ATLAS Level-1 Calorimeter Trigger
Software Note 012

29 November 2003
Version Draft 0.4

L1Calo Database User Guide

Murrough Landon

1 Introduction

This document describes the configuration database for the ATLAS Level 1 Calorimeter (L1Calo) Trigger [1]. It is primarily intended as a user guide and its intended audience is the (small!) community of L1Calo software developers.

The L1Calo database requirements and some early ideas for the database schema are given in [2]. That document is a little out of date, but is still useful in giving an overall picture.

Everything described in this document is an extension of the standard ATLAS Online configuration database. This has extensive documentation which is accessible via its website [3].

The Online database software is organised in two layers. The underlying OKS layer deals with the storage of objects in physical database files. It is very generic and flexible. Above this is the data access library (DAL) which converts generic OKS database objects to instances of C++ runtime classes that are very specific to the ATLAS Online software.

The L1Calo extensions are of three types. The schema of the OKS database has been extended to include new classes and subclasses of existing classes. Secondly, the DAL has been extended to implemented these as new C++ runtime classes. Lastly an "integrated" database layer has been added which combines static configuration data with volatile run parameters obtained from the Online software Information Service (IS) and calibration and trigger menu data.

1.1 Organisation of the Document

This document is organised as follows. The next section describes how the database and the associated libraries are divided into CMT packages, then section 3 shows the layout of the database files.

The L1Calo extensions to the standard Online software DAL are briefly covered in section 4 and the "integrated" database layer, which is intended as the main user interface, is described at some length in section 5.

Section 7 discusses the tools available to edit the database and lastly section 8 describes a few utility programs.

2 Organisation of CMT Packages

The term "L1Calo database" covers both the data files themselves, the database schema and also data access libraries (DALs) for reading (and updating) the data. These different aspects are implemented in a number of CMT packages.

- `dbFiles`: this package contains both the database schema files and the contents of the database. The files are organised in directories following the Online software conventions and which are described in more detail below. The `dbFiles` package depends on the schema files and some data files provided by the Online software.
- `isL1Calo`: this small package is used to generate C++ and Java classes for use in the Online software Information Service (IS). The package contains no code or data of its own. The `isL1Calo` package depends on the `is` package from the Online software.
- `infraL1Calo`: this package is also used to generate C++ "wrapped enum" classes which convert the string enumerations in the database schema to numeric enumerations wrapped up as a C++ class.
- `confL1Calo`: this rather large package contains all the code to implement the data access library (DAL) for the L1Calo configuration data. It also implements DALs for our other data (eg calibration and trigger menus) and an integrated software layer which combines all this information for each module. The `confL1Calo` package depends on the `oks` and `confdb` packages from the Online software.

2.1 The `dbFiles` package

The `dbFiles` package contains the OKS style XML files that comprise the database, ie the data and its schema – at least the L1Calo specific aspects of this.

The directory layout described below follows the recommendations of the Online software group, with some L1Calo extensions and the CMT infrastructure.

- `cmt`: this has the CMT infrastructure, ie the `requirements` file which installs the database files in our installation area, and the `checkxml.pl` script which checks the XML files for gross syntax errors which may be introduced by manual edits.
- `schema`: contains our schema files. For the moment these are split into several files but some may be combined in future. The `l1calo_hw` schema file contains the L1Calo extensions of the configuration database schema, the `l1calo_is` schema file defines our IS variables, the `calibration` schema file describes the objects in our calibration database and the `triggermenu`

schema file those forming the trigger menu. The latter two may be superceded in future by the conditions database and the CTP defined integrated trigger menu.

- `calib`: contains calibration data files.
- `hw`: standard location for data files describing the hardware configuration, eg workstations, crates and modules. We also use this directory for data files specifying cables and for the test vectors to be loaded into modules.
- `menu`: contains trigger menu data files.
- `mrs`: contains the MRS database for L1Calo error messages.
- `partitions`: standard location for data files describing top level TDAQ partitions and the run control hierarchy. We also have a file defining environment variables common to all our partitions. The private L1Calo naming convention is to use Capitals for top level partition files and lower case for files which are included by the top level files.
- `sw`: standard location for data files describing the software configuration, eg programs and instances of them in processes ("applications"). Examples include our run controllers and some test programs.

2.2 The `isL1Calo` package

The `isL1Calo` package is just used as a place holder for generated code. The `requirements` file and the `Makefile` (modified from the CMT default one) run the `is_generator.sh` script from the Online software `is` package to generate C++ and Java classes from the `l1calo_is` schema file.

2.3 The `infraL1Calo` package

The `infraL1Calo` package contains general infrastructure classes for the rest of the L1Calo software. Some of these are developed by hand. Others, the so called "wrapped enum" classes are generated by the `enumgen.pl` script. Some of these wrapped enum classes encapsulate string enumerations from the `l1calo_hw` database schema file as numeric enumerations inside a C++ class. These are safer than using bare strings in the rest of the code as typos can lead to runtime errors, whereas typos in enum constants should be flagged by the compiler.

