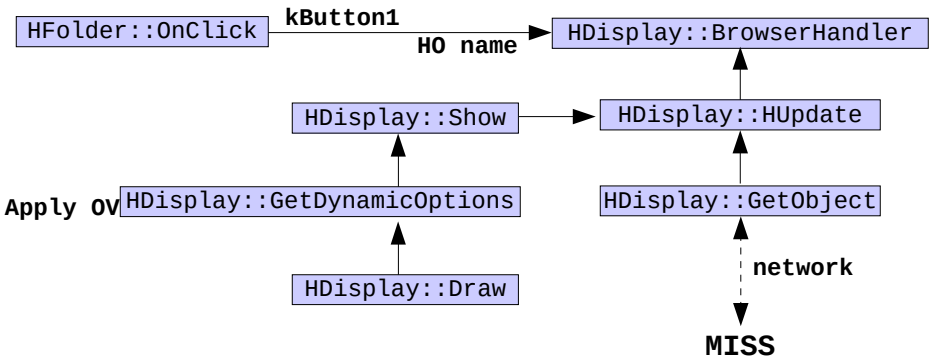

Figure 7. The process diagram of the HO update and display by selection (left button click) from the object browser.

1.3.2 The options vector (OV) handling

HD implements and handles two types of options: one for the display canvas and another one for the HO's. These options are mainly handled through a set of *user friend functions* of `HDisplay` organised in a *global namespace* (`DisplayUserFunctions`). The package comes with some default implementations of these functions and they are supposed to be modified, extended or overwritten by the users.

When `HDisplay` is instantiated, a call to the `HDisplay::Setup` is made. The purpose of this method is to read and parse the configuration file, if any (`HDisplay::ReadSetup`), to setup the canvas (`DisplayUserFunctions::CanvasSetup`) and a set of default options for the HO's (`DisplayUserFunctions::SetDefaultOptions`). All HO OV's are collected in the container `HDisplay::ObjectOptions`. The purpose of this container is to preserve and allow the user to change dynamically at runtime on a *object-per-object basis* the individual options.

For every attempt to display an HO on the sceen (`HDisplay::Show`) a check for a current OV in `HDisplay::ObjectOptions` for that HO is performed and if none is found, it is generated by cloning the default OV (`HDisplay::DefaultOptions`). The `HDisplay::ObjectOptions` contains only the OV's of those HO's that have been displayed at least once on the

screen. This is a reasonable optimisation feature particularly for MISS with a large number of HO's. By *File->Save* or *File->Save As* menu selection from the object browser (`HFolder::HandleMenuActivated`) a callback is performed (`HDisplay::WriteSetup`) and the current content the OV container[13] and that of the canvas (`HDisplay::CanvasOptions`) is saved in a configuration file. By reloading and parsing such a configuration file at start-up, the saved snapshot of the canvas and OV container content is generated.

The content of every individual OV can be accessed and modified at runtime via a dedicated GUI implemented by the `DynamicOptionsPanel` class and displayed by a middle button click (`kButton2`) over the selected HO in the object browser (`HFolder`). The mechanism is the following: before the instantiation of the `DynamicOptionsPanel`, a callback (`HDisplay::GetDynamicSetup`) is performed. This returns a copy of the HO's OV which is passed as an argument to the `DynamicOptionsPanel` constructor. The GUI is built according to the current content of that OV and the user is allowed to change its content. When the user selects the OK button in the GUI, a callback in `HFolder` (`HFolder::SetDynamicSetup`) is performed and the OV returned is forwarded by an new callback into the `HDisplay` (`HDisplay::SetDynamicSetup`).

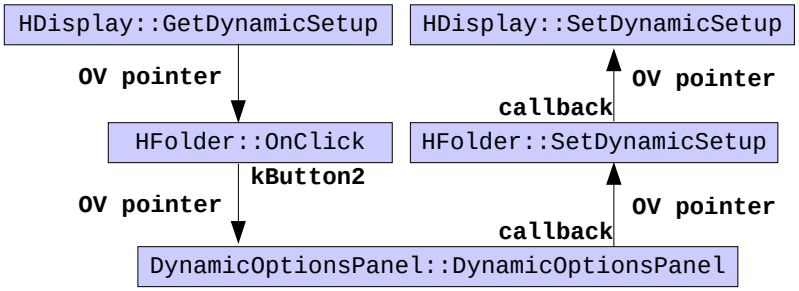

Figure 8. The process diagram of the HO OV's update by selection and middle button click from the object browser.

