ATLAS Level-1 Calorimeter Trigger 22 August 2001
Software Note 008 Version Draft 0.3

Software Organisation

L1Calo Group [1]

1 Introduction

This note makes some suggestions for organising our software development for the slice tests and the final system.

It covers division of our software into packages, the code repositories, documentation, etc.

2 Software Packages

Figure 2 shows the decomposition of the L1Calo online software into a number of separate packages and shows the network of dependencies between them.

Many of our packages have dependencies on external software, mostly the packages of the ATLAS Online software, but also Dataflow software, other LVL1 software, histogramming packages such as ROOT, etc.

The scope of each of our packages is detailed in the following sections. For each package we generally give the kind of classes it will contain, which binary applications it provides and its external dependencies.

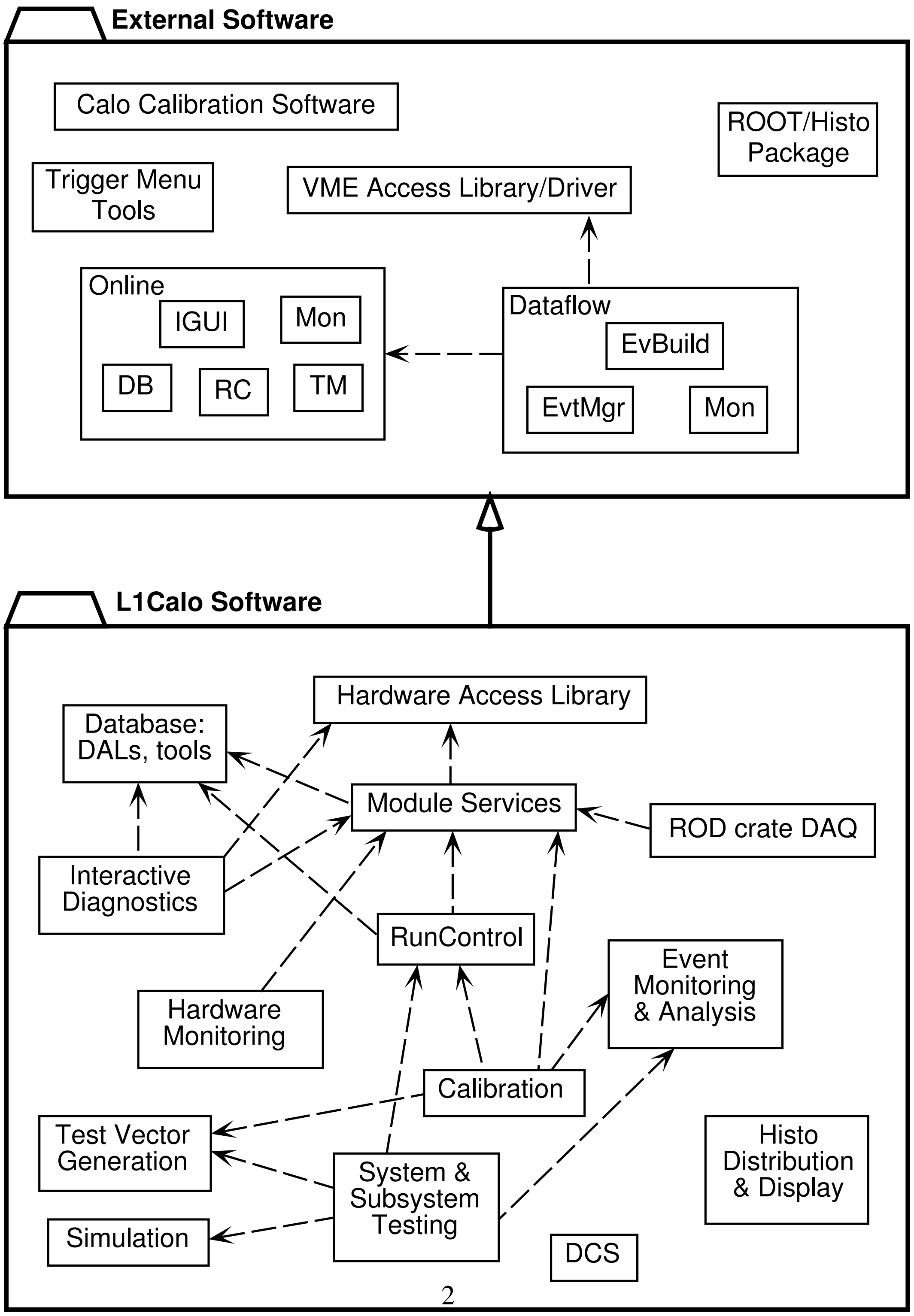
Experience from the Online group and from the software industry suggests that the general principle of assigning one person to be responsible for a given package or software component works best.