2.4 The `confL1Calo` package

The `confL1Calo` package contains all the code to create run time instances of L1Calo specific C++ classes from the OKS objects in the database. This is done by extending the data access library (DAL) provided by the Online software for the standard database classes.

The source code is split into a number of subdirectories according to function. All the headers are in the single standard `confL1Calo` directory.

- `src/calib`: the classes in this directory describe the calibration data to be loaded into each module. At present the calibration objects are read from OKS XML files. In future there may be some more sophisticated connection to the ATLAS Conditions Database.
- `src/db`: this directory holds the top level "integrated" database classes. These classes act as facade classes bringing together data from the various databases and, to some extent, hiding their implementation. They are the classes which other parts of the L1Calo software are expected to use directly.
- `src/hw`: contains the classes which implement extensions of the standard DAL. The OKS schema for these classes is in `l1calo_hw.schema.xml`. In general, these classes are not expected to be used directly by other L1Calo software. The higher level `src/db` classes should be used instead.
- `src/qt`: this directory has the source and headers for an incomplete graphical trigger menu editor based on Qt. It is likely to be superceded by developments from the CTP group.
- `src/runtype`: this directory has classes which implement module settings which depend on the run type. However this idea may be abandoned in future.
- `src/trig`: has the classes which implement the trigger menu and which are based on the `triggermenu.schema.xml` schema file. These classes may be superceded by developments from the CTP group.
- `src/ui`: contains the Qt Designer user interface description for the graphical trigger editor in `src/qt`.
- `src/utils`: this directory has some utility classes used in reading OKS classes.

3 Database File Structure

The configuration database is physically stored in a number of OKS files. The database library defines three main categories of objects (hardware, software and partition) and requires them to be kept in separate files. Each TDAQ partition has one top level "partition" data file. However this may then include any number of schema files and other data files. This "federated" approach can be very useful in allowing several top level partition files to share a lot of their data.

3.1 Workstations

Every workstation (or crate based module with a CPU) on which the Online software runs must be identified in the database. It is convenient to have a separate file for the workstations at each site. Partitions which are specific to the hardware setup at a given site need only include that sites workstations. More general test partitions can include the workstation files for all our sites so that they can be run anywhere.

3.2 Environment Variables

Both the Online and L1Calo software need a number of environment variables set correctly for use by processes across the distributed system. These all need to be defined in the database. Since they are generally common to all partitions, all our environment variables are defined in a single `environment.data.xml` file.

NB by setting the value of the relevant `Environment` object in the database to the `${name}` style shell syntax, the database value will be taken from the users environment when the DAQ is started with the `play_daq` script. This is useful for keeping the database files themselves as site-independent as possible.

3.3 Crates and Modules

It is probable that we will try to have only a single top level partition file (with many test procedures defined) for a given hardware setup. However there may be several partitions which all use the same hardware. And in any case, the Online database requires "hardware" objects to be kept in separate files from "partition" objects.

At present we have one file describing the hardware setup for the CPROD tests at RAL and another file describing a generic slice test setup.

3.4 Cables

It may be that the same hardware setup is cabled up in different ways for different tests. In this case it may be useful to keep the cable objects separate from the crate and module objects.

3.5 Test Vectors

A similar argument applies to the specification of test vectors. So at present we have a separate file for each partition which lists the test vector setup for that partition.

3.6 Run Control Tree

The run control hierarchy and run control applications are considered as part of the "partition" class of database objects. However it can be useful to keep them separate from the top level partition file.

3.7 Software Repository

The software class of database objects comprised the descriptions of programs (the `SW_Object` class) and their implementation as binary files on different operating systems (the `Program` class). These objects are likely to be shared by all our partitions so they are all kept in a single file.

4 L1Calo Extensions to the Standard Database

This section describes the main L1Calo extensions and additions to the Online database schema and how these appear in the L1Calo extensions to the standard DAL. The schema extensions are kept in the `l1calo_hw` schema file.

Note that for C++ users, the standard DAL is generally hidden behind the facade of our "integrated" database layer which is described in section 5. However this section may be useful in understanding how to edit the database which is discussed in section 7.

4.1 `L1CaloPartition`

The OKS `L1CaloPartition` is a subclass of the standard OKS `Partition` class. It adds relationships to two `GenericFileList` objects: `calibrationList` and `triggerMenuList` are lists of calibration and trigger menu XML files available for this partition. There is also a relationship `dataGenRecipes` to the list of test vector descriptions ("DataGenRecipes") and a relationship `l1caloRunTypes` to the set of run type objects defined for this partition.

In the L1Calo extension to the standard DAL this is used to create a runtime `L1CaloConfdbPartition` object.

Note that having our own `Partition` subclass means that the `TDAQ_DB_SCHEMA` environment variable must contain our `l1calo_hw` schema file.

4.2 `L1CaloCrate`, `L1CaloModule` and Subclasses

The `L1CaloCrate` class is a subclass of the standard `Crate` class. `L1CaloCrate` has one extra attribute, the `backplaneType` which is an enumeration of the possible types of backplane in our system.

The DAL creates a C++ L1CaloConfdbCrate object from each L1Calo-Crate OKS object. The backplane type is implemented in a wrapped enum class BackplaneType generated by the infraL1Calo package.

