ATLAS Internal Note

DAQ-No-87

April 14 1998

ATLAS DAQ

Back-end software

High-Level Design

Issue: Draft
Revision: 1

Reference: ATLAS-DAQ-BE-HD
Created: 14 April 1998
Last modified: 14 April 1998

Prepared By: ATLAS DAQ Backend Software Group

Abstract

A summary of the high-level design for the ATLAS [1] Data Acquisition System (DAQ) back-end software is presented. This is the basis for the detailed-design and implementation of the back-end software which covers the needs of the ATLAS DAQ prototype named “-1”. The software described in this document is the work of the ATLAS back-end DAQ sub-group including the following people:

CERN: D.Burckhart, M.Caprini (on leave from Institute of Atomic Physics, Bucharest), R.Jones, L.Mapelli, A.Patel (now with Alcatel, Brussels), I.Soloviev (on leave from PNPI, St Petersburg), T.Wildish

CPPM, IN2P3-CNRS, Marseille: L.Cohen, P.Y.Duval, R.Nacasch, Z.Qian

Institute of Atomic Physics, Bucharest: E.Badescu, A.Radu

JINR, Dubna: I.Alexandrov, V.Kotov, K.Rybaltchenko

University of Geneva, Geneva: Lorenzo Moneta

LIP, Lisbon: A.Amorim, H.Wolters

NIKHEF, Amsterdam: H.Boterenbrood, W.Heubers. R.Hart

University of Sheffield, Sheffield: S.Wheeler

PNPI, Gatchina, St. Petersburg: S.Kolos, Y.Ryabov

1 Overview

This document is a summary of the high-level design of for the ATLAS DAQ Back-end software. This design is intended to satisfy the requirements defined in the User Requirements Document [2]. The intended audience are project reviewers and developers of the back-end software. It is the product of a ATLAS DAQ Back-end software group and is drawn from the more detailed individual component design documents. A description of the software process used for the development of this software and testing plans are also included.

The back-end software encompasses all the software to do with configuring, controlling and monitoring the DAQ but specifically excludes the management, processing or transportation of physics data. Figure 1 is a basic context diagram for the back-end software showing the exchanges of information with the other sub-systems. This context diagram is very general and some of the connections to the other sub-systems may not be implemented in the prototype DAQ/Event Filter Prototype "-1-" project.

Figure 1 DAQ Back-End software context diagram showing exchanges with other sub-systems

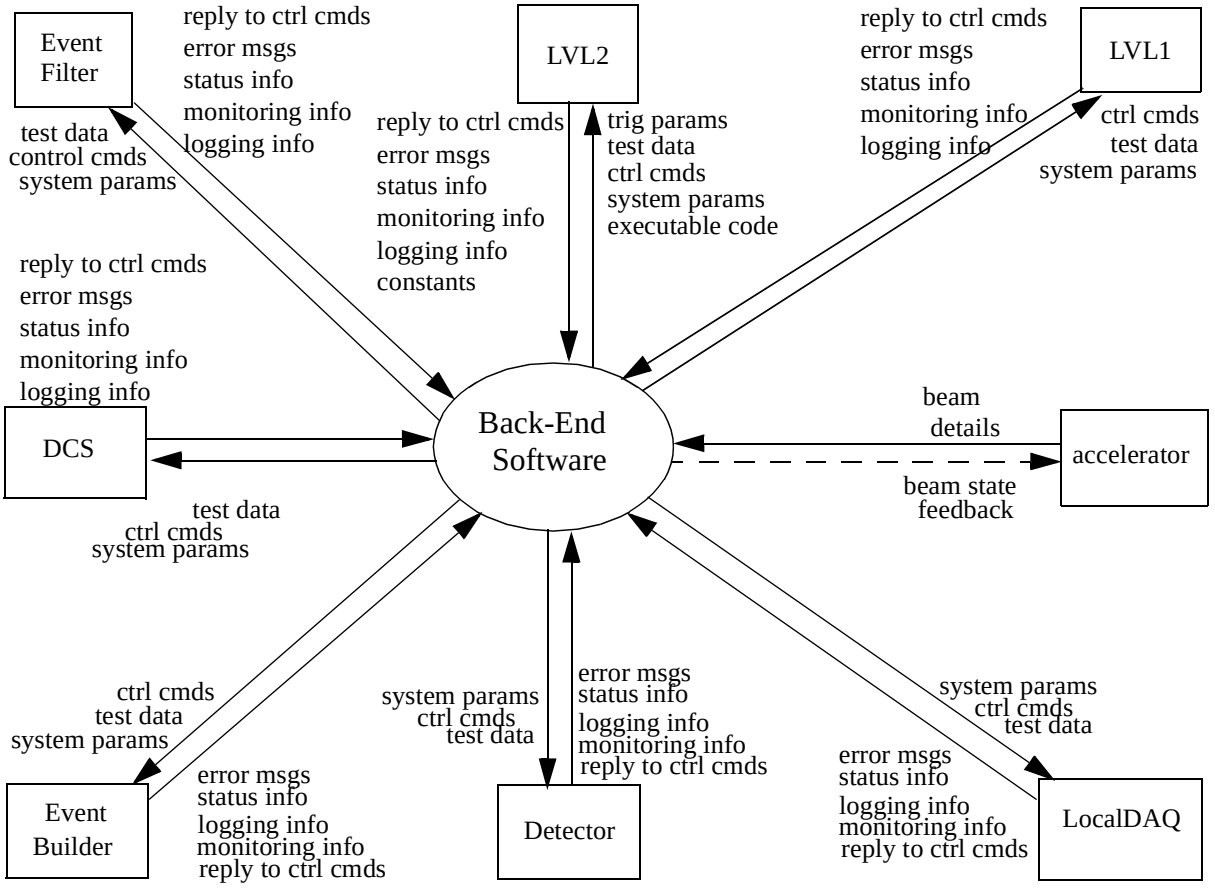
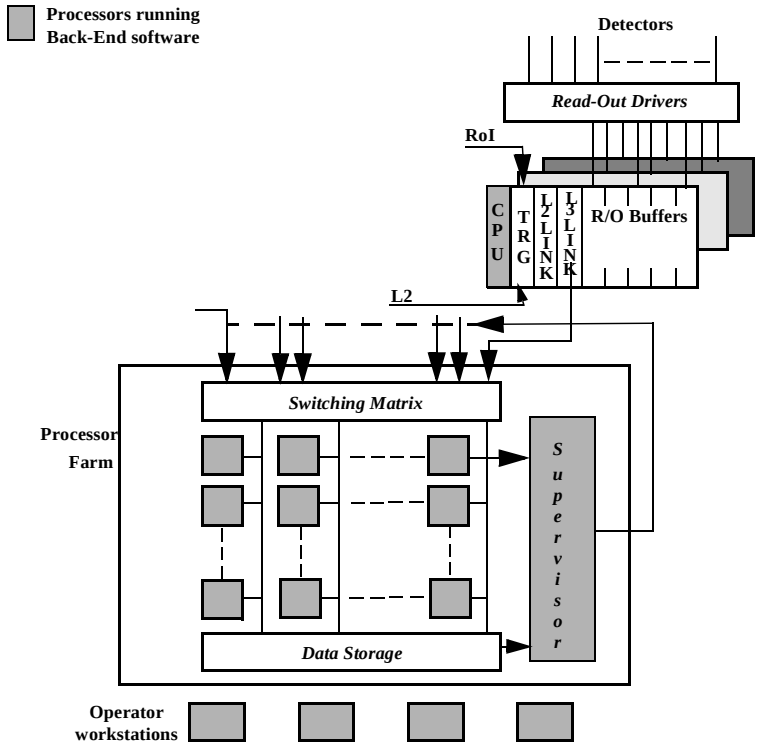


Figure 2 shows on which processors the back-end software is expected to run, that is the event filter processors, supervisor and the LDAQ processors in the detector read-out crates. Note that the back-end software is not the only software that will be run on such processors. A set of operator workstations, situated after the event filter processor farm, dedicated to providing the man-machine interface and hosting many of the control functions for the DAQ system will also run back-end software.

Figure 2 ATLAS DAQ/Event Filter Prototype "-1" architecture



The back-end software is essentially the "glue" that holds the various sub-systems together. It does not contain any elements that are detector specific as it is to be used by all possible configurations of the DAQ and detector instrumentation.

The back-end software is but one sub-system of the whole DAQ system and it must co-exist and co-operate with the other sub-systems. In particular, interfaces are required to the following sub-systems of the DAQ and external entities:

trigger
: receives trigger configuration details and can modify the configuration

processor farm
: back-end software synchronises its activities with the processor farm when handling physics data

accelerator
: beam details are retrieved from the accelerator and feedback is provided to the machine operators on the state of the beam at the detector site

event builder
: back-end software synchronises its activities with the event builder for the handling of physics data

Local DAQ
control and configuration information as well as synchronization

DCS
receives status information from detector control system and sends control commands

1.1 Operational environment

The environment in which the software is to run is partly dependent on the platforms chosen by the Data-Flow group and described in [3]. It is expected that this environment will be a heterogeneous collection of UNIX workstations, PCs running Windows NT and embedded systems (known as LDAQ processors) running various flavours of real-time UNIX operating systems (e.g. LynxOS) connected via a Local Area Network (LAN).

A great number of hardware components have to be used to provide the necessary computing power. These will be back-end and LDAQ processors loosely coupled by local networks and data flow channels. We assume that network connections (e.g. ethernet or replacements) running the most popular protocols (e.g. tcp/ip) are available to all the target platforms for exchanging control information and that its use will not interfere with the physics data transportation performance of the DAQ.

The ATLAS prototype DAQ system will need to be able to run using all or only a part of its sub-systems. It will be assembled in a step by step manner, according to financial and technical dependencies and constraints. A high degree of modularity is needed to connect, disconnect or add new components at will.

Many groups of people will interact at the various hardware and software levels, and so we have to foresee a significant level of sub-system unavailability and that this shall be detected and tolerated during DAQ system startup. Hence checking procedures shall be a concern to detect such configuration problems.

The failure of an individual component shall not affect the operation of other components. Every software component shall be designed with some form of self-test capability that can be used to verify a minimum of functionality.

1.2 Back-End DAQ Software Components

1.2.1 The software component model

The user requirements gathered for the back-end sub-system have been divided into groups related to activities providing similar functionality. The groups have been further developed into components of the back-end with a well defined purpose and boundaries. The components have interfaces with components and external systems, specific functionality and their own architecture.

From analysis of the components, it was shown that several domains recur across all the components including data storage, inter-object communication and graphical user interfaces.

1.2.2 Core components

The following 5 components are considered to be the core of the back-end subsystem, The core components constitute the essential functionality of the back-end subsystem and have been given priority in terms of time-scale for development.

1.2.2.1 Run control

The run control system controls the data taking activities by coordinating the operations of the DAQ sub-systems, back-end software components and external systems. It has user interfaces for the shift operators to control and supervise the data taking session and software interfaces with the DAQ sub-systems and other back-end software components. Through these interfaces the run control can exchange commands, status and information used to control the DAQ activities.

1.2.2.2 Configuration database

A data acquisition system needs a large number of parameters to describe its system architecture, hardware and software components, running modes and the system running status. One of the major design issues of Atlas DAQ is to be as flexible as possible, parameterized by the contents of databases.

1.2.2.3 Message reporting system

The aim of the Message Reporting System (MRS) is to provide a facility which allows all software components in the ATLAS DAQ system and related subsystems to report error messages to other components of the distributed DAQ system. The MRS performs the transport, filtering and routing of messages. It provides a facility for users to define unique error messages which will be used in the application programs.