So far, with HDMC and our earlier software, we have generally had a less formal approach. There has usually been few enough of us working on any part of the software to cope with a free for all.

We probably dont want to become too formal, however it may be sensible to assign individuals to packages. Any changes to those packages, ie new code or bug fixes committed to CVS by other people should at least be notified, preferably in advance, to the person looking after that package.

[1]Please send any comments and corrections to Murrough Landon.

L1Calo software packages and external libraries



L1Calo package diagram.

2.1 Databases

This package contains database schema, database classes, data access libraries (DALs) and standalone utilities for defining, editing and using the various databases we need.

The three main databases are the Online hardware and software configuration database; the trigger menu description; and our calibration data.

For the hardware and software configuration, there will be:

- an extension of the core schema of the Online software to define OKS database classes which describe our system. For example we need Module subclasses and some classes describing the connections between Modules
- extensions to the recently enhanced Online configuration database GUI to allow us to edit and display our OKS database classes
- a data access library comprising a set run time classes and additional software to initialise them from the OKS database classes

For the trigger menu, we need to:

- define the (or other?) database classes for the L1Calo part of the whole ATLAS trigger menu schema
- provide a suitable set of run time classes
- contribute to developing a DAL to read initialise the run time classes from the database
- contribute to writing the tools for creating, editing and displaying trigger menus

Initially, for the slice tests, we will probably do this ourselves. Subsequently this part of our database package may perhaps be combined into a single LVL1 trigger menu package?

For our calibration data, we need to:

- define classes (or data structures) for our calibration data
- provide code to save and retrieve them
- provide code to do simple editing or generation of simple dummy sets of calibration data

Our database package has dependencies on the OKS and Confdb packages of the Online software and (eventually?) overall LVL1 trigger menu software.

2.2 Hardware Access Library

The Hardware Access Library package provides the sort of services most of which are currently encapsulated in the hardware Parts of HDMC (augmented by some necessary modifications described in a recent document). These include:

- a layer of VME classes, insulating us from choices of the DAQ group VME library or other VME drivers
- a suite of low level classes modelling basic generic components such as Registers, Memories, FIFOs
- classes describing some of our specific, but still fairly basic components, eg simple chips
- a set of less basic classes handling the common features of complex components, eg FPGAs
- a generic Module class

This package depends on the ATLAS DAQ VME library and/or VME drivers such as Juergen Hannappels driver or the proprietary one recently provided by Concurrent.

The intention is that this package has as few dependencies as possible so it can be used in basic standalone test and diagnostic programs, eg the current HDMC GUI, without requiring too much extra support infrastructure.

So the higher level behaviour of modules and complex subcomponent is included in a separate package. In particular The Hardware Access Library does not include any classes which take complex database objects (eg calibration classes or data structures, trigger menu classes, etc) as arguments.

2.3 Module Services

The Module Services package contains classes which implement the high level functions of our modules and their main, complex subcomponents such as the PPrAsic and the FPGAs in the algorithm processors.

The kind of functions envisaged include things like:

- load thresholds into all the CPchips on a CPM by passing a suitable thresholds object obtained from the database
- load PPrAsic lookup tables using calibration objects
- get an event fragment from a spy buffer on a ROD

The classes in this package will use the lower level classes of the Hardware Access Library. This package also depends on the Database package.

2.4 Interactive Diagnostics

This package contains a collection of GUI and line mode programs for detailed debugging, display and control of the hardware from the lowest level at least up to single module level.

The HDMC GUI code already provides a lot of the required functionality though again more changes and development are needed.

A simple GUI might just be able to display the system in terms of objects from the Hardware Access Library. However it may also be useful to have an enhanced version which can access higher level functions of the Module Services package.

2.5 Run Control

The Run Control package contains the classes and applications which form our interface with the ATLAS run control system.