The L1CaloModule class is an abstract OKS subclass of Module. which is only used as the common base class of OKS subclasses for each of our modules. It has an additional attribute ttcAddress to specify the TTCrx address of modules which dont have geographical addressing. There is also a relationship fpgaConfiguration providing a link to the set of FpgaPrograms for this module. At the time of writing the schema contains non-trivial subclasses Cmm, Cpm, CpRod, Dss, Jem, Ppm and Ttvci. Classes for other modules can be added in future, but modules with no extra attributes can just be left as L1Calo-Module objects.

In the L1Calo DAL these are used to create runtime L1CaloConfdbCmm etc C++ objects. Modules with no special OKS class of their own become L1Calo-ConfdbModule C++ objects.

4.3 DataGenRecipe, GenericFile and GenericFileList

GenericFile is just a filename. Unlike the standard DataFile it is not expected to be an XML file. GenericFileList contains a list of GenericFile objects which may all live in a default directory. DataGenRecipe is a subclass of GenericFile specifically to describe a test vector file. Apart from the filename, it has a data type and a flag identifying it as either an input file (source) or an output file (sink).

4.4 CableConnection and CableBundle

CableConnection objects link two (or more) modules which are identified by the srcModule and dstModule relationships. The connectors on each module are specified by the srcConnector and dstConnector attributes. There are a number of CableConnection subclasses for each type of cable and for each one, the appropriate connector names for both ends are enumerated In the DAL they all become CableConnection C++ objects (no L1Calo or Confdb prefix for a change!).

A CableBundle object in OKS describes a set of cables between the same two modules. The only subclasses are for LVDS cable bundles between PPMs and either CPMs or JEMs. In the DAL single CableBundle objects are converted to sets of CableBundle objects.

5 Using the Database in C++

This section describes how to use the database from C++ code.

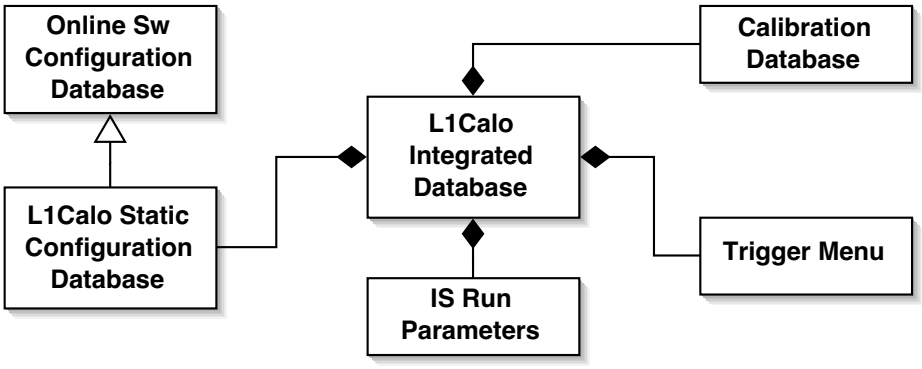

Figure 1: *The L1Calo "integrated" database brings together a number of other databases.*

5.1 Initialisation of the Database

The L1Calo database empire consists of several parts: the integrated database layer which brings together the extended standard DAL and the separate DALs for the calibration data and the trigger menu. Additionally the integrated database allows some parameters from the standard (static configuration) DAL to be overridden by run parameters taken from the Online software Information Service (IS). Figure 1 shows the general idea. All the component databases (apart from the IS run parameters) use the same "transient" initialisation scheme as that of the standard Online configuration database DAL.

Each DAL has one top level class which is responsible for reading the OKS database files and creating the runtime C++ objects. For the standard Online DAL, this class is called `ConfdbConfiguration`. For the L1Calo extension of the standard DAL, the equivalent class is `L1CaloConfdbConfiguration`, and for the L1Calo integrated database layer, the class is `L1CaloDatabase`.

The initialisation process proceeds roughly as follows. The user creates an instance of the relevant top level class and (typically) passes a list of OKS schema and data files. These are read by an OksKernel object which creates the transient OKS C++ objects.

The OKS C++ objects are very generic. The schema is used to create a set of OksClass objects, one for each class. The data files are used to create OksObject objects, one for each object in the XML files. OksClass objects can have OksAttribute and OksRelationship objects describing their attributes and relationships. OkjObject objects have OksData objects which contain the values of the attributes and relationships for that instance of the OksClass.

These very generic OKS objects are rather cumbersome to use directly in other C++ code. So they are only used to initialise the customised C++ classes which

comprise the DAL. The DAL typically contains a different C++ class for every OKS class (or sometimes just each OKS base class) in the database schema.

Once the top level DAL class has created all the runtime C++ objects from the OksObject objects, the OksKernel is deleted and consequently all the OksClass, OksObject, etc objects are also deleted. This is why they are referred to as "transient".

Note that the top level DAL object owns all the other runtime C++ database objects. If the instance of the top level DAL class goes out of scope or is deleted, the whole database is deleted.