1.2.2.4 Process manager

The purpose of the process manager is to perform basic job control of software components of the DAQ. It is capable of starting, stopping and monitoring the basic status (e.g. running or exited) of software components on the DAQ workstations and LDAQ processors independent of the underlying operating system. In this component the terms process and job are considered equivalent.

1.2.2.5 Information service

The Information Service (IS) provides an information exchange facility for software components of the DAQ. Information (defined by the supplier) from many sources can be categorised and made available to requesting applications asynchronously or on demand.

1.2.3 Trigger / DAQ and detector integration components

Given that the core components described above exist, the following components are required to integrate the back-end with other on-line subsystems and detectors.

1.2.3.1 Partition and resource manager

The DAQ contains many resources (both hardware and software) which cannot be shared and so their usage must be controlled to avoid conflicts. The purpose of the Partition Manager is to formalise the allocation of DAQ resources and allow groups to work in parallel without interference.

1.2.3.2 Status display

The status display presents the status of the current data taking run to the user in terms of its main run parameters, detector configuration, trigger rate, buffer occupancy and state of the subsystems.

1.2.3.3 Run bookkeeper

The purpose of the run bookkeeper is to archive information about the data recorded to permanent storage by the DAQ system. It records information on a per-run basis and provides a number of interfaces for retrieving and updating the information.

1.2.3.4 Event dump

The event dump is a monitoring program with a graphical user interface that samples events from the data-flow and presents them to the user in order to verify event integrity and structure.

1.2.3.5 Test manager

The purpose of the test manager is to organise individual tests for hardware and software components. The individual tests themselves are not the responsibility of the test manager which simply assures their execution and verifies their output. The individual tests are intended to verify the functionality of a given component. They will not be used to modify the state of a component or to retrieve status information. Tests are not optimized for speed or use of resources and are not a suitable basis for other components such as monitoring or status display.

1.2.3.6 Diagnostics package

The diagnostics package uses the tests held in the test manager to diagnose problems with the DAQ and verify its functioning status. By grouping tests into logical sequences, the diagnostic framework can examine any single component of the system (hardware or software) at different levels of detail in order to determine as accurately as possible the functional state of components or the entire system. The diagnostic framework reports the state of the system at a level of abstraction appropriate to any required intervention.

2 Software Technologies

2.1 Introduction

The various components described in this document all require a mixture of facilities for data storage, inter-process communication in a LAN network of processors, graphical user interfaces, complex logic-handling and general operating system services. To avoid unnecessary duplication, the same facilities are used across all components. Such facilities must be portable (the prototype DAQ will include processors running Solaris, HPUX and WNT). In particular they must be available on the LynxOS real-time UNIX operating system selected for use on the LDAQ processors. Candidate freeware and commercial software packages were evaluated to find the most suitable product for each technology.

2.2 General purpose programming toolkit

Rogue Wave Tools.h++[9] is a C++ foundation class library available on many operating systems (Unix, MS Windows, WNT, OS/2) that has become an industrial standard and is distributed by a wide variety of compiler vendors. It has proven to be a robust and portable library that can be used for DAQ programming since it supports many useful classes and can be used in a multi-threaded environment. We have acquired the sources for the library and ported it to LynxOS.

Since this decision, the C++ language has been accepted as an international standard and the Standard Template Library (STL) has become widely available. However, for the length of the current project we will continue to use Tools.h++ since migration would involve modifying source code.

2.3 Persistent data storage

Within the context of the ATLAS DAQ/Prototype “-1” project, the need for a persistent data manager to hold configuration information was identified. The ATLAS DAQ group has evaluated various commercial and shareware data persistence systems (relational databases, object databases and object managers) but no single system satisfied all the documented user requirements.

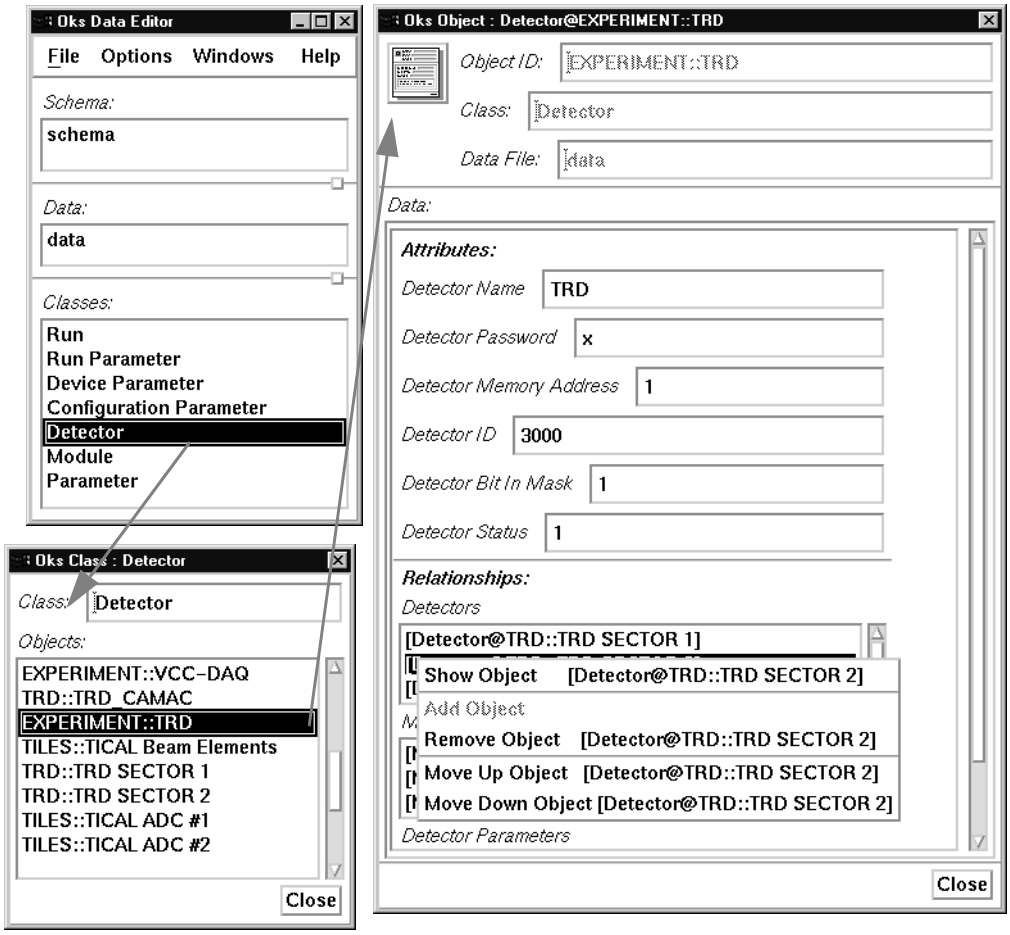
As a consequence, it was decided to adopt a two-tier architecture, using a light-weight in-memory persistent object manager to support the real-time requirements and a full ODBMS as a back-up and for long-term data management.

2.3.1 Light-weight in-memory object manager

For the object manager, a package called OKS has been developed on top of Rogue Wave’s Tools.h++ C++ class library. The OKS system is based on an object model that supports objects, classes, associations, methods, data abstraction, inheritance, polymorphism, object identifiers, composite objects, integrity constraints, schema evolution, data migration, active notification and queries. The

OKS system stores database schema and data in portable ASCII files and allows different schema and data files to be merged into a single database. It includes Motif based GUI applications to design database schema and to manipulate OKS objects (Figure 3) A translator has been developed between the OMT object model and OKS object model implemented with StP [10].

Figure 3 The OKS Data Editor



2.3.2 Objectivity/DB database system

Objectivity/DB is a commercial object oriented database management system introduced to CERN by the RD45 project [11]. We evaluated the basic DBMS features (schema evolution, access control, versioning, back-up/restore facilities etc.) and the C++ programming interface. A prototype translator has been developed between the OMT object model and Objectivity object model implemented with StP [10].

2.4 Inter-process communication

Message passing in our distributed environment is a topic of major importance since it is the communication backbone between the many processes running on the different machines of the DAQ system. Reliability and error recovery are very strong requisites as well as the ability to work in an event driven environment (such as X11). Many components of the back-end software require a transparent means of communicating between objects independent of their location (i.e. between objects inside the same process, in different processes or on different machines).

We chose to evaluate the Object Management Group's [12] Common Object Request Broker Architecture (CORBA) standard. ILU [13] is a freeware implementation of CORBA by Parc Xerox. The object interfaces provided by ILU hide implementation distinctions between different languages, between different address spaces, and between operating system types. ILU can be used to build multi-lingual object-oriented libraries with well-specified language-independent interfaces. It can also be used to implement distributed systems and to define and document interfaces between the modules of non-distributed programs. ILU interfaces can be specified either in the OMG's CORBA Interface Definition Language (OMG IDL) or ILU's Interface Specification Language (ISL). We have ported ILU to LynxOS.

2.5 Dynamic object behaviour

Many applications within the ATLAS DAQ prototype have complicated dynamic behaviour which can be successfully modelled in terms of states and transitions between them. Previously, state diagrams, implemented as finite state machines, have been used which, although effective, become ungainly as system size increases. Harel statecharts address this problem by implementing additional features such as hierarchy and concurrency.

CHSM [14] is an object-oriented language system which implements Harel statecharts as Concurrent, Hierarchical, finite State Machines supporting many statechart concepts as illustrated in the abstract example shown in Figure 4 (a):

Hierarchy
states f and e are child-states of parent state q.

Clusters (logical-exclusive-or state groups)
states a, b and c are in cluster x. To be in x is to be in a, b or c. The transition to d is taken regardless of x's child-state.

History
when cluster x is re-entered the previous child-state is entered.

Sets (logical-and state groups)
cluster p and cluster q are child states of set s. To be in set s is to be in both child-states p and q.

Concurrency
if event α occurs while the statechart is in child-states a and f then it will simultaneously make transitions to states e and c.

Guard conditions
the transition from b to c only occurs if $v < 4$.

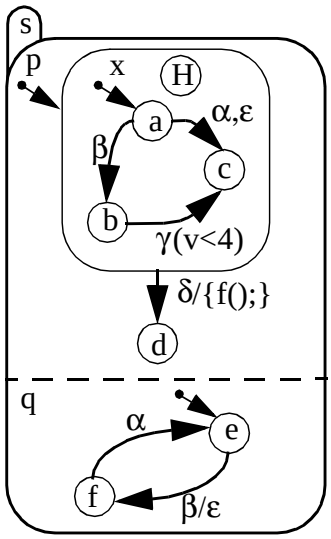
Broadcasting
event ε is broadcast when the transition from f to e is made.

Actions

function f() is executed when the transition from x to d is made, in addition actions can be executed on entering or exiting a state (not shown).

CHSMs are described by means of a CHSM description text file and Figure 4 (b) shows the CHSM description corresponding to the statechart shown in Figure 4 (a). The CHSM compiler converts the description to C++, which may be integrated with user defined code.

Figure 4 (a) StateChart (b) CHSM description



(a)

```
set s(p,q) is {
   cluster p(x,d) is {
      cluster x(a,b,c) history {
            upon enter %{enterx();%}
            upon exit  %{exitx();%}
            delta->d %{ f(); %};
     } is {
            state a {
                  alpha,epsilon->c;
                   beta->b;
            }
           state b { gamma [v<4]->c; }
            state c;
      }
      state d;
   }
   cluster q(e,f) is {
      state e { beta->f %{epsilon();%};}
      state f { alpha->e;}
   }
}
```

(b)

We have evaluated the CHSM language system and have shown it to be suitable for describing the dynamic behaviour of typical DAQ applications. For example, the prototype DAQ run control has been implemented using CHSM.