13 As only the displayed HO's trigger an OV cloning into the OV container, at the moment of the options saving, it might be possible to have different content for the same effective setup. It is not an error to have OV's for non existing objects (i.e. fetched from an old configuration file that doesn't match the actual HO content of the object browser (and MISS), but it might be an error if an OV exists for an HO with the same name and wrong casting type (`TH1`, `TH2`, etc).

2 How To

2.1 How to implement a Monitoring Provider

For a practical implementation of the MP the users should:

i. Extend the `RawProviderUserClass` by adding the following declaration:

```
class myRawProviderUserClass :
       public RawProviderUserClass;
```

ii. Overwrite its virtual methods:

```
////////////////////////////////////////////////
void myRawProviderUserClass::Init(u_int n)
{
  //Init the bitmasks for the estimated number of
  //1d and 2d HO's you intend to create
  Init(n, n);
  //Any other global initialization may go here
  ...
}

////////////////////////////////////////////////
bool myRawProviderUserClass::Setup()
{
  u_int n = 100 //estimated no of HO's to create.
  Init(100);
  /* a better optimised version would be:
    u_int n1d = 100 //estimated no of 1d HO's
    u_int n2d = 20 //estimated no of 2d HO's
    Init(n1d, n2d);
  */
  //create your 1d and 2d HO's here (see next subsection)
  ...
  //any other user setup may go here
  ...
  return true;
}

////////////////////////////////////////////////
void * myRawProviderUserClass::Analyze(void *arg)
{
  //suppose the argument to pass is a single integer
  int varg = *((int*)arg);

  //the main event loop
  while ( true ) {
    //do your analysis here and store
    //monitored data in local variables
    ...
```

```
      //fill the HO's here (see next subsection).
      //Optimisation hint: fill only changed variables.
      ...
      //anything else (e.g. Conditional checks, etc.)
      ...
    }
    //end of the event loop
}
```

This method is spawned automatically as a separate thread by the helper class `HThread` and the event loop is endless[14].

```
////////////////////////////////////////////////
void myRawProviderUserClass::Dispose()
{
    //add anything you want to do just before exit
    //e.g. save configurations, objects states, logs, etc.
    ...
}
```

iii. Build the Monitoring Provider application. In your `main()` routine or anywhere you would like to spawn the MP add a sequence simmilar to the following one:

```
...
//do some argument initializations or take them from
//somewhere else and skip all this section here
CmdArgStr partition_name('p', "partition",
                         "partition_name", "Partition name",
                         CmdArg::isREQ );
CmdArgStr server_name('s', "server", "server_name",
                         "OH (IS) Server name",
                         CmdArg::isREQ );
CmdArgStr provider_name('n',"provider","provider_name",
                         "Histogram provider name",
                         CmdArg::isREQ );
CmdArgInt timeout('t', "timeout", "timeout",
                         "Histogram provider timeout in sec
                         (default 1).");
CmdArgInt dbg('d', "debug", "dbg",
                         "Debug level (default 0).");
timeout = 1;
debug   = 0;
CmdLine cmd( *argv, &partition_name, &server_name,
             &provider_name, &timeout, &debug, 0, 0 );
CmdArgvIter arg_iter( --argc, ++argv );
cmd.description( "My RAW Monitor Provider example" );
cmd.parse(arg_iter);
```

14 Using conditional checks or throwing user defined exceptions (see for example the helper classes of CPPexception.h file) might be useful methods to break the loop.

```
 //create/attach to the IPC partition
 IPCPartition p(partition_name);

 //instantiate your extended provider class
myRawProviderUserClass *user =
   new myRawProviderUserClass(p,
                   (const char*)server_name,
                   (const char*)provider_name);

 //do the MP setup
user->Setup();

//prepare the helper HThread tool
HThread *thread = new HThread();

//start the event loop thread
pthread_t thId  = thread->start(analyze,(char*)&debug);

//enter the publishing loop[15]
while(true)
{
   //publish all updated HO's.
   user->Publish();

   //sleep for timeout seconds
   sleep(timeout);

   //do anything else here(e.g. conditional checks, etc.)
   ...
}
pthread_join(thread->getId(), NULL);

//last chance to do something at the end
//of the publishing loop
user->Close();

//stop the event loop thread
if ( thread ) thread->stop();

//do some clean-up
if ( user ) {
   delete user;
   user = 0;
}
...
```

15 Using conditional checks or throwing user defined exceptions (see for example the helper classes of CPPexception.h file) might be useful methods to break the loop.