5.2 Integrated Database

The integrated L1Calo database is initialised and provided by a class called, unsurprisingly, `L1CaloDatabase`. The class has a number of constructors, including a default constructor which is probably what you should normally use. The default constructor takes the list of OKS schema and data files from the environment variables `TDAQ_DB_SCHEMA` and `TDAQ_DB_DATA` respectively.

The following code is thus normally sufficient to create the database and load the static configuration data:

```
#include "confL1Calo/L1CaloDatabase.h"
// Create L1Calo database and read static data.
L1CaloDatabase db();
```

Remember that all the database objects you subsequently obtain from the database will be deleted if the `L1CaloDatabase` object is deleted. The run controller, for example, creates the database at the Load transition and keeps it until the Unload transition.

The `L1CaloDatabase` class has a number of methods which are described in the reference documentation (see section 5.5).

A simplified schema of the integrated L1Calo database is shown in figure 2.

5.2.1 Linking Other Databases

The above example code only loads the static configuration data using our extended version of the Online configuration DAL. But the main purpose of the integrated database layer is to provide a fairly seamless interface to other data for our modules. This data will **only** be available if you **make** it available and tell the `L1CaloDatabase` about it.

To make the volatile run parameter data stored in IS available to the `L1CaloDatabase`, you need to tell it the name of the IS server and also pass the current IPC partition:

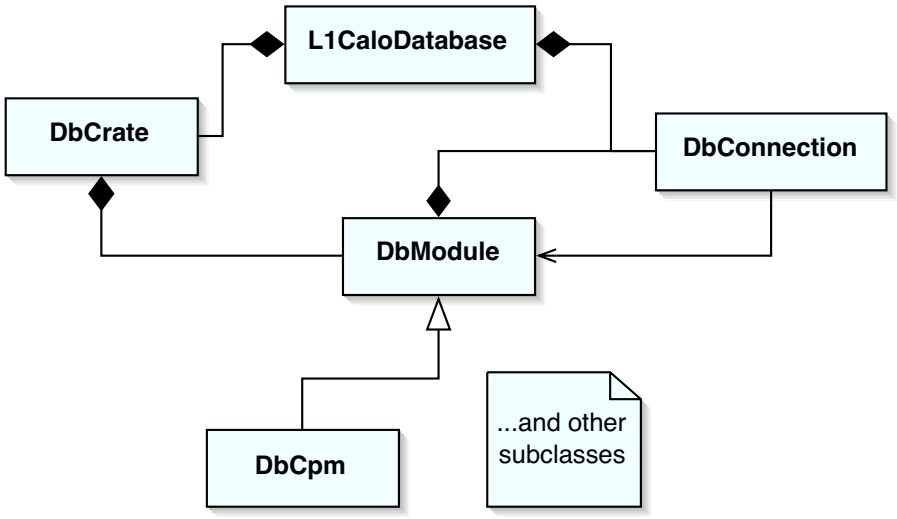

Figure 2: *Simplified schema of the L1Calo integrated database.*

```
// Read run parameters from IS.
std::string partitionName = "RalTest"; // for example
std::string isServerName = "RunParams.L1Calo";
IPCPartition ipcPartition(partitionName);
db.readIsParameters(ipcPartition,isServerName);
```

The run controller, for example, calls `readIsParameters()` at the start of each state transition action.

You may also want to provide your modules with trigger menu and calibration data. These are kept in separate OKS files which need to be read in via their own DALs (see section 5.4). The top level objects from those DALs can then be passed to the `L1CaloDatabase`.

```
#include "confL1Calo/L1CaloCalibration.h"
// Read calibration data giving schema and data files.
// NB you can just call db.setDefaultCalibration()
std::string calibSchema =
  "${L1CALO_DB_PATH}/schema/calibration.schema.xml";
std::string calibData =
  "${L1CALO_DB_PATH}/calib/myCalibration.data.xml";
L1CaloCalibration calib(calibSchema,calibData);
db.setCalibration(calib);
// Read trigger menu. Schema and data file names may
// be given, but defaults can also be taken as below.
db.setDefaultTriggerMenu();
```

5.2.2 Finding Crates and Modules

The Online configuration database allows to you to find any crate or module in the system. In constrast, the `L1CaloDatabase` expects to find a `Detector` called `L1Calo` and will only show you the crates and modules which belong to the `L1Calo` detector. The crates can be found via the `getCrates()` method. This returns two iterators (passed to the method by reference) which you can use to iterate over all L1Calo crates. You can also ask for a particular crate by name using the `getCrate()` method.

In a similar way, given a `DbCrate`, you can ask it for iterators over its collection of modules or for a particular named module. `L1CaloDatabase` also has a short cut method to find a particular named module in any L1Calo crate.

The crate and module classes have a number of methods to return their attributes. These mostly just return the attribute from the underlying static configuration database DAL. However for some attributes, the value can be taken instead from an IS run parameter variable for that module – provided that the `readIsParameters()` method has been called and the IS server is running and has the appropriate variable.