2.6 Graphical user interfaces

Modern data acquisition systems are large and complex distributed systems that require sophisticated user interfaces to monitor and control them. X11 and Motif are the dominant technologies on UNIX workstations but the advent of WNT has forced us to reconsider this choice.

2.6.1 Java

Java is a simple object-oriented, platform-independent, multi-threaded, general-purpose programming environment. The aim of our evaluation was to understand if Java could be used to implement the status display component of the DAQ. A demonstration application was developed to investigate such topics as, creating standard GUI components, client-server structure, use of native methods, specific widgets for status displays and remote objects. The performance was compared to X11 based alternatives. The demo contains three essential parts: servers, simulators and applets. Servers realise the binding with remote objects, simulators create simulation data and update remote objects (to mimic the DAQ), applets put simulation data to remote objects or get them from remote objects and display them. The appearance of the Java implementation of the status display is shown in Figure 18

The entire application is written in Java (JDK1.0), communication is realised by Java IDL (alpha2). The servers and simulators run on a SUN (Solaris2.5) and the applets can be loaded from any machine using a Java compatible browser. Work is continuing on the integration of the demo status display with the ILU CORBA system described above.

2.6.2 MVC and X-Designer

X-Designer [15] is an interactive tool for building graphical user interfaces (GUIs) using widgets of the standard OSF/Motif toolkit as building blocks that has been used extensively in the RD13 project [16]. It is capable of generating C, C++ or Java code to implement the GUI. We investigated the construction of GUIs with a Model-View-Controller (MVC) architecture using the C++ code generation capabilities of the tool.

A number of widget hierarchies were created which correspond to commonly used patterns in GUIs (e.g. data entry field with a label). These also correspond to views or controller-view pairs in the MVC architecture. By making each hierarchy a C++ class it could be added to the X-Designer palette and used in subsequent designs. A GUI for the existing RD13 Run Control Parameter database was successfully built using these definitions.

X-Designer also provides a facility called XD/Replay which can be used to record or playback any Xt based application. In record mode it writes a high-level description of actions performed on the application in a script. On replaying the script, the recorded actions are executed on the application. This facility will be useful for automating the testing of any DAQ application with a Motif GUI and for producing automated demonstrations of applications.

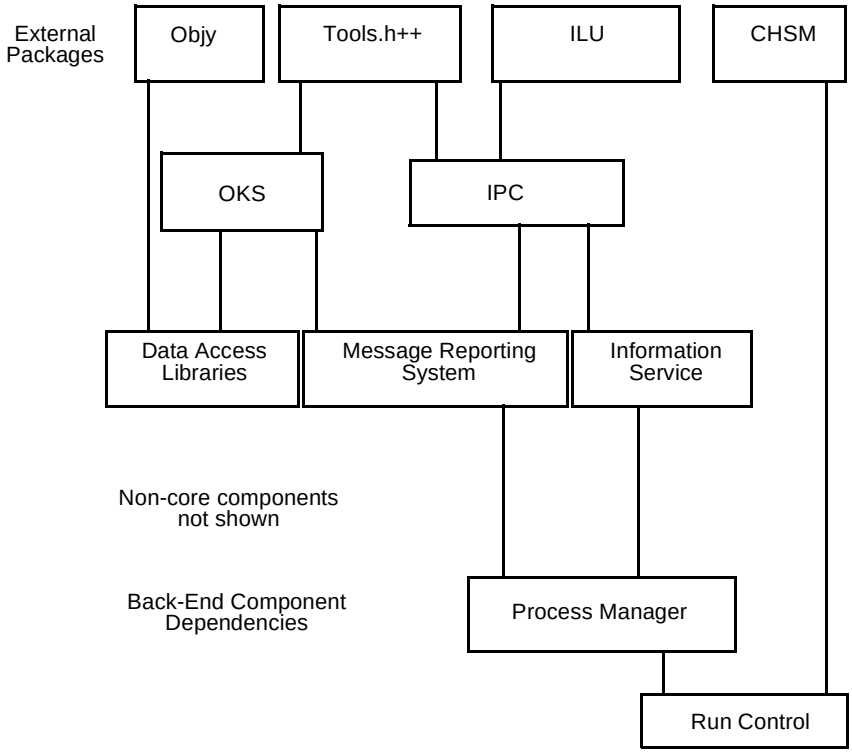
Work will continue to investigate if the MVC approach and Java code generation facilities of X-Designer can be used to develop the DAQ graphical user interfaces. X-Designer is currently being used to develop a GUI for the run control component.

3 Back-end DAQ software components

3.1 Introduction

The back-end components described use a common base for key software technologies. They also have inter-dependencies between them as represented (in a simplified manner) below (Figure 5).

Figure 5 Dependencies between back-end core components and external packages



This chapter examines each of the back-end components defined in the previous chapter in greater detail.

3.2 Run control

The run control system controls the data taking activities by coordinating the operating of the DAQ sub-systems, back-end software components and external systems. It has user interfaces for the shift operators to control and supervise the data taking session and software interfaces with the DAQ sub-systems and other back-end software components. Through these interfaces the run control can exchange commands, status and information used to control the DAQ activities.

Through the user interface it receives commands and information describing how the user wants the DAQ system to take data. It allows DAQ users to select a DAQ system configuration, parameterize it for a run and start and stop the data taking sub-systems.

The Run Control component operates in a environment consisting of multiple partitions that may take data simultaneously and independently. Each copy of the run control is capable of controlling one partition marshalled by the Partition Manager (section 3.7.1 on page 35).

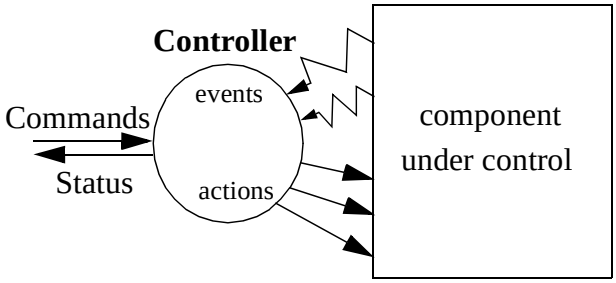
In general the run control needs to send commands to the other DAQ sub-systems in order to control their operation and receive change of state information. The external sub-systems are autonomous and independent of the run control so their detailed internal states remain hidden. If a sub-system changes state the run controller reacts appropriately, for example stop the run if a detector is no longer able to produce data. The run control will interact with a dedicated controller for each sub-system.

3.2.1 Run Control architecture

The architecture of the run control can be seen as a hierarchy of control entities called controllers, each with responsibility for a well defined component or part of the DAQ. The controller's state is the simplified external view of the current working condition of the component or part of the DAQ under its responsibility.

Each controller can receive commands from the outside world. Commands cause a controller to execute actions which potentially change the visible state of the component. A controller can also react to local events occurring in the DAQ component under its responsibility (see Figure 6). Typically its reaction will be to execute some actions and potentially change its visible state.

Figure 6 Interactions between a controller and the component under control

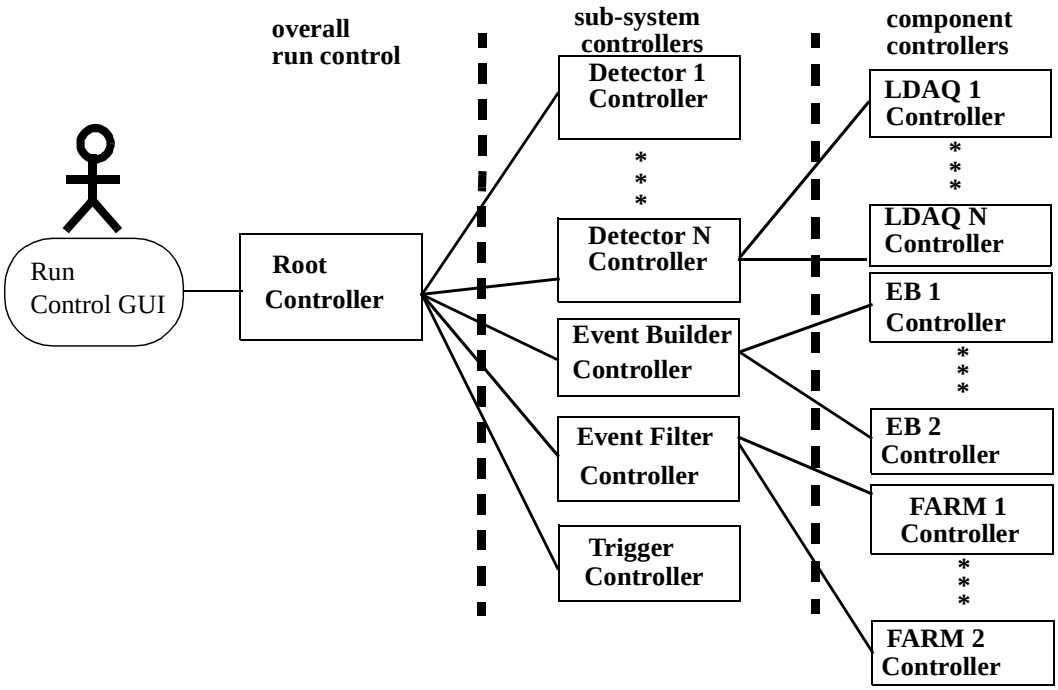


The controllers are organised into a hierarchical tree structure that reflects the general organisation of the DAQ system itself. Each controller in the tree can have one parent (or superior) controller and any number of child (subordinate) controllers. At the top of the tree is a single controller which represents the overall state of the entire on-line system. Below the overall, or general, controller are a set of controllers, one for each major sub-system of the DAQ and the physics detectors. Below each sub-system controller there may be further component controllers which are responsible for individual components.

The hierarchical tree of controllers transmit messages between themselves to exchange commands and status information. In general, commands starting from the human operator, are sent to the general controller which forwards them to the sub-system controllers who in turn forward them to component controllers and so on. In this respect commands flow from the root of the tree towards the leaves. The result of commands are sent back through the tree so that the human operator is made aware of any change in the state of the system. Any node in the control tree can perform actions on the commands it receives.

We assume, from observation of the structure of the on-line system, that three levels of controllers will be sufficient but the design of the run-control component does not set any limitation on the depth or width of the control tree. The following figure (Figure 7) shows, as an example, a typical configuration of the run-control component.

Figure 7 Run control architecture: example configuration



3.2.2 Generic controller state chart

The dynamic behaviour for each controller in the Run Control tree is modelled by the same generic state chart (see Figure 8). Developers of the various controllers in different parts of the tree customise the behaviour of their particular controller by adding code to implement the required behaviour within this generalised framework.

The complete state chart consists of a number of nested and concurrent states. Each state is labelled with the name of the state. Transitions between states are indicated by arrows labelled with the name of the event which will cause the transition to occur. A guard condition is evaluated before the transition occurs. If it evaluates to true the transition will take place, if it evaluates to false the transition will not occur. The action is a function or other piece of code to be executed if the transition takes place successfully. Actions can also be performed on state entrances and exits.