2.2 How to create, update and manipulate HO's

Here are provided only examples related with the `RawProviderUserClass` methods. Nevertheless it is also possible to employ directly the whole range of methods to create and fill raw histograms of the SAO-API [10].

2.2.1 Creating a 1-dimensional HO

A simplified method for creating a 1-dimensional HO would look like this:

```
float min   = 0.;
float max   = 100.;
int   nbins = 50;
int hId10 = CreateObject(“a/b/c/myhist10”,
                     “My Title for h10” ,
                     “X axis”,
                     min, max, nbins);
//check for error
if ( !hId10 )
  std::cerr<<“Failed to create a/b/c/myhist10”<<std::endl;
...

//for some “exotic” axis labes and annotations
//to be attached to the HO use this one
std::vector<std::string> labels;
std::vector<std::string> annot;
int nlabels = 50;
for ( u_int i = 0 ; i < nlabels ; i++ ) {
  labels.push_back("label_”+i);
  annot.push_back("annotation ”+i);
}
int hId11 = CreateObject(“a/b/c/myhist11”,
                     “My Title for h11” ,
                     “X axis”,
                     labels, annot, min, max, nbins);
//check for error
if ( !hId11 )
  std::cerr<<“Failed to create a/b/c/myhist11”<<std::endl;
...
```

The index returned by `CreateObject` method is an integer `> 0` for 1d HO's, `< 0` for 2d HO's and `0` in case of error.

2.2.2 Creating a 2-dimensional HO

A simplified method for creating a 2-dimensional HO would look like this:

```
float xmin   = 0.;
float xmax   = 1.;
int   nxbins = 30;
float ymin   = 0.;
float ymax   = 1.;
int   nybins = 30;
```

```
int hId20 = CreateObject(“a/b/c/myhist20”,
                     “My Title for h20” ,
                     “X axis”, “Y axis”,
                     xmin, xmax, nxbins,
                     ymin, ymax, nybins);
//check for error
if ( !hId20 )
  std::cerr<<“Failed to create a/b/c/myhist20”<<std::endl;
...

//for some “exotic” axis labes and annotations
//to be attached to the HO use this one
std::vector<std::string> labels;
std::vector<std::string> annot;
int nlabels = 25;
for ( u_int i = 0 ; i < nlabels ; i++ ) {
  labels.push_back("label_”+i);
  annot.push_back("annotation ”+i);
}
int hId21 = CreateObject(“a/b/c/myhist21”,
                     “My Title for h11” ,
                     “X axis”, “Y axis”,
                     labels, annot,
                     xmin, xmax, nxbins,
                     ymin, ymax, nybins);
//check for error
if ( !hId21 )
  std::cerr<<“Failed to create a/b/c/myhist21”<<std::endl;
...
```

2.2.3 Retrieving the HO's index by name

Every HO has the references to its parameters, DO and RHO stored in some `std::deque` type containers in the `UserData` global namespace so they can be accessed from anywhere. The (absolute values of the) indexes returned by `CreateObject` methods are in fact the indexes of the objects in these containers. If the manipulation of integers is simpler and faster[16], nevertheless their values might have less meaning then the HO's names.

`RawProviderUserClass` has methods that allow to retrieve the index of an HO once the name is known. For this, use the following sequence:

```
...
int index = GetIndexByName(“a/b/c/myhist10”);
...
```

16 A good programming technique might be to place these indexes in the same `UserData` global namespace and use these ones instead of the HO's names.

2.2.4 Retrieving the HO type

One can retrieve the HO type (1d or 2d defined by the macros `H_1D` or `H_2D`) with the following sequences:

```
...
int type1 = GetType("a/b/c/myhist10");
//if idx is the index of HO "a/b/c/myhist10"
int type2 = GetType(idx);
...
```

2.2.5 Retrieving the HO's parameters

The pointer to the parameters structure of an HO (`Object1DParameters` or `Object2DParameters`) can be retrieved with the following sequence:

```
...
//1d case
Object1DParameters *par10;
int idx10 = GetIndexByName("a/b/c/myhist10");
if ( idx10 && (GetType(idx10)==H_1D) )
  par10 = Get1DParameters(idx10);
...
//2d case
Object2DParameters *par20;
int idx20 = GetIndexByName("a/b/c/myhist20");
if ( idx20 && (GetType(idx20)==H_2D) )
  par20 = Get2DParameters(idx20);
...
```