5.2.3 Finding Connections

You can find out about cable connections in two ways. The top level `L1CaloDatabase` provides a method `getConnections()` which returns iterators over all cable connections in the database. Each connection is described by a `DbConnection` object. This in turn has methods to return `DbModConn` objects, which hold the `DbModule` and its connector, for the source and destination ends of the cable.

```
L1CaloDatabase db();
L1CaloDatabase::ConnsMap::const_iterator ix;
L1CaloDatabase::ConnsMap::const_iterator ixend;
// Find all connections.
db.getConnections(ix, ixend);
for (; ix != ixend; ix++) {
  const DbConnection* cable = ix->second;
  // Get source module and its connector.
  const DbModConn* mc = cable->getSrcConn();
  const DbModule* m = mc->getModule();
  const L1CaloConnector& c = mc->getConnector();
  // etc...
}
```

Alternatively, if you already have a `DbModule` object, you can use its `getConnections()` method which also returns iterators over all the cable connec-

tions to that module. You can also use the `getConnection()` method to return the `DbConnection` (if any) for a given named connector.

```
DbModule* m;                  // Obtained earlier...
L1CaloConnector c("Glink0");
const DbConnection* cable = m->getConnection(c);
```

If you just want to find the identity of the module at the other end of the cable, you can use the `getConnectedId()` method for a named connector. This returns a `DbModCrateId` object that holds the logical (numeric) identifiers of the connected module and its crate.

```
DbModule* m;                  // Obtained earlier...
L1CaloConnector c("Glink0");
const DbModConn* mc = m->getConnectedId(c);
if (mc != 0) {
  unsigned int moduleId = mc->getModuleId();
  unsigned int crateId  = mc->getCrateId();
}
```

5.2.4 Finding Test Vectors

The input test vector and the simulated test output files are held in instances of the (misnamed) `DataGenRecipe` class which is part of the `infraL1Calo` package. Given a `DbModule`, you can find its input test vector (ie its data `Source`) or the simulated output (ie its data `Sink`) via the `getDataGenRecipe()` method and requesting the appropriate `L1IOType`.

```
DbModule* m;                  // Obtained earlier...
L1IOType src = L1IOType::Source;
const DataGenRecipe* dg = m->getDataGenRecipe(src);
if (dg != 0) {
  // Then use the DataGenRecipe to create a test
  // vector reader via the TVReaderFactory class...
}
```

5.2.5 Finding Calibration Data

Provided that the top level `L1CaloDatabase` object has been passed a top level `L1CaloCalibration` object, you can obtain the calibration data for a particular module type, and hence its submodules (if any) via the `getCalibration()` method of the appropriate `DbModule` subclass. NB this method is only implemented in the subclasses.

From the module calibration object, eg `CpmCalibration`, you can get the `TtcrxSettings` and the other settings objects for the CPM submodules.

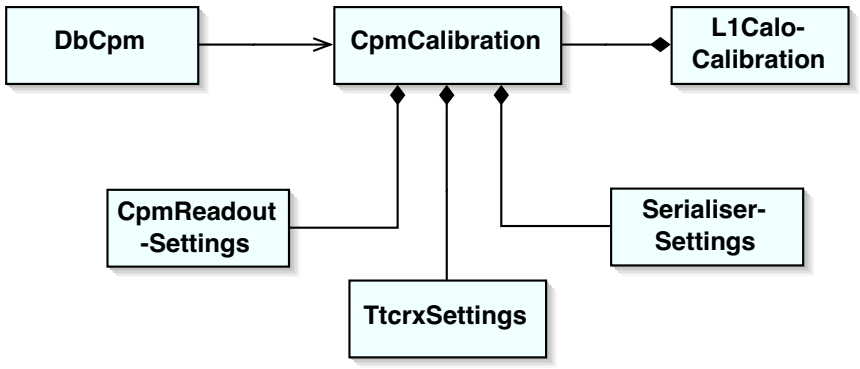

Figure 3: *Schema of the CPM calibration. The schema for other modules is similar, though the PPM is more complicated.*

```
DbCpm* m;                        // Obtained earlier...
const CpmCalibration* calib = m->getCalibration();
if (calib != 0) {
  // Get a TTCrx setting.
  int ttcPhase1 = calib->getTtcrxSettings()->getPhase1();
  // Get a readout setting.
  int daqOffset = calib->getReadoutSettings()->getDaqOffset();
  // Get a setting for each serialiser.
  for (int = 0; i < CpmDefs::numSerialisers(); i++) {
    int delay = calib->getSerialiserSettings(i)->getDelay();
  }
}
```

Note: it is important to check that a valid `CpmCalibration` object is returned. This will be zero if no `L1CaloCalibration` was passed to the `L1Calo-Database`.

The CPM calibration schema is shown in figure 3.

The JEM and CMM calibration objects are similar to that of the CPM, though the CMM one is rather simpler. Calibration objects for the PPM have not yet been implemented.

5.2.6 Finding Trigger Menu Data

As with calibration data, the trigger menu data for a given module is also obtained via the appropriate `DbModule` subclass. This also requires that the `L1Calo-Database` object has already been given a `LVL1TriggerMenu` object containing the whole of the trigger menu.