3.2.3 Error Recovery

The Run Control generic controller has been designed to cope with three different levels of error which may occur whilst the system is running:

- If an error occurs when making a transition between two states, a mechanism is foreseen to take the controller back to the last error-free state undoing any actions which were made during the transition which caused the error.
- If the error is more serious and cannot be cleared by the above mechanism, the whole controller can be reset, during which, all allocated resources should be reset and freed and the controller is put back to its initial state.
- If a fatal, non-recoverable error occurs somewhere in the overall run control system, the whole system needs to be shutdown as cleanly as possible. Since system integrity cannot be guaranteed in such a situation each individual controller cannot rely on any communication or external interaction during the shutdown.

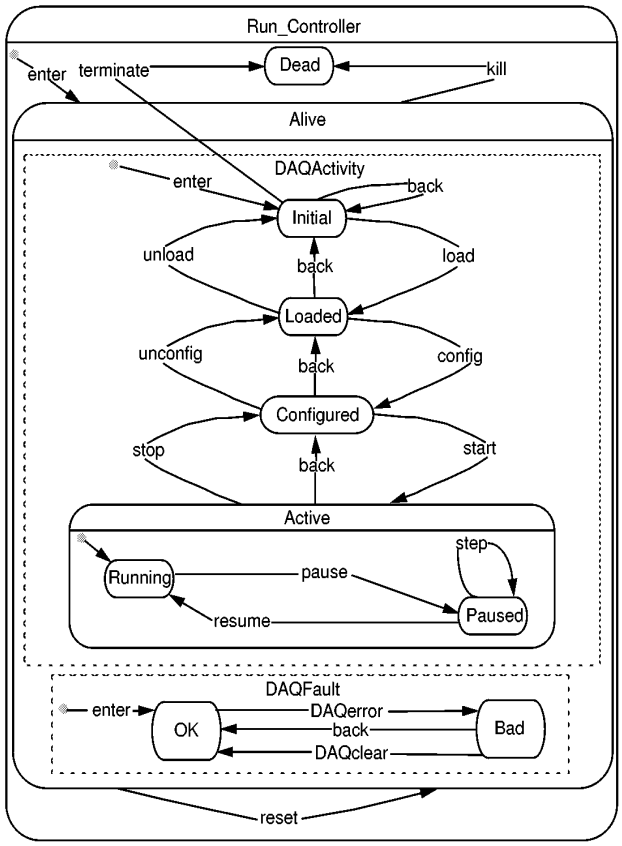
Referring to Figure 8, consider the central, *Alive* superstate. This state is a superstate composed of two concurrent states: *DAQActivity* and *DAQFault* - this means that when the *Alive* state is active then both *Activity* and *Fault* are active too. The reason for the concurrent states is to allow error recovery when a problem arises on a transition. Both *DAQActivity* and *DAQFault* are superstates. The current child state in *DAQActivity* reflects the status of the controller. The current child state in *DAQFault* indicates whether or not a fault occurred when the controller performed its last transition.

For instance, the root controller could instruct a local controller to go from state *Initial* to state *Loaded* by issuing the *load* event. The local controller will first check that it is not already in a faulty state by evaluating the guard condition. Assuming that the guard condition evaluates true, the transition takes place, the *Loaded* state is entered and the associated transition action takes place to perform the necessary actions. However, if something goes wrong during this process the local controller will issue the *error* event internally which causes the *DAQFault* concurrent state to make the transition to *Bad*. The only way the fault can be cleared is to issue the *back* event which will cause the *DAQActivity* and *DAQFault* concurrent superstates to make transitions simultaneously to states *Initial* and OK respectively. In the *DAQActivity* cluster a user-supplied action should be specified for the back transition which tries to “undo” any actions which were carried out during the failed load action to ensure that the controller is returned correctly to the Initial state.

More serious errors are dealt with at the next higher level in the diagram. If the above mechanism fails to clear an error then a reset event can be issued. This causes the *Alive* state to exit, independently of which states are currently active in the *DAQActivity* and *DAQFault* clusters. An exit function is called for the *Alive* set followed by a transition action and an enter action when the *Alive* set is re-entered. All these actions can be user-defined and should carry out whatever actions are necessary to put the controller back to its *Initial* state. When the *Alive* set is re-entered it will enter the default states of the *DAQActivity* and *DAQFault* clusters i.e. *Initial* and *OK*.

Finally, when a fatal error occurs in the Run Control system the *kill* event can be issued which will cause the controller to change state from *Alive* to *Dead*. On exiting the *Alive* set the corresponding *exit* function will be called. Furthermore, a transition action will be called when the transition to *Dead* is complete. The user-defined action should perform all the necessary actions required to shut down the system as cleanly as possible, but without communicating with any other modules as system integrity cannot be guaranteed in such a situation.

Figure 8 Generic controller state chart



The generic controller states and the behaviour of the associated component under control expected in each state is described below:

Initial

The controller is visible to and can communicate with the Run Control.

Loaded

The component under control has read any configuration data needed for initialisation.

Configured

The component under control has executed all its initialisation phases and is ready to operate.

Running

The component under control is running and fulfilling its functions for data taking in the selected DAQ configuration.

Paused

The component under control is in “halt” mode, (i.e. frozen), after it has been *Running*. It is able to switch back into the *Running* state and recover its execution context as if it had never been paused.

Bad

The component under control failed to execute its last command or an error has occurred. It is likely to be in an undefined state.

OK

The component under control reverted to its previous valid state (after being in the *Error* state)

Dead

The process or thread executing the controller terminates.

All transitions are defined for all controllers. If a particular controller has no need to perform any actions during a transition then it should immediately complete successfully. Such dummy actions are provided with the controller skeleton and are the default behaviour. Functions are associated to transition actions by the Controller developer. Such functions are executed automatically when the transition is fired and should perform all the actions that are necessary to change the module’s internal state to be in accordance with the required semantic for the new state. The developer may optionally specify entry and exit actions for each state. Entry and exit actions are called no matter which transition caused the state change.

load

component under control initialises any software packages it will require for operating (e.g. access to database, networks etc.)

unload

component under control disengages itself from any software packages it used during operation (e.g. database, networks etc.)

config

component under control reads required parameters (e.g. from database or command line), starts any other services required (e.g. creates threads or processes), connects to other modules (e.g. establishes network connections) and requests any resources required.

unconfig

component under control deallocates any resources acquired, stops any threads or processes created and disconnects from other modules.

start

component under control reads any dynamic parameters and starts operation (e.g. taking data).

stop

component under control stops operating and becomes ready but idle.

pause

component under control stops operating temporarily and becomes ready but idle.

resume

component under control resumes operation.

step
component under control resumes operation for only one event or unit of work and then pauses again.

DAQerror
an error occurred during a component under control's activity that it could not correct itself and which inhibits it from continuing.

DAQclear
clears existing error without going back to previous state or performing any actions. This is equivalent of saying the error is not important.

back
retracts the actions performed to go back to the previous error-free state.

reset
all the component under control's internal resources are reset or freed and the module goes back to the *Initial* state.

kill
the component under control tries to stop as cleanly as possible any on-going activity associated to its current state and then exits (i.e. termination of process). The component under control should not rely on any communication or external interaction in this transition since the global Run Control system is potentially corrupted.

terminate
complete shutdown of component under control under normal circumstances.

3.2.4 The DAQ supervisor

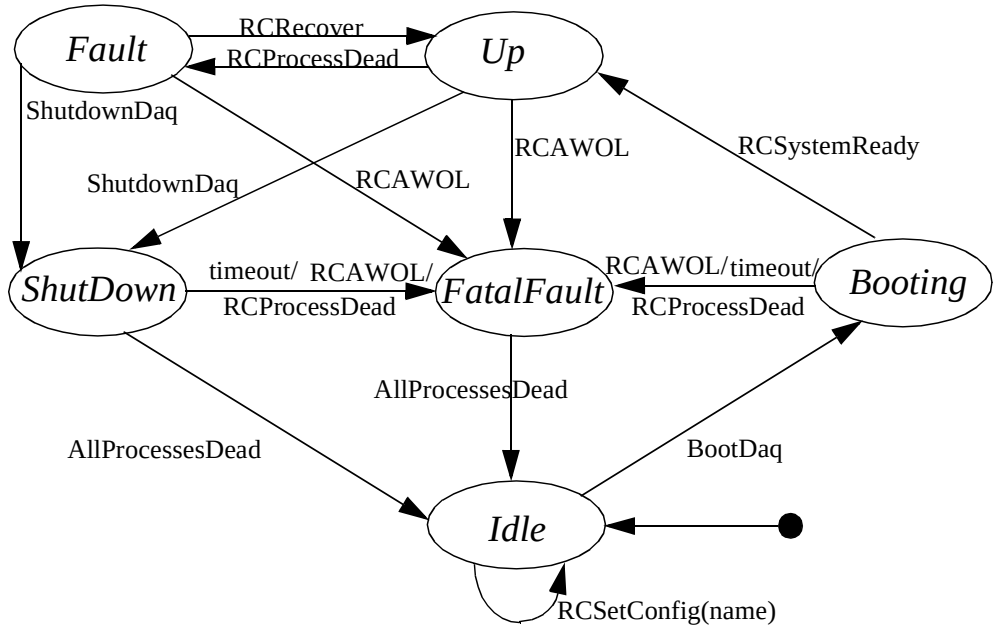
An important task of the DAQ back-end software is to marshal the DAQ through its startup and shutdown procedures so that they are performed in an orderly manner. Such procedures should be as complete and automatic as possible. The procedures must take the minimum amount of time to execute since this will affect the total amount of data that can be taken with the DAQ during a shift period.

The startup procedure marshals DAQ resources from an initial state (i.e. powered-up but not configured) to an operational level at which they can be used to take data. The shutdown procedure marshals DAQ resources from a data taking state back to their initial state, cleaning-up and deallocating any assigned resources.

The DAQ supervisor encompasses the functionality required to perform the startup and shutdown procedures. It is responsible for the creation of all software processes (including all the controllers of the run-control component) during start-up of the DAQ according to the configuration defined in the database. It uses the Process Manager (section 3.5 on page 31) to create processes on the processors of

the DAQ system. It is also responsible for ensuring the orderly shut down of the DAQ system at the end of data taking activities. A separate state chart (as shown in Figure 9) is used to model the behaviour of the DAQ supervisor as it marshals the DAQ during these setup and shut-down phases.The significant states are described below.

Figure 9 The DAQ supervisor state chart



Idle

The Run Control is not usable. It is waiting for a configuration name for data taking.

Booting

The various objects of the Run Control are being created, initialising and interconnecting according to the chosen configuration.

Up

The Run Control is available. Run Control commands can be sent to the various components. The root controller's state can be seen as a sub-state associated to this *Up* state.

FatalFault

One of the critical Run Control objects has collapsed and the Run Control cannot work. It is doing an emergency shutdown.

ShutDown

The Run Control is stopping all of its configuration dependant objects and is cleaning everything to be able to restart with a new configuration.

Fault

One or several of the run control non critical processes have collapsed. The system is waiting for the user to decide if it should disconnect them from the configuration and recover or to shutdown the system as cleanly as possible.

The semantics and details of the DAQ supervisor state chart represent a minimal supervisor. In a later version of the Run Control component, a more sophisticated supervisor capable of taking decisions based on the information available from many sources (i.e. controller status, information service, message reporting system, configuration database, resource manager etc.) to rectify faults and perform error recovery is foreseen.