There are two main aspects involved:

- code which implements the local run controllers for the various types of crate in our system. These are responsible for correctly configuring all the modules in each crate using data read from our various databases.
- code in the run control GUI (java) to edit and display L1Calo specific run parameters (eg calibration types etc) and to display L1Calo run status information.

Our Run Control package clearly depends on and extends the run control and IGUI packages from the Online software. It uses our Database and Module Services packages to execute the appropriate actions for each module at each state transition.

Some aspects of this package, eg the actions for calibration and test runs, may be split off into the separate Calibration and System Testing packages.

2.6 ROD Crate Readout

This package is responsible for collecting event fragments from our RODs (and perhaps in some circumstances from test modules, eg DSS). It should make these fragments available for monitoring, using the standard monitoring skeleton package of the ATLAS DAQ software.

The extent to which this package will be integrated into (or provided by) software provided by the ATLAS Readout System (ROS) is still unclear.

We will presumably use our Module Services package to collect event fragments from our RODs. We will probably use some packages from the ROS group to build crate fragments from several RODs in the same crate and to build events from fragments read from several crates.

2.7 Hardware Monitoring

This package contains the programs (or threads?) which will run in each crate CPU monitoring the modules in that crate via the VME bus. Typically they will check for link errors, read the rates histograms from the PreProcessor system, check that the modules are still loaded correctly etc.

These programs should store or publish the information they collect, eg using the Online Information Service. Other programs in this package will collect this information from our distributed system and display it at user request.

Note that this package does not include any aspects of monitoring carried out by DCS via the CAN bus, though it may perhaps log its information to DCS. Nor does it include monitoring of the readout data – that is the domain of the Event Monitoring package below.

The programs in this package will use the Module Service package.

2.8 Distributed Histogram Package

We expect to have processes in multiple crates and PCs all busily making histograms. There is likely to be too much data to ship around all the time. Instead the data should only move on explicit request by a user. So histograms (and tables and other data) are stored in the local system, but the display must know where to find the data when needed and be able to collect it.

The CDF package adapted for us by Tara provides this functionality, though a serious memory leak was never fixed and there is probably room for other improvements. Alternatively, the Level 2 people have similar requirements but seem set on developing a slightly different solution. If that becomes an ATLAS standard we should probably use it instead.

2.9 Event Monitoring

This package includes programs with monitor the system by analysing events and event fragments read out through the normal DAQ path. There may be many such programs, some running during all normal runs, some used only for debugging, some for checking the results of feeding test vectors into the system, some for processing data from calibration runs and so on. However the latter categories might be split off into the separate Calibration and System Testing packages.

This package also includes our system specific extensions to the java event dump package in the Online software. And the C++ programs here could use any Event ”unbuilding” classes if they were in future provided by the Online software.

2.10 Test Vector Generation

This package contains a suite of programs, ideally within a single overarching framework, to generate test vectors for all parts of the system: single modules, groups of the same module type, small subsystems etc.

2.11 Simulation

The Simulation package is a library of classes allowing the system, or subsets of it, to be modelled at various granularities. Specifically it allows an input set of test vectors to be propogated through the system to predict what should be read from subsequent stages in the trigger and readout chain.

2.12 System Testing

The System Testing package contains the framework necessary for carrying out detailed tests of the whole system or subsystems.

It uses the facilities of the Test Vector Generation and Simulation packages. It may carry out some tests using the run control system and check for correct operation by analysing events collected through the DAQ monitoring system. However other tests may be carried out by standalone programs. Some of these may be suitable for use by the Test Manager and Diagnostic Verification System on the ATLAS Online software.

2.13 Calibration

The Calibration package brings together the programs used to carry out our various calibration procedures.

These may be implemented using the run control and analysis of events. However some may be done using standalone programs.

2.14 DCS

Our DCS package is intended to include any of our software which runs in the ATLAS DCS domain, ie from the CAN bus to the higher level SCADA system.

3 CVS Repositories

In the long term we will probably want to move our software repositories to CERN. The Online group have even mentioned the idea of a common repository for all online software, though this may not be practical.

For the moment though, it will be more convenient to keep our existing CVS repositories at Heidelberg (for HDMC) and RAL. These are both accessible by the CVS pserver and the RAL repository can also be accessed via AFS.