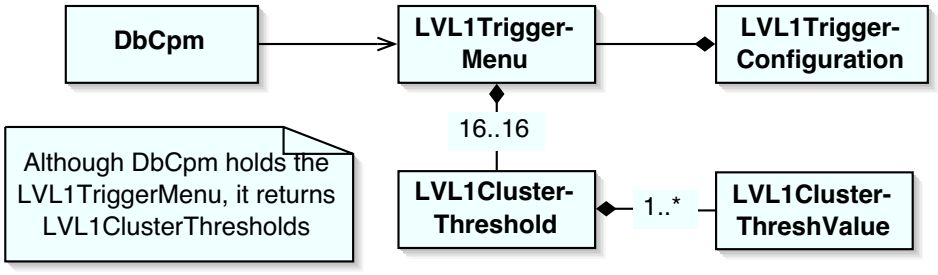

Figure 4: *Schema of the trigger menu for the CPM. The schema for the JEM is similar, but with jet, forward jet and the global energy thresholds.*

The `DbCpm` class returns a `LVL1ClusterThreshold` object for each threshold and actual threshold values for each CpChip (which may in principle be different) can be obtained from that. The structure of the trigger menu objects is described in a bit more detail in section sec:triggermenudal.

```
DbModule* m;                    // Obtained earlier...
const LVL1ClusterThreshold* thresh;
int n = LVL1TriggerMenuDefs::NumClusterThresholds;
for (int i = 0; i < n; i++) {
  thresh = m->getClusterThreshold(i);
  if (thresh != 0) {
    // Get actual values for a given CpChip
    // using its (phi,eta) coordinate.
    // But for the moment, just use (0,0).
    unsigned int cluster = thresh->clusterThreshold(0,0);
    unsigned int emIsol  = thresh->emIsolThreshold(0,0);
    unsigned int hadIsol = thresh->hadIsolThreshold(0,0);
    unsigned int hadVeto = thresh->hadVetoThreshold(0,0);
  }
}
```

Note: it is important to check that a valid `LVL1ClusterThreshold` object is returned. This will be zero if no trigger menu was passed to the `L1CaloDatabase`.

The trigger menu schema, as seen by the CPM, is shown in figure 4.

5.3 L1Calo Static Configuration Database

Users of the L1Calo database software should, in general, not have to use the classes which form our extension to the standard DAL. In principle everything

in this DAL is made available via the integrated database layer (section 5.2). So the descriptions here will be brief.

The Online configuration database DAL is intended to make a fairly automatic conversion of OKS objects to C++ runtime objects. It is not completely straight forward as it provides some extra algorithms for specific use cases. There are also some cases where unidirectional relationships in the OKS database are reversed in the DAL.

The Online `confdb` package provides a number of classes which all start with the `Confdb` prefix. The whole database is read in via the `ConfdbConfiguration` class.

The L1Calo extensions (almost?) all start with the rather verbose `L1Calo-Confdb` prefix. In most cases these classes have identical attributes to the OKS classes and make them all available via `getXXX()` methods.

About the only exception is in the handling of the cable connections which are rather complicated. I hope to change this in future (in fact they may be taken on by the Online group and implemented in the standard DAL) so I wont bother describing it all here.

5.4 Calibration and Trigger Menu DALs

At the moment, calibration data and the trigger menu are both read from OKS files. In both cases the data is read separately from the configuration data, so that different calibrations and trigger menus can be used in successive runs with the same configuration.

The calibration data and trigger menu each has a little DAL of its own. They both follow the same principle as the configuration database DAL. Transient OKS objects are read from the data files and used to initialise C++ classes.

The two DALs each have a top level class, equivalent to the `L1CaloCalibration` or `L1CaloConfdbPartition` class in the other DALs. In the trigger menu DAL this is `LVL1TriggerConfiguration`, and in the calibration DAL it is `L1CaloCalibration`. Both of these classes can be constructed either directly by specifying an OKS schema and data file, or by passing an existing `Confdb` object.

The trigger menu is expected to be contained in a single OKS file. However the calibration data is implemented in a similar way to that of the configuration database in that there is a top level file containing a `Partition` object which can specify other files to be included. This scheme allows different types of calibration data, eg energy, timing, BCID, which may change on different timescales, to be stored in separate files. However the final system will probably actually obtain data from the ATLAS Conditions Database, so this OKS based scheme may not be used.

In principle the federated scheme for the calibration data should allow data files to handle a single crate or a collection of crates. However this has not yet been tested.

5.4.1 Trigger Menu

To read in the trigger menu, you need to create an instance of the DALs top level class, `LVL1TriggerConfiguration`. From this object you can get the trigger menu itself. This is a single object of type `LVL1TriggerMenu`. The trigger menu object can then be queried to find the 16 `LVL1ClusterThreshold` objects. There is one of these for each threshold. They cover the whole eta-phi space, however different eta-phi subspaces may require different values of the various thresholds. The actual threshold values for a specific (phi,eta) coordinate is held in a `LVL1ClusterThreshValue` object.

Similar classes exist for the jet and global energy thresholds.

Further details are in the reference documentation (see section 5.5) and look at section 5.2.6 for some examples.