2.2.6 Retrieving the HO's DO or RHO components

The pointer to the DO of an HO can be retrieved with the following sequences:

```
...
//1d case
RAW1DData *do10;
int idx10 = GetIndexByName("a/b/c/myhist10");
if ( idx10 && (GetType(idx10)==H_1D) )
  do10 = Get1DData(idx10);

...
//2d case
RAW2DData *do20;
int idx20 = GetIndexByName("a/b/c/myhist20");
if ( idx20 && (GetType(idx20)==H_2D) )
  do20 = Get2DData(idx20);
...
```

and the pointer to the RHO of an HO can be retrieved with the following sequence:

```
...
//1d case
RawH1D *rho10;
int idx10 = GetIndexByName("a/b/c/myhist10");
if ( idx10 && (GetType(idx10)==H_1D) )
  rho10 = Get1DObjectByIndex(idx10);
//or equivalent
rho10 = Get1DObjectByName("a/b/c/myhist10");

...
//2d case
RawH2D *rho20;
int idx20 = GetIndexByName("a/b/c/myhist20");
if ( idx20 && (GetType(idx20)==H_2D) )
  rho20 = Get2DObjectByIndex(idx20);
//or equivalent
rho20 = Get2DObjectByName("a/b/c/myhist20");
...
```

2.2.7 Filling 1-dimensional and 2-dimensional HO's

The current implementation of `RawProviderUserClass` allows for the following types of OH updating:

1. ***Time evolution of simple types of variables (1d-HO's) and arrays of variables (2d-HO's)***
   - `PushBack` and `PushFront` – 1d HO's

```
...
t_Content myvar;
t_Error   myerror;
int idx10 = GetIndexByName("a/b/c/myhist10");
if (idx10 && (GetType(idx10)==H_1D))
  PushBack(idx10, myvar, myerror);

int idx11 = GetIndexByName("a/b/c/myhist11");
if (idx11 && (GetType(idx11)==H_1D))
  PushFront(idx11, myvar, myerror);
...
```

   - `PushBackX` – 2d HO's

```
...
int idx20 = GetIndexByName("a/b/c/myhist20");
if ( idx20 && (GetType(idx20)==H_2D) ) {
  Object2DParameters *par20 = Get2DParameters(idx20);
  u_int size = par20->nxbins;
  t_Content data[size];
  for (u_int i = 0 ; i < size ; i++ ) data[i] = sin(0.1*i);
  PushBackX(idx20, &data[0], size);
}
...
```

2. Individual bins (1d HO's[17])

AddEntry

```
...
int idx10 = GetIndexByName("a/b/c/myhist10");
if ( idx10 && (GetType(idx10)==H_1D) ) {
  Object1DParameters *par10 = Get1DParameters(idx10);
  u_int size = par10->nxbins;
  for ( u_int i = 0 ; i < size ; i++ ) {
    AddEntry(
      idx10,                                  //HO index
      i,                                      //bin number
      sin(0.1*i),                             //content
      sin(0.05*i),                            //error
      (t_Axis)(i*10)/(t_Axis)size));          //axis
  }
}
...
```

3. Weighted distributions (1d and 2d HO's)

Fill – 1d HO's

```
...
gRandom->SetSeed();
int idx10 = GetIndexByName("a/b/c/myhist10");
if ( idx10 && (GetType(idx10)==H_1D) ) {
  Object1DParameters *par10 = Get1DParameters(idx10);
  t_Axis x      = (par10->xmax – par10->xmin)/2.;
  float  weight = 0.5;
  Fill(idx10, gRandom->Poisson(x), weight);
}
...
```

Fill – 2d HO's

```
...
gRandom->SetSeed();
int idx20 = GetIndexByName("a/b/c/myhist20");
if ( idx20 && (GetType(idx20)==H_2D) ) {
  Object2DParameters *par20 = Get2DParameters(idx20);
  t_Axis x      = (par20->xmax – par20->xmin)/2.;
  t_Axis y      = (par20->ymax – par20->ymin)/2.;
  float  weight = 0.5;
  Fill(idx10,
       gRandom->Poisson(x),
       gRandom->Poisson(y),
       weight);
}
...
```