3.2.5 Elements of the run-control component

Based on the above design, the following elements of the run control component are provided:

- An implementation of a controller skeleton that can be used in any of the example situations described above. This skeleton operates according to a generalised statechart (section 3.2.2) and is capable of receiving and sending commands via network messages. **The actions performed by the controllers will need to be implemented by the developers working on the subsystems,**
- The means of organising a set of controllers into a hierarchy representing the logical structure of the experiment. The hierarchy is defined via the configuration database,
- A graphical user interface for monitoring and interacting with the set of controllers that form the control hierarchy,
- A simplified graphical user interface that is only capable of communication with a single controller. The intended purpose of this GUI is to allow a controller to be tested and debugged in isolation,
- A software interface so that ancillary programs (such as a graphical status display) can retrieve information about the status of each controller,
- A supervisor program capable of starting the various software elements of the DAQ according to a predefined configuration stored in the database.

3.3 Configuration database

The DAQ database management system is the repository and the server for all data describing the DAQ system configurations. Configuration information is divided into several parts: hardware components, software architecture, run and sub-detector parameters. The information can be grouped according to the frequency at which it changes:

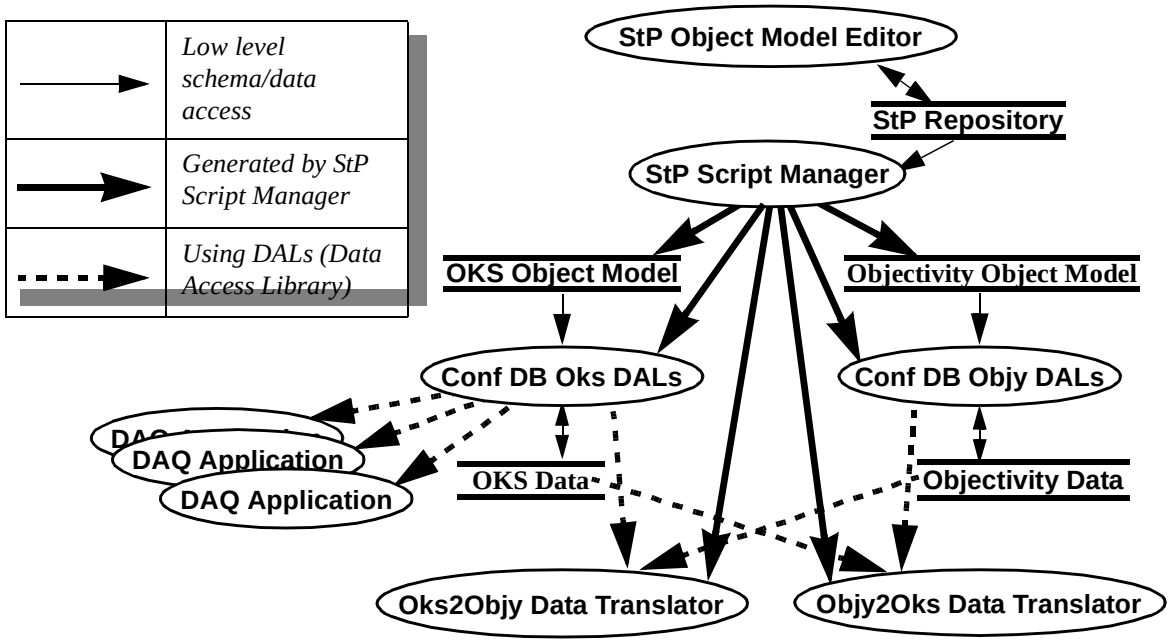
- static information that does not change during a data taking session (e.g. hardware components and software architecture),
- periodically updated information that does not change during a DAQ run (e.g. run and sub-detector parameters).

Several other databases are required by individual components or sub-systems (e.g. Message Reporting System database containing message definitions, routing and filtering information or Calibration database for detector calibration purposes etc.) but their requirements are defined elsewhere.

In order to use the database in a stand-alone system, it is useful to have the Configuration databases organised along subsystem boundaries. For the prototype -1 each of these subsystem will have all that is required in terms of hardware, software, run-control etc. configuration.

To hide from the DAQ configuration database user the details of the low level persistent data manager (i.e. OKS and Objectivity/DB) it is proposed to use Data Access Libraries (DAL). As well it is necessary to create data translator between run-time and back-up databases.

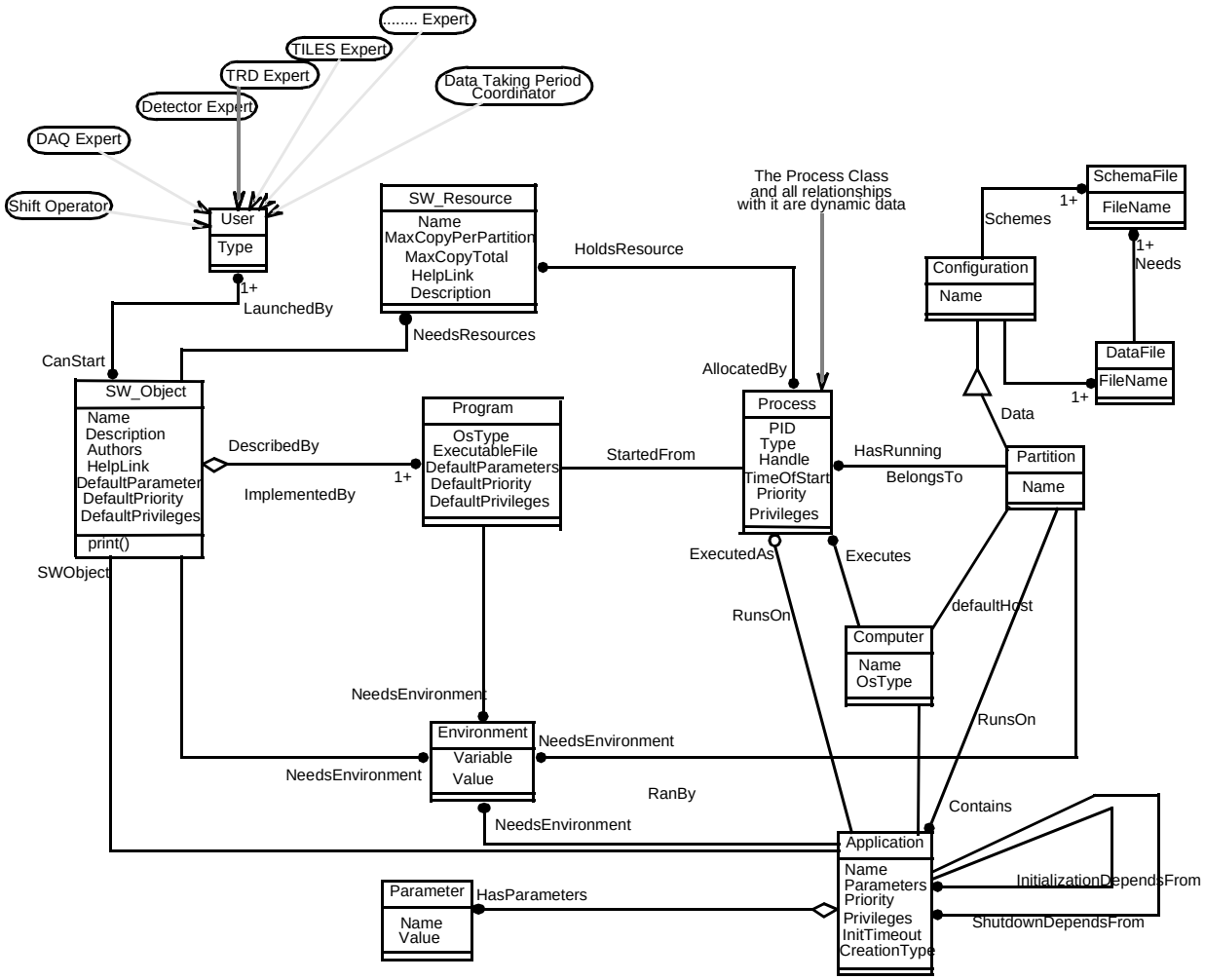
Figure 10 Planned Development Process for ATLAS DAQ Configuration Database



3.3.1 Configurations, Partitions and Authorization Control

Because the run-time persistent data manager reads data from several files and keeps all data in-memory, it seems reasonable to organise the run-time database repository as a set of files where a file contains data about a subsystem or part of a subsystem (Objectivity/DB has similar structure and stores data in *Federated database* which consists of *databases* and a *database* can be a file). This improves performance of applications and reduces memory consumption (for example, to receive a value of some detector's parameter from the Detector parameters database it is sufficient to load only the description of the actual detector [one data file] instead of the whole database). It also allows authorization control on the level of subsystems or any part of a subsystem (for example, if one group of users is responsible for changing some piece of database information, any other users will have read-only access).

Figure 11 Setup and Software configuration database

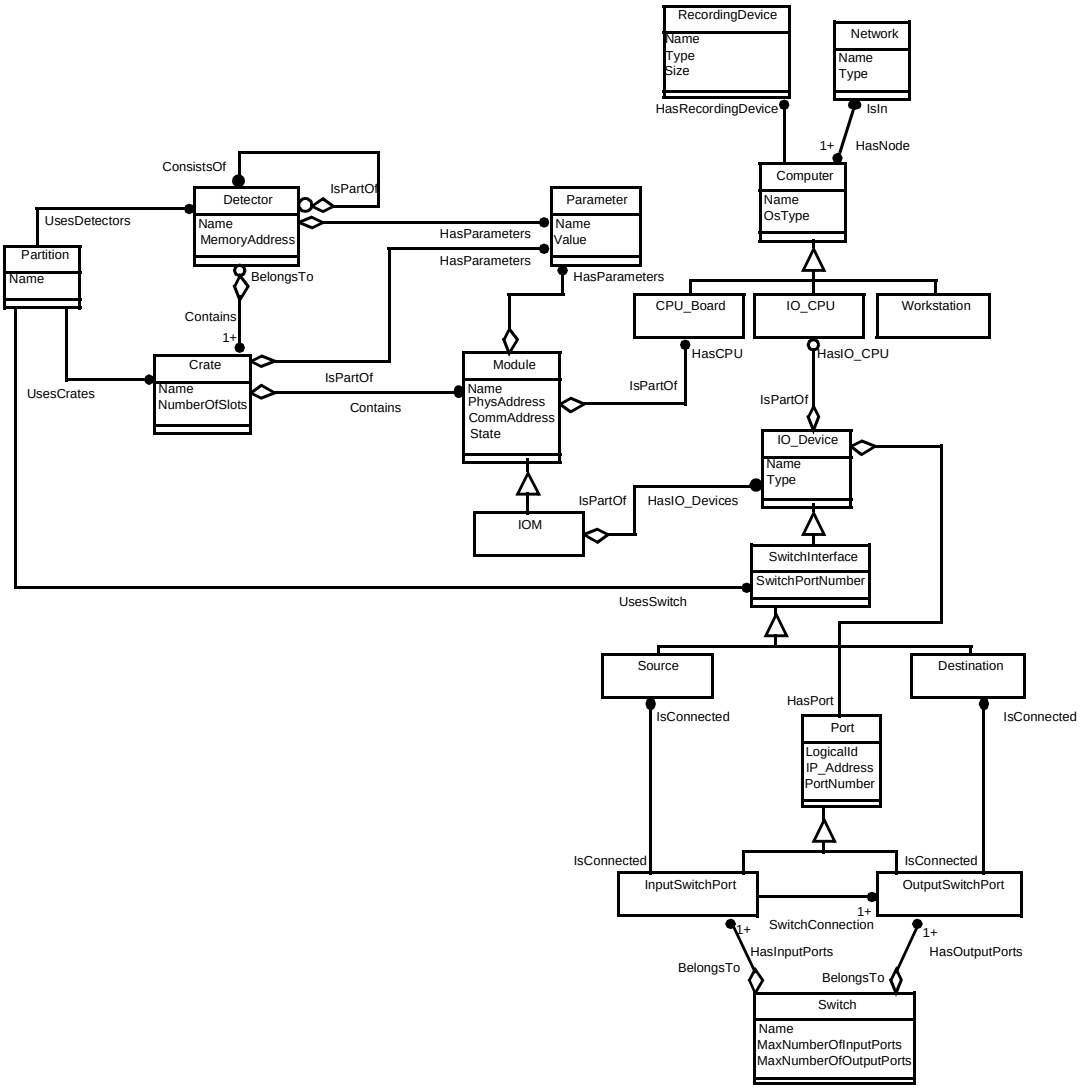


The instances of *SW_Object* and *Program* classes describe the state of DAQ software which is installed. An instance of *SW_Object* class must be created if a new software component is installed in DAQ environment. An instance of *Program* must be created if a software component was ported for a new operating system. An instance of *SW_Resource* must be created if a new limited resource was identified. The situation is similar with instances of *User* and *Environment* classes. The life-time and modification time of these instances differ from instances of *Application* class. Their instances and relationships between them can be different from one DAQ run to another. Hence the instances of above classes (SW_Object, Program, SW_Resource, User and Environment) should be stored in one database (which is seldom updated) and instances of *Application* class (probably together with instance of *Partition* class) should be stored in another database (which is updated more frequently). The access rights for these databases may be different according to usage.