I suggest that the future work on our Online software be kept for the moment on the RAL CVS repository under a new directory. Other packages such as histogramming and the various previous developments such as the UK diagnostics, old DAQ system, ProZAQ tests, etc should perhaps remain outside a new L1Calo specific area. Our firmware should also be kept separately. A suggested initial directory structure is shown in table 1. Some of the directory names have been selected to be the same as those for the corresponding packages in the Online software. This may or may not be a good idea.

Should include a table for the Heidelberg repository?.

3.1 Use of CVS

Apart from the CVS manual [1], we have a few web pages [2] which summarise how to use CVS and the pserver.

They briefly mention the procedure for importing a new package into the repository. In more detail:

- create a clean directory tree (eg `$HOME/mypackage`) containing only those subdirectories and files from the package to be imported into CVS.
- define CVSROOT environment variable to point to the RAL CVS repository. For importing new packages into the repository this may need to be via AFS not the pserver?
  ```
  export CVSROOT=/afs/rl.ac.uk/atlas/groups/level1/Repository
  ```
- login to the RAL AFS if required
  ```
  klog username@rl.ac.uk
  ```

Directory	Package
`$CVSROOT/l1calo`	Root directory for new software
`$CVSROOT/l1calo/calib`	Calibration programs
`$CVSROOT/l1calo/confdb`	Database classes and utilities
`$CVSROOT/l1calo/database`	Database schema and datafiles
`$CVSROOT/l1calo/doc`	Documentation (not package specific)
`$CVSROOT/l1calo/ed`	Event dump (or add ons to Online package)
`$CVSROOT/l1calo/evmon`	Event monitoring and analysis
`$CVSROOT/l1calo/hwmon`	Hardware monitoring programs
`$CVSROOT/l1calo/igui`	IGUI panels
`$CVSROOT/l1calo/modserv`	Module services (unless with HDMC?)
`$CVSROOT/l1calo/rc`	Run controller(s)
`$CVSROOT/l1calo/readout`	ROD crate readout
`$CVSROOT/l1calo/sim`	Simulation package
`$CVSROOT/l1calo/testvec`	Test vector programs
`$CVSROOT/firmware`	Root directory for firmware
`$CVSROOT/*`	Other directories for old stuff

Table 1: RAL CVS repository directory structure

- `cd $HOME/mypackage`
- Finally create the new package, called eg "`mypackage`" as a subdirectory of l1calo. The so called "vendor tag" and version (`start` and `v000` below) are arbitrary.
- `cvs import -m "First CVS version" l1calo/mypackage start v000`
- cd to a new empty directory and verify that you can check out the new package:
  ```
  mkdir -p /tmp/workdir
  cd /tmp/workdir
  cvs co l1calo/mypackage
  ```
- The package should have been checked out into `/tmp/workdir/l1calo/mypackage`

4 Structure and distribution of packages

4.1 Package distribution

In general, within each previous package we have had `src` and `doc` directories for source code (including header files) and documentation (where that exists). The `src` directory contained a `Makefile` and, unless the package was very small, subdirectories for groups of source files, eg a subdirectory `Mains` for main programs. When building the software, we have in the past created `obj`, `lib` and `bin` directories within the directory structure extracted from CVS.

However, with several packages, we should probably change to a system similar to that used by the ATLAS offline and online software groups.

Although no final decision has yet been taken, the offline software community will at some point move to a replacement for their present Software Release Tools (SRT) package. The choice is expected to be the CMT package which is used in a few other HEP experiments. We should then follow whatever conventions are expected by the new package.

Typically, with many packages maintained by different people, the aim is that the developer of one package should be able to use precompiled libraries and binaries (and include files) from distributions of other packages. The libraries and binary programs from all packages are made available in a single tree with subdirectories for different machine architectures. The include files are also in a single tree, but with a separate subdirectory for each package. The layout of the distribution is indicated in table 2.

Directory	Comment
`$DISTROOT/bin/`*arch*`/`	Binary programs for all packages
`$DISTROOT/lib/`*arch*`/`	Libraries (*.so) for all packages
`$DISTROOT/include/`*package*`/`	Include files for each package
`$DISTROOT/share/`*package*`/`	Data/help files for each package

Table 2: Directory structure of installed software distribution

4.2 Internal package layout

The implications of that organisation for each package, as seen by the developer are:

- to include header files from other packages you need to
  `#include "`*package*`/`*header*`.h"`
- this means that the same style must also be used within each package – at least for header files which are to be made externally visible in the distribution.
- thus each package must store (or link) its externally visible header files in a directory `src/`*package*

This means that the layout of each individual package should be similar to that shown in table 3.