5.4.2 Calibration Data

Calibration data is read in and run time calibration objects are created by the `L1CaloCalibration` class. Once you have created an instance of `L1Calo-Calibration`, you can query it to obtain the module calibration object for a named module.

Further details are in the reference documentation (see section 5.5) and look at section 5.2.5 for some examples.

5.5 Reference Documentation

Fairly complete reference documentation on the API of all classes in the `confL1Calo` package is generated using Doxygen every night as part of the nightly build.

All the `Db`-prefixed classes have at least a brief description for every method. The underlying `L1CaloConfdb`-prefixed classes have descriptions for important methods, but some methods are inlined in the header files without special Doxygen style comments.

The reference documentation is available at
`http://www.hep.ph.qmul.ac.uk/l1calo/dox/confL1Calo/html`

6 Using the Database in Java

The DAL described in section 5 is only available in C++. In future the configuration database package may provide tools to generate low level DALs automatically

from the schema in both C++ and Java. Until then, the database can only be accessed in Java via the remote database server (RDB and RDBplus). The method of doing this is rather laborious and is poorly documented by the Online group.

This section of the present document may be expanded in future. In the mean time, if you are really interested, you can look at the classes `L1CaloModPars`, `L1CaloRunPars` and `RdbUtilities` in the L1Calo `iguiL1Calo` package.

7 Editing the Database

The Online software provides a number of tools for editing the database. The most convenient and graphical one is extendable and has been extended to display L1Calo specific subclasses of the standard classes and also to handle entirely new L1Calo classes.

7.1 Graphical Database Editor

7.1.1 Invoking the Editor

The graphical database editor provided by the Online software is called `confdb_gui`. It can be used directly, but it will normally be more convenient to invoke it via our `edconf` wrapper script. The main reason for this is that our top level partition files use other data files which are expected to be found in the `${L1CALO_DB_PATH}` tree. This normally points to the read-only installation area. In order to edit the writeable files in the dbFiles package area, the `L1CALO_DB_PATH` environment variable needs to be reset to point there. This is done by the `edconf` script.

To invoke the editor via `edconf` you need to specify the top level partition file to be edited. Any files this uses will be opened automatically. Assuming you are working within the `dbFiles` package you need to do:

```
cd partitions
edconf -p MyPartition.data.xml&
```

Note that this file must already exist. To create a new partition, copy and rename an existing file.

7.1.2 Usage Summary

The Online `confdb` package user guide [4] describes how to use `confdb_gui`. Only the L1Calo extensions will be described in any detail here. But first, here is a very brief summary of how to use the editor.

At startup, the editor displays two windows. One shows the set of schema and data files which have been opened, the other shows error messages.

To edit the database, use the `Edit` menu and open one of the editing windows. The `Partitions` window shows the top level structure. The `Hardware` window is used to edit crates, modules and also the L1Calo "hardware" additions such as cables. The `Software Repository` window can be used to add new programs. The `Run Control` window is used to set up the run control hierarchy, but confusingly some aspects are also shown in the `Partitions` window which must also be used to add the run control applications to the partition object.

The `Test Vectors` window is an L1Calo addition for describing the test vector input and output files.

Before adding new objects (via the right mouse button popup menu) you have to select an existing database file (or create a new file) using the menu at the bottom of each window.

Remember that the editor uses the right mouse button a lot. Most editing is done via popup contextual menus accessed via the right button. The middle mouse button is used to make links between objects. Press the middle button over an object and drag it to another object to establist the relationship. Choose the appropriate relationship from the popup menu.

7.1.3 L1Calo Extensions

When creating a new partition (in the `Partitions` window), be sure to select `L1CaloPartition` from the menu. This allows the partition to have links to lists of test vector and trigger menu files.

The menus in the `Hardware` window for creating crates and modules also contain options for creating `L1CaloCrates` and subclasses of `L1CaloModule` for each of our module types.

The `L1CaloCrate` class has an additional attribute to set the backplane type. This is necessary for the cluster and jet/energy processor crates so that the DAL can set up VME addresses automatically using the geographical addressing algorithm. This also requires that each module has its `slot` attribute set correctly.

7.1.4 L1Calo Additions

In addition to subclasses of standard Online configuration database classes, there are also some entirely new L1Calo specific classes. These are easily spotted by their very mundane black and white icons.

In the `Hardware` window, you can create `CableConnection` and `CableBundle` objects each of which can be of several different subclasses.

A `CableConnection` object typically describes a single cable between specific connectors on two modules. Actually it can be used to describe so called "octopus" cables with multiple ends as well, though this feature has not been properly tested.

To do this, create a CableConnection and give it a name which reflects the two modules it connects. Link it with its source and destination modules by dragging the module objects onto the cable object and selecting the appropriate relationship. Set both the srcConnector and dstConnector attributes to the appropriate connectors on each module.

A CableBundle object may be used to describe a set of cables between the same two modules. This is only used for LVDS cables from PPMs to CPMs or JEMs. The CableBundle is only provided for convenience of editing the database. At run time each CableBundle is converted to groups of CableConnection objects.