3.3.2 Hardware Configuration Database Skeleton

The hardware database (Figure 12) describes all the hardware components (crates, modules, computers, devices and their interconnections) that are available to the DAQ system. A hardware component can be described even if it is not used by the DAQ software in view of a future use or in order to have a more complete idea of the hardware available.

Figure 12 Hardware Configuration Database Schema

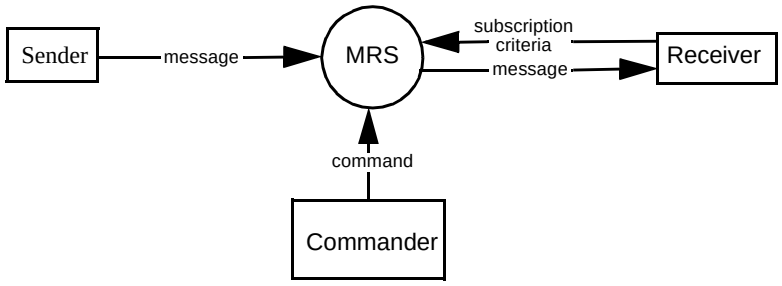


3.4 Message reporting system

The aim of the Message Reporting System (MRS) is to provide a facility which allows all software components in the ATLAS DAQ system and related processes to report error messages to other components of the distributed DAQ system. The MRS is responsible for the transport, filtering and routing of the messages. It includes a facility for users to define unique error messages to be used in the application programs.

MRS is not intended to be a general purpose communication tool but more an error and information message reporting system which can be used by any DAQ component.

Figure 13 Message Reporting System architecture



The MRS architecture (Figure 13) is composed of a MRS sender application interface, the MRS server, the MRS receiver interface, the MRS command interface and the MRS message definition database. In the planned DAQ system a high number of users requiring access to MRS is expected. Therefore a hierarchical internal architecture is envisaged, which is not visible to the user applications. The server functions of MRS are split into a **public MRS server** and a number of **private MRS servers**. The number of installed private servers can be adjusted according to the size and complexity of the DAQ system and the required message throughput. Each private server can receive messages from a given group of senders, which may be logically or geographically defined and could vary according to the observed traffic. Having been started up the private MRS servers *join* the public MRS server sending a *handle* (self handle) which can be used later to access the methods of the private MRS servers.

A user application wishing to **report messages** has to *connect* to the public MRS via the **MRS sender application interface** (MRS sender API). The public MRS decides for a particular private MRS to be in charge of by this application. It then sends a handle to the MRS sender API, which is used for the following operations when *reporting* the messages. Therefore the message will automatically be sent directly to the private MRS server where the final message is composed by adding the corresponding **MRS database** information. For the case where the MRS server is not present in the system, the MRS sender writes the message to a local log file.

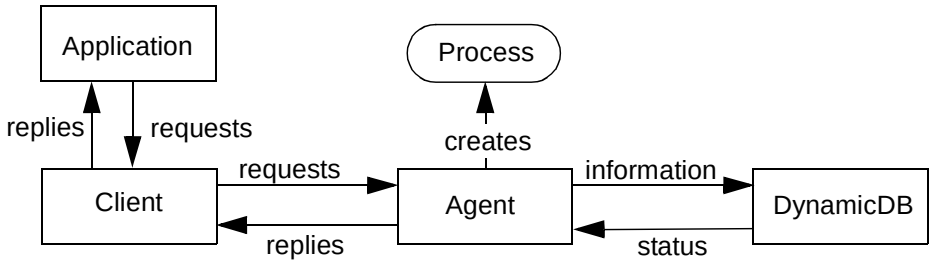
An application wishing to **receive messages** must *subscribe* to the public MRS server with the help of the MRS receiver interface. Subscription includes the request to receive messages and the subscription criteria. The public MRS server then transmits this subscription to all private MRS servers. According to their subscription list, all private MRS servers *report* any corresponding message directly to the **MRS receiver application interface** (MRS receiver API). When an application wants to *unsubscribe* from a particular message group it sends this request in the same way to the public MRS server, which transmits it to all the private MRS servers.

The DAQ expert can initiate MRS functions via the **MRS command application interface** (MRS command API). A number of commands can be send. Messages can be filtered (*set_filter*) on their way through the private MRS servers. The filter criteria are composed in the same way as the subscription criteria. If a message fulfils the filter criteria then it will not be reported to a receiver, even if it fulfils the subscription criteria. The MRS command API sends the filter command to the public MRS server. From there it is passed on to the private MRS servers, where the filter operation is performed. The current filter value can be retrieved with the *get_filter* command. MRS can be requested to update (*command*) the MRS internal information from the MRS definition database via the MRS command API. Other commands may be defined when they become necessary.

3.5 Process Manager

The purpose of the process manager is to perform basic job control of software components of the DAQ. It shall be capable of starting, stopping and monitoring the basic status (e.g. running or exited) of software components on the DAQ workstations and LDAQ processors. In this description the terms process and job are considered equivalent. The process manager consists of three elements: the client object, agents and the dynamic database (DynamicDB).

Figure 14 Process Manager architecture



3.5.1 The Client

The client is the only object visible by user programs. It is the interface between user programs and the process manager. It implemented as an object instance (linked into the application) via which a client requests Process Manager services.

A client may start a process either by passing all the necessary information (executable file, host machine etc.) or by passing instructions for the Process Manager to retrieve the information from the configuration database (i.e. SW_Object name and configuration). Processes may be started synchronously (i.e. the client is blocked until the creation of the process is confirmed or fails) or asynchronously (i.e. the client is continues processing and is informed when the process creation is confirmed or fails).

In these two possible starting operations the client may be passed the reference of a callback routine. This callback routine is a user provided function to be called automatically by the client under the following conditions:

- A process that this client requested to start has died.
- A process whose creation was requested in asynchronous mode is now created.
- A process whose creation was requested in asynchronous mode can’t be created.

3.5.2 The Agent

The agent is the object running on a machine which is in charge of creating, deleting and detecting death of the processes that are under the control of the Process Manager on that host machine. It uses the local machine operating system (OS) to manage the local processes. There must be a agent on a machine in order to manage processes on that machine. The agent is intended to be started (like a daemon) at boot time on each machine on which processes should be managed. The agent is also used as an interface between the local Clients and the DynamicDB thus only the agents need to know where the DynamicDB is running (i.e. on which machine) and how it is implemented.

3.5.3 The DynamicDB

The DynamicDB contains dynamic data about the Process Manager state:

- List of machines concerned by the Process Manager service
- List of running agents
- List of processes created by the Process Manager with their description data:
    - handle
    - user defined name
    - machine
    - local pid
    - executable binary program file

The user interface program (and any other applications) may read the above data from the DynamicDB as required.

3.6 Information Service

The Information Service (IS) is a a means of making available information which would normally be provided by a component or sub-system (i.e. sources) to other components and ancillary applications (i.e. receivers). IS thus provides a gateway between receivers and the internal functioning of the source components with the benefit of not obliging the sources to service the application requests for information themselves. It is intended that this will avoid users degrading the performance of data-taking activities and will also simplify their implementation by concentrating all the information they need in one service. The information made available (i.e. published) by sources can be updated and receivers can retrieve the latest copy.

From the information transfer point of view the modules in a data acquisition system can be considered as **sources** of information or **receivers** of information.

Information delivery can be done in several modes of which the following seems to be most useful:

- at regular time intervals;
- when the information changes;
- on request of a receiver.

The sources decide (depending on information content) how to deliver the information (at regular time intervals or when the information changes).

3.6.1 Relationship between the Message Reporting System and Information Service

From the description above, there may appear to be an overlap with the facilities proposed by the Message Reporting System (MRS). While the underlying functionality is similar (both use a common implementation), the style and usage of the information distributed by the two components is different. In short, MRS is used to send *notifications* (e.g. Run Stopped) while IS provides distributed access to *data values* (e.g. LDAQ crate 1 buffer occupancy = 30%). Another difference between MRS and IS concerns synchronization between sources and receivers of information. In MRS, a receiver has to request an information item *before* it is produced by a source and will not have access to it *after* it has been notified. In IS, the synchronization between sources and receivers of information is more lax since the service keeps a copy of the last value of information items which have been produced that can be accessed at any time.

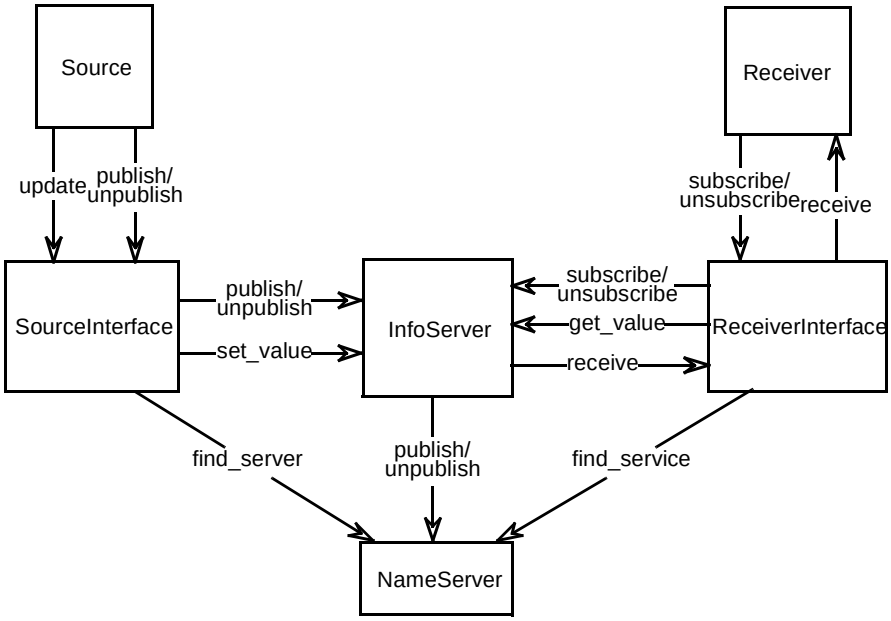
Experience and usage will determine if the two components remain independent or will be merged in later versions.