Directory	Comment
package`/Doxyfile`	Control file for Doxygen
package`/Makefile`	Makefile for the package (unless using CMT)
package`/cmt/`	CMT control directory (if using CMT)
package`/doc/`	Package documentation
package`/share/`	Data etc files to be included in the distribution
package`/src/`	Overall source directory
package`/src/`*package*	Single directory for all externally visible header files
package`/src/`*subdir*	Subdirectory for selected classes (and their headers if not externally visible) if the package is large enough to warrant subdivision of the `src` directory

Table 3: Directory structure of a package

5 Software Development Process

A fairly light software process is being proposed within the Trigger/DAQ/DCS project. We are expected to follow this process for development of new software.

5.1 Overview

The proposed process involves a few identified steps, some of which are marked by production and review of documents. Typically the process for development of a new package starts with some brainstorming which should elucidate the requirements of the package. The requirements should be captured in a document which can be reviewed. Given the requirements, a high level design can be proposed (in a document) and reviewed. At the same time, a test plan for the package should be devised.

The package, and additional code used to test it, can now be implemented and a user guide written. The package should then be tested, first alone and then together with the rest of the system. Ideally a test report should be written.

The process is iterative: each stage may generate feedback into previous steps, eg overlooked requirements, etc.

5.2 Application within L1Calo

This software process, even though fairly light by offline standards, is more formal that we have been used to. In particular we have not tended to write many documents describing our existing packages.

It seems reasonable to try to follow this process in future. Though the formality may vary depending on the size of the package or development. For a new package, we should write and review requirements and high level design documents. For small extensions to existing packages though, perhaps a summary of the new requirements being addressed and the proposed implementation would be sufficient. These could just be circulated by email.

6 Documentation

Given the long timescales of the project, good documentation is going to be essential. Following the software development process described above, we should produce a requirements document, a high level design document and a user guide for each package.

In addition to those, it will be useful to establish a standard mechanism for extracting and generating detailed reference documentation from the code.

HDMC has made good use of the Doxygen [3] tool. Doxygen has also been used in the LHCb Gaudi package adopted as the basis for the ATLAS offline framework ATHENA. It therefore seems a good choice for documenting our software.

The ATLAS Online group (and others) have also used the LXR tool for cross referencing source code. There is some overlap (may not be complete?) with what is offered by Doxygen. To be investigated.

7 Coding Standards

The ATLAS offline community has proposed sets of coding and naming rules for both C++ and Java code. We should generally follow these for new code. For old code, it is probably best to keep using any existing conventions used within that code, though perhaps major extensions or revisions of existing code should be used as an opportunity to adhere move towards the standard rules.

The latest version of the C++ coding standards is at
`http://www.cern.ch/Atlas/GROUPS/SOFTWARE/OO/qc/C++_Coding_Standard.pdf`
and some standards for Java are available via
`http://www.cern.ch/Atlas/GROUPS/SOFTWARE/OO/tools/java`.

Many of the rules are common sense. The most salient ones include:

- exactly one class per file (ie source and header in C++) with the same name, including case, as the class it contains.
- extensions `.cxx` and `.h` for source and header files
- class names (also structs) start with an uppercase letter
- variables and functions start with a lowercase letter
- normal and static member data start with `m_` and `s_` respectively
- form names by concatenating capitalised words without underscores, converting all uppercase acronyms to lowercase first, eg `LoadCpmRoc`.
- avoid using the C preprocessor
- try to avoid multiple inheritance and friends

References

[1] CVS documentation
`http://wwwinfo.cern.ch/asd/cvs`

[2] Use of our RAL and Heidelberg CVS repositories
`http://hepwww.pp.rl.ac.uk/Atlas-L1/Software/cvs_uk.html`
`http://hepwww.pp.rl.ac.uk/Atlas-L1/Software/cvshints.html`

`http://hepwww.pp.rl.ac.uk/Atlas-L1/Software/cvs_pserver.html` `http://hepasic.kip.uni-heidelberg.de/atlas/support/cvs.html`

[3] Doxygen home page
`http://www.stack.nl/∼dimitri/doxygen`