The Test Vectors window can be used to add DataGenRecipe and GenericFileList objects containing the available trigger menus and calibration files. However this window (like the other windows) will, by default, only show objects which are not used by other objects (depending on the display option chosen via the right mouse button menu). If the objects are already linked, you can see them via the relationships of the module or partition objects.

Once you have created a DataGenRecipe, you can link it with its relevant module(s) by dragging the module to the DataGenRecipe object using the middle button. NB you also need to link each DataGenRecipe to the L1CaloPartition.

You can build up GenericFileLists by creating GenericFiles and dragging them to the GenericFileList you created earlier. The GenericFileList of either calibrations or trigger menus can then be linked to the L1CaloPartition.

7.2 OKS Data Editor

An alternative way to edit the data files is to use the basic OKS data editor. This is a generic editor which, unlike the confdb_gui, has no knowledge of the ATLAS database schema. The user of the tool has to know the schema. It is not extendable, so the complete description can be found in the confdb user guide [4].

7.3 OKS Schema Editor

This is the only tool for editing the schema files (apart from editing the XML files by hand). Like the OKS Data Editor, it is a generic editor and has no knowledge of the ATLAS database schema. Refer to the confdb user guide [4] for further details.

A graphical schema editor is promised for a future Online software release but is not available yet.

7.4 Editing XML By Hand

Although the graphical editor is normally the most convenient and safest method for editing the databases, it can sometimes be useful to edit the XML files by hand. For example if you need to add many instances of a particular class it can be quicker to do the first one using `confdb_gui` to establish the correct syntax and then add the rest by cutting and pasting with a text editor.

The drawback of this approach is that it is easy to introduce syntax errors into the XML file. Since XML parsers are expected to be very unforgiving of errors, this can render the file unusable and unreadable by the standard tools so that you can not use them to correct errors you have introduced.

To ameliorate this problem, the `dbFiles` package includes a `checkxml.pl` script to perform checks for the most common errors introduced by manual editing. It will point out what you have to fix.

7.5 Naming Conventions

Each object of the same class in the database must have a unique ID. Eg the CPMs in slot 6 of crates 1 and 2 must have different IDs. So it is necessary to use a hierarchical set of prefixes, adding the crate name to the module name, and appending submodule names if appropriate. See also the discussion in [2].

8 Database Utilities

The `confL1Calo` package provides a number of utilities – in addition to those provided by the Online software `confdb` package. They are all described briefly here.

All the command line parameters should be listed here someday. Until that happy time, you can find them by giving `-h` or `--help` on the command line to each program.

8.1 `calib_dump`

This provides a simple dump of a named calibration database or (by default) the one referred to by the L1CaloDatabase.

8.2 `confdb_evolve`

This program just reads and rewrites all the OKS files named on its command. This has the (desired) side effect of bringing the files up to date with any changes in the schema.

It is most convenient to run this program via the `confdb_evolve.pl` script which will keep all the objects in the file in the same order and reports which files have changed and need committing back to CVS.

8.3 `confdb_setdata`

A utility program to set a specified attribute of all instances of a particular class to a given value or to show its present value. It may be useful in filling calibration databases.

8.4 `initcalib`

This program creates or updates a calibration database. It ensures there are calibration objects appropriate for the crates and modules found in the default configuration database (ie `TDAQ_DB_DATA`). Any existing calibration objects will be kept and new objects created where necessary, eg if new modules were added to the configuration. Note that all the attributes of **new** objects are set to zero (or whatever the default initial value specified in the schema is).

8.5 `l1calo_confdb_dump`

This is an extended version of standard `confdb_dump` which also dumps new L1Calo classes and extra attributes of standard classes.

8.6 `l1calo_db_dump`

A utility to dump the integrated L1CaloDatabase including where static configuration data may be overridden by IS variables. At present it only handles crates and modules.

8.7 `trigedit`

This is a graphical trigger menu editor implemented with Qt. This program is **incomplete** and hence not very useful. It will probably be removed when the CTP group provide a proper trigger editor tool.

9 Examples

There are a number of examples scattered through the document, mostly in section 5. Should there be more?

For the moment though, you will have to look through existing code if the above examples are not enough. A good place to start is the `DbSim` package, especially `DbSimulation`. Also various of the module services packages.

10 To Do...

Things to do for this document (and related documents):

- Update the database design document [2], especially its diagrams.
- Incorporate some of its diagrams into this document?
- Create some (more) new diagrams for this document as well?
- Pictures illustrating use of the `confdb_gui`.
- More examples?
- More details of the utility programs?

References

[1] ATLAS Level 1 Calorimeter Trigger: home page
`http://hepwww.pp.rl.ac.uk/Atlas-L1`

[2] Configuration Database
`http://www.hep.ph.qmul.ac.uk/l1calo/doc/pdf/ConfigDatabase.pdf`

[3] ATLAS Online Software Configuration Databases
`http://atddoc.cern.ch/Atlas/DaqSoft/components/configdb`

[4] Configuration Databases User Guide
`http://atddoc.cern.ch/Atlas/Notes/135/Note135.pdf`