3.6.2 Information Service Architecture

The main elements of the IS are the application interfaces (for source and receiver) and the information server. IS transfers information from sources to receivers. Sources publish and send information and receivers subscribe for information and receive it. The aim of the application interfaces is to hide all communication aspects to sources and to receivers. It is a multiple server configuration in which servers are dedicated to different domains of the data acquisition system. The event flow diagram (including a source and a receiver in order to present the complete event flow) is shown in Figure 15. In order to be accessible, the information is registered in a name server.

The SourceInterface allows sources to **publish** or **unpublish** information and then **update** it. The **publish** method in the InfoServer creates an object which will contain the information and registers it with the NameServer. The **update** invokes the **set_value** method of this object.

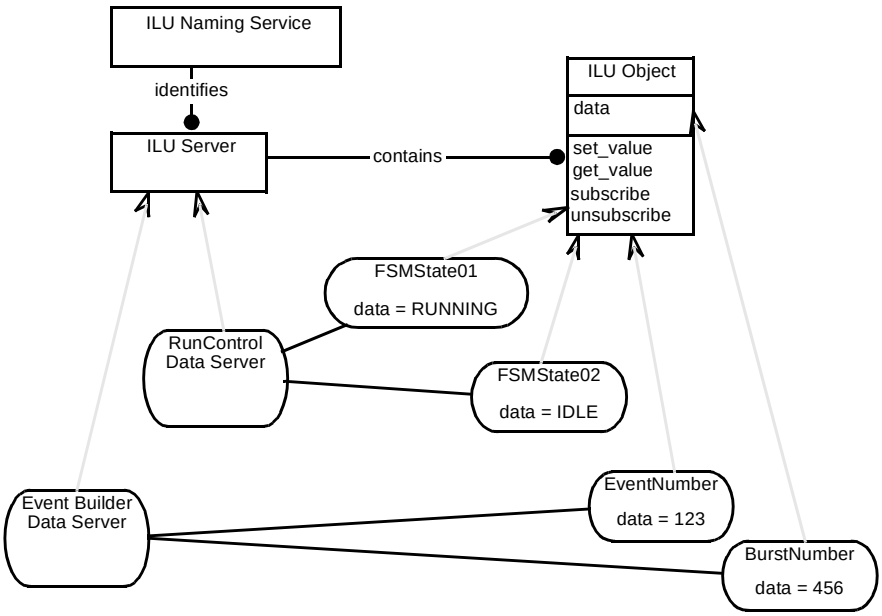
Figure 15 Information service event flow diagram



The ReceiverInterface allows receivers to **subscribe** or **unsubscribe** for information, indicating the information name and a delivery mode (only one value of information, for a periodic update or whenever information changes). The ReceiverInterface finds at NameServer the InfoServer where the information exists and contacts this server with a subscribing message or with a request for the last information value. If the information request is for a single value, the **get_value** method of the information object is called. If the subscription is for another delivery mode the request is passed to the information object adding the requesting receiver to the subscribing list. When information is updated the **receive** method of the ReceiverInterface is invoked. InfoServer can offer information about available services through the **query** operation.

3.6.2.1 Multiple servers architecture using ILU

Figure 16 Multiple servers architecture using ILU



This example shows that the information service can be implemented using concepts of the ILU system. The architecture of the information service looks very similar to the architecture of ILU system itself. The ILU Naming Service can be used to access a server, which are intended to store information. Information can be stored in ILU objects with remote access methods implementation.

The algorithm to determine how many servers are required and how information items are allocated to each server is not yet defined. The choice of the algorithm does not affect the design described in this paper.

3.7 Trigger/DAQ and Detector Integration Components

3.7.1 Partition and Resource manager

During setup and testing periods, many groups of people will work in parallel and require the use of various DAQ resources. The DAQ contains many resources (both hardware and software) which cannot be shared and so their usage must be controlled to avoid conflicts. The purpose of the Partition Manager is to formalise the allocation of DAQ resources and allow groups to work in parallel without interference.

The Partition Manager provides a context in which all components (e.g. run-control, MRS etc.) can operate. The operator may create and modify multiple partitions. The components and resources a partition contains can only be utilised when a partition is locked by the operator. A data taking run can only be made when a partition is locked. Any modifications to be made to a partition (e.g. add or modify a component or resource) can only be made when a partition is unlocked and hence not taking data.

Similar restrictions apply for DAQ resources which can only exist in a limited number of copies (e.g. only allowed to run 3 copies of a particular piece of software because of licence restrictions).

Currently, the design has been elaborated for resource management which constitutes the basis of partition management.

The configuration database (section 3.3 on page 26) contains a class that represents a limited resource. The resource class is used to describe shared and exclusive resources and gives the maximum number of copies per partition and per system (i.e. total). If access to a piece of software or hardware is limited then an instance of this resource class is created and associated to the application or piece of hardware. This operation on the database is performed off-line.

During run-time, when software applications are started that are associated to a resource class then a token must be requested for the resource from the resource manager. The resource manager controls the tokens for all resources. When it receives a request for a token it first checks what is the maximum number of tokens permitted (per partition or per system) as well as how many tokens have already been allocated. If the allocated number of tokens is less than the permitted limit then the new request is granted and another token allocated. If the maximum number of tokens have already been allocated then the request is refused. When an application terminates then any tokens it was allocated should be freed and made available for other applications. The user of the application is responsible for freeing any allocated tokens via the Resource Manager. The Resource Manager provides an interface for the operator to monitor and control resource allocation in order to avoid the system freezing if there is a problem.

The resource manager is a server on the LAN with global scope (one server for all partitions) to which clients make requests to reserve and release resource tokens. However, an approach where the Resource Manager is more closely linked to other DAQ back-end components such as the DAQ supervisor or Process Manager (section 3.5 on page 31) cannot be excluded.

In general some software applications can be associated to more than one instance of the resource class. For example, if an application is composed of two software modules which use different licensed widget libraries, it would be associated with two resource class instances.

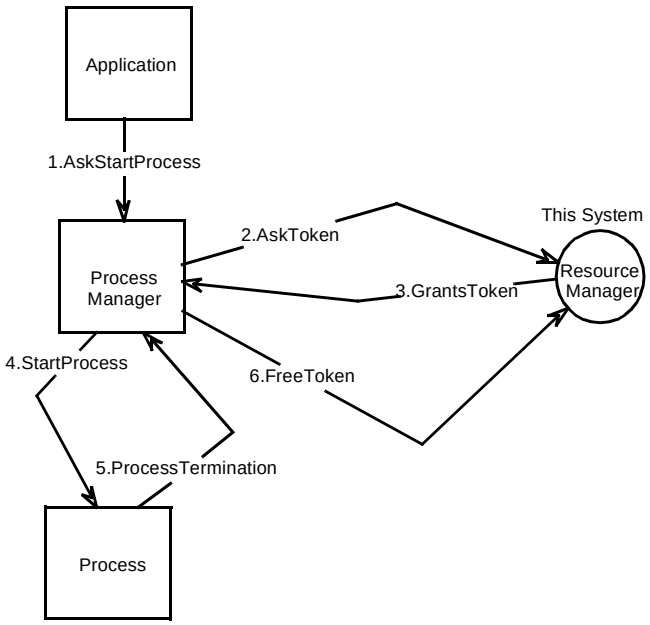
The Resource Manager controls tokens but not access to the associated resources themselves has no means to restrict access to resources other than token allocation. For example the Resource Manager cannot stop a user from starting an application that requires limited resources by hand.

It is the responsibility of applications (directly or indirectly via the process manager) to ensure they do not access limited resources without first acquiring the necessary tokens.

3.7.1.1 Relationship between the Process Manager and the Resource Manager

The Process Manager can act as the intermediate Resource Manager agent for applications. Thus applications can avoid use of the Resource Manager programming interface directly and allows the introduction of the Resource Manager at a point when resource allocation becomes critical. A typical scenario is shown below (see Figure 17).

Figure 17 Resource allocation via the Process Manager



An application asks the Process Manager to start a process in a given partition. The Process Manager asks the Resource Manager to allocate a token for the application. If the token is granted then the Process Manager starts the process. When the process terminates the Process Manager frees all resource tokens that were granted for this process via the Resource Manager.

The details of allocated tokens are stored in the DynamicDB together with the process handle in order to free the resource when the process terminates.

Since the Resource Manager controls rights to access resources it can be used in partitioning [17]. The Resource Manager partition class can be used for partition locking and unlocking. In such a manner the Resource Manager will check if the partition is locked before allocating resources.

3.7.2 Status display

The status display presents the status of the current data taking run to the user in terms of its main run parameters, detector configuration, trigger rate, buffer occupancy and state of the run controller.

The DAQ Status Display will obtain information to be displayed from the following sub-systems:

- configuration databases (run parameters, detector read-out)
- DAQ read-out creates (buffer activity and occupancy, counters, event rates, states of front-end tasks) via the LDAQ
- event builder (event rates, buffer activity) from the event builder supervisor
- run control (states of run control modules)
- processor farm supervisor (farm occupancy and activity)
- accelerator (beam-details)

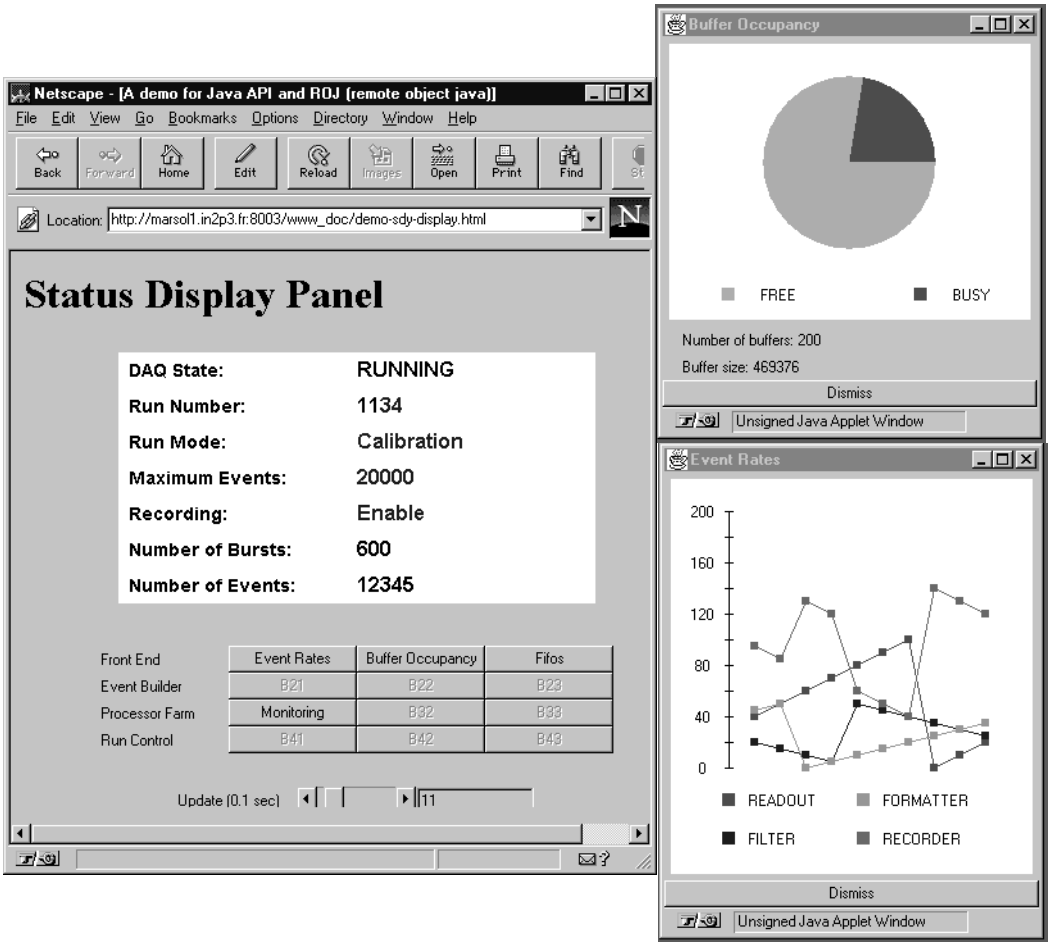
It is expected that most of this information will be made available via the Information Service. The status display presents information about:

For each piece of information an appropriate representation (digital, analog, tree form, etc.) will be used and will be presented in a hierarchical manner to permit the user to choose between summary and progressively more detailed information.