17 Currently only the 1d HO version is implemented. The 2d version is forseen too.

2.3 How to implement a Histogram Display

The HD can be implemented as a `main()` routine or embedded in an already existing program.

```
...
//do some argument initializations or take them from
//somewhere else and skip all this section here
CmdArgStr partition_name('p',"partition","partition_name",
         "Partition name", CmdArg::isREQ);
CmdArgStr server_name('s',"server","server_name",
         "OH (IS) Server name", CmdArg::isREQ);
CmdArgStr provider_name('n',"provider","provider_name",
         "Histogram provider name");
CmdArgStr histogram_name('h',"histogram","histogram_name",
         "Histogram type name");
CmdArgInt dtimeout('t',"timeout","dtimeout",
         "Display refresh time (ms)");
CmdArgStr config('c',"config","config",
         "Configuration file");
CmdArgInt verbosity('v',"verbosity","verbosity",
         "Verbosity level (default 0)");

verbosity = 0;

CmdLine cmd(*argv,&partition_name,&server_name,
         &provider_name,&histogram_name,&dtimeout,
         &config,&verbosity,0);
CmdArgvIter arg_iter(--argc,++argv);
cmd.description("My Histogram Display");
cmd.parse(arg_iter);

// Check some command line parameters
if( provider_name.isNULL( ) )  provider_name  = ".*";
if( histogram_name.isNULL( ) ) histogram_name = ".*";

//create/attach to the partition
IPCPartition p(partition_name);

//instantiate the HDisplay class
HDisplay *display;
if ( (display = new HDisplay(config, verbosity)) == 0 ) {
  if ( verbosity ) std::cerr <<
         "ERROR: creating HDisplay!..." << std::endl;
  return;
}
```

```
//setup the server. In case the MISS server is not found or
//is empty it throws CPPException::NoObjectsException.
try {
  display->SetServer(p,
                     (c_char*)server_name,
                     (c_char*)provider_name,
                     (c_char*)histogram_name);
}
catch ( CPPException::NoObjectsException )
{
   if ( verbosity ) std::cout <<
          "No objects found on the "<<partition_name<<"@"<<
          server_name<<"@"<<provider_name<<std::endl;
    return;
}

//setup the HD update timeout
display->SetTimeout(dtimeout);

//start & stay in the HD thread
display->Start();

//you can't get here
...
```

2.4 How to add new display options

One can add new (canvas or display) options in the following steps:

1. Define a name and give a default value (might be any kind of the simple types) in the `HOptions.h` file:

```
...
#define O_MYOPTION “myoption”
...
#define D_MYOPTION myvalue
...
```

2. If it is a canvas option implement the appropriate parsing sequence in the friend function `DisplayUserFunctions::CanvasSetup` of the file `DisplayUserFunctions.cxx`.
3. If it is an HO option then implement the appropriate parsing sequence in the friend function `DisplayUserFunctions::SetDefaultOptions` of the file `DisplayUserFunctions.cxx`.
4. Implement the parsing sequence in the method `DynamicOptionsPanel::DecodeOptions` in the file `DynamicOptionsPanel.cxx`.

5. Implement the parsing sequence in the method `DynamicOptionsPanel::WriteOptions` in the file `DynamicOptionsPanel.cxx`.
6. Add the appropriate ROOT widget in the constructor `DynamicOptionsPanel::DynamicOptionsPanel` of the file `DynamicOptionsPanel.cxx`.

Bibliography

1: ATLAS L1 Calorimeter Trigger Group, http://hepwww.rl.ac.uk/Atlas-L1/Home.html, 2004,ATLAS, CERN,,

2: ATLAS Online Group, http://atlas-onlsw.web.cern.ch/Atlas-onlsw/, 2004,,,

3: ATLAS Online Group, IS User's Guide, http://atddoc.cern.ch/Atlas/DaqSoft/components/is/ug/is_ug.html, 2004,ATLAS, CERN,,

4: ATLAS Online Goup, http://atddoc.cern.ch/Atlas/Notes/157/mon-ug.html, 2004,ATLAS, CERN,,

5: Rene Brun at All., ROOT, http://root.cern.ch/, 2004,CERN,,

6: Leroy, Xavier, Linux Threads Library, http://pauillac.inria.fr/~xleroy/linuxthreads/, 2004,INRIA,,

7: Stevens,W. Richard, Advanced Programming in the UNIX environment, 1992,Addison-Wesley Publishing Company,ISBN-0201563177,

8: Stevens,W. Richard, UNIX Network Programming, 1990,Prentice-Hall Inc.,ISBN-0139498761,

9: Bovet,P. Daniel, Cesati,Marco, Understanding the Linux Kernel, 2001,O'Reilly & Associates Inc.,ISBN-0596000022,

10: ATLAS Online Group, Online Histogramming Reference Manual, Core and User API, http://atlas-onlsw.web.cern.ch/Atlas-onlsw/oh/online-doc/developers/main.html, 2004,ATLAS, CERN,,

Alphabetical Index