The information is updated at a rate determined by the user or when some significant events occur (change of the state of the control modules, start or end of run, etc.).

The main user of the DAQ Status Display will be the shift operator but will also be used by DAQ experts.

Figure 18 Elements of a DAQ status display implemented with Java



3.7.3 Event dump

An event dump task shall be provided for checking the integrity of sampled data. The event display task checks event integrity with respect to the contents in the read-out configuration database.

Such a tool shall be capable of the following functions:

- Dump data and perform range checks
- Display data in different modes
- Look for patterns in the events
- Check event size coherency with the detector read-out configuration

Monitoring in this document refers to the monitoring of physics data as it flows through the DAQ. The issue of monitoring is to provide a program skeleton with which users can build their own tasks for analysis. One of these tasks is the event dump which tests event integrity and is useful as a event structure debugging tool. It also acts as an example monitoring task and is developed by the DAQ group.

The monitoring skeleton must allow tasks to interface with the following back-end software components:

- Configuration database (run parameters, detectors read-out configuration)
- Process manager (to start/stop other processes)
- Message Reporting System to receive and send information and error messages
- Run Control for synchronization and control
- Information Service

In order to get physics data the monitoring task is linked to the data flow at a pre-defined point in the acquisition chain.

The monitoring task must be able to run on-line (i.e. during a DAQ run) and off-line for further data analysis.

3.7.4 Run bookkeeper

The purpose of the Run bookkeeper is to archive information about the data recorded to permanent storage by the DAQ system. It records information on a per-run basis and provides a number of interfaces for retrieving and updating the information.

The following data shall be archived for each DAQ run:

- run configuration (run #, mode, # events, detector read-out list)
- trigger state and configuration details, number of events taken per trigger type
- recording device details (device name, type, location, record size etc.)
- run dates (start/stop date and time)
- accelerator beam information (beam type, energy etc.)
- data quality (abnormal end of DAQ run etc.)
- error statistics
- structured comments (author, date, topic, title, free format body text)

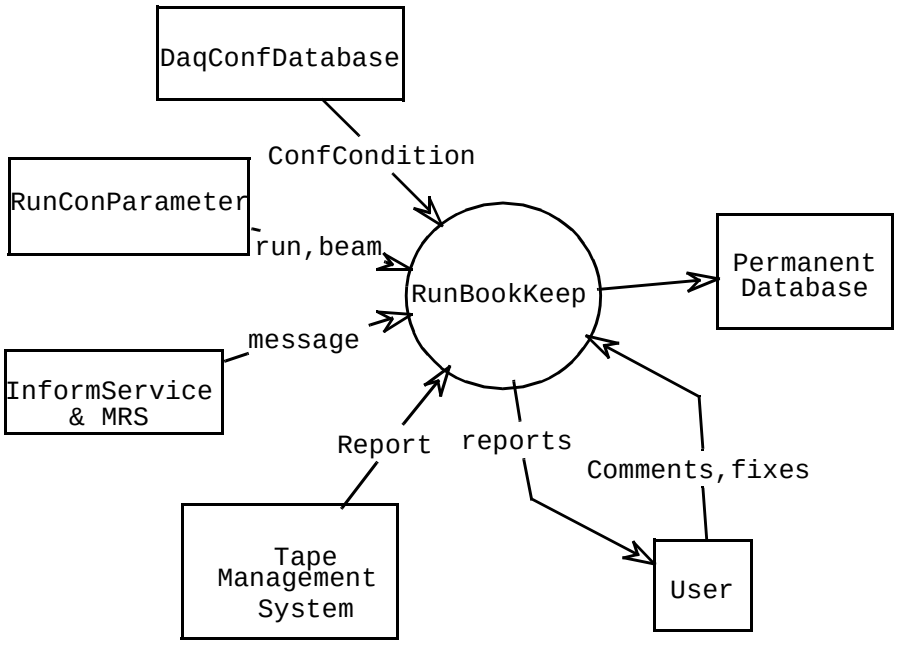
The Run bookkeeper is a typical database application that defines the schema of the data to be stored, and provides subsystems to collect the data from different sources, including the DAQ Message Reporting System, the Beam control software and the Tape Management System. These data storing procedures must be performed by three different threads that, in order to increase robustness, can be implemented as different processes.

The user interface will allow comments to be inserted for each run, on-site retrieving of data, and wide-area simple access to the stored parameters. This will be provided through an networked application that connects to an database server through an SQL interface. The user access to SQL is envisaged as form-based to perform ad-hoc contemplated queries to the database. The graphical interface will therefore contain an easy layout for the most common queries and a command line field that will allow users to issue more elaborate queries.

Five different subsystems have to be developed:

- The Database Schema in the Data Definition Language (DDL)
- The RBK_DAQ daemon process that will collect data from the DAQ and store it in the database.
- The RBK_Beam thread/process that will collect data from accelerator beam control and stores it in the database.
- The RBK_TMS thread/process that will collect data from the Tape Management System and associate this with the run information and update the database.
- The RBK_APPLET application that will run in a WWW browser and that will allow user interaction with the database.

Figure 19 Context diagram for the Run Bookkeeper



3.7.5 Test Manager

The purpose of the test manager is to provide a means of organising individual tests for hardware and software components. The individual tests themselves are not the responsibility of the test manager which simply assures their execution and verifies their output. The individual “atomic” tests are intended to verify the functionality of a given component. They will not be used to modify the state of a component or to retrieve status information. Tests are not optimized for speed or use of resources and are not a suitable basis for other components such as monitoring or status display.

The outcome of a test should comply to POSIX 1003.3 (i.e. Pass, Fail, Unresolved or Untested). However there should also be a mechanism to convey more elaborate and detailed results of tests from test to initiator of the test.

The Run Control system may test (software/hardware) resources when necessary before starting data taking using the Test Manager.

The Diagnostics Package performs (logical sequences of) tests to diagnose problems with the DAQ system and/or confirm its functionality, using the Test Manager.

A DAQ or detector expert may perform individual tests or test sequences, through an interactive application with a graphical user interface; this application may be part of the Diagnostics Package or of the Test Manager itself.

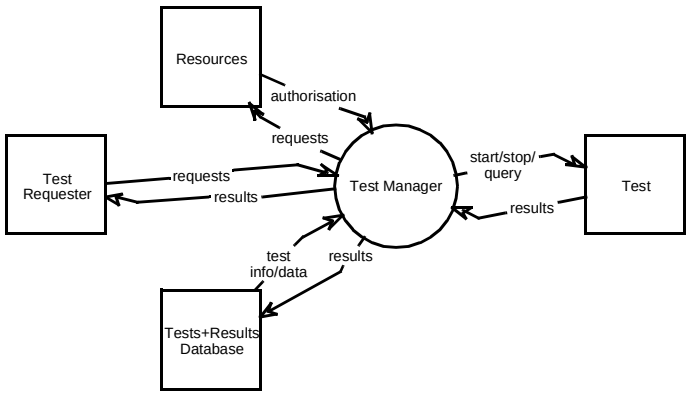
A test cannot be run when the hardware or software component to test is in use, by the DAQ for example (except possibly as part of the DAQ procedure itself).

There are two situations when tests need to be run are:

- on the detection of an error; the aim is to identify the problem and localise the error.
- to check the functionality of a component; this can be done:
    - (automatically) on a regular basis (in between ‘physics runs’)
    - (automatically) on the basis of the state of the system: at start-up, at start-of-run, at a change of configuration
    - at a user’s request

The Test Manager will use the Process Manager to start, stop and monitor the individual tests.

Figure 20 Context diagram for the test manager



3.7.5.1 Databases

Several databases to hold information related to the Test Manager will be needed:

- a database containing descriptions of the “atomic” tests; an “atomic” test is a single test that cannot be divided into smaller ‘subtests); the software implementing a test is provided by the component experts to test a component or a particular feature of the component; these test program descriptions -being software objects- could be provided by the Software Configuration Database.
- a database to hold the (history of) test results.
- a dynamic database containing the current status of ongoing tests; the services provided by the Information Service could be used for this purpose.
- a database containing logical sequences of tests; this could be part of the future Diagnostics Package.

3.7.6 Diagnostics Package

The diagnostics package uses the tests held in the test manager to diagnose problems with the DAQ and confirm its functionality. By grouping tests into logical sequences, the diagnostic framework can examine any single component of the system (hardware or software) at different levels of detail in order to determine as accurately as possible the functional state of components or the entire system. The diagnostic framework will report the state of the system at a level of abstraction appropriate to any required intervention.

Example uses of the diagnostics package could include:

- An off-line program that uses the configuration databases to verify that components of the DAQ are functioning normally. This task could be run whenever the DAQ configuration is changed.
- An interactive application with a graphical user interface that presents the list of all tests known to the test manager and allows the user to select and start tests and informs him of the result.
- An expert system that uses the test manager to diagnose problems with the DAQ and either suggest solutions or perform recovery actions itself.

It is expected that the diagnostics package will be one of the last components to be implemented since it requires the existence of many other components in order to function and is only useful when the DAQ system itself is relatively stable.

4 References

[1] ATLAS Technical Proposal, CERN/LHCC/94-43, 15 December 1994, (ISBN 92-9083-067-0)

[2] ATLAS DAQ Back-end Software User Requirements Document, PSS05-ATLAS-DAQ-SW-URD http://atddoc.cern.ch/Atlas/DaqSoft/document/draft_1.html

[3] ATLAS F/E DAQ Discussion Group Summary Document and Work Plan April 1996. http://atddoc.cern.ch/Atlas/FrontEnd/document/draft.ps.

[4] ATLAS Detector Control System User Requirements Document PSS05-ATLAS-DCS-URD

[5] ATLAS Detector Interface Working Group Summary Document, 2 April 1997 http://atddoc.cern.ch/Atlas/DetfeIf/Detinfo.ps

[6] Event Builder Working Group. Summary document and workplan. August 1996. http://atddoc.cern.ch/Atlas/EventBuilder/document/summary.ps.

[7] ATLAS Event Filter Project Definition document, 26 June 1997. http://wwwcn1.cern.ch/~ftouchar/URD.bk.ps.gz

[8] ATLAS Level-2 Trigger User Requirements: DAQ-No-79, November 03, 1997

[9] Tools.h++ Introduction and Reference Manual Version 6. Thomas Keffer, Rogue Wave Software Inc, 1994.

[10] A. Patel et al, Use of a CASE tool for developing ATLAS DAQ software, CHEP 97, Berlin Germany. Information on StP is available on WWW at: http://www.ide.com/

[11] LCRB Status Report / RD45. LCRB 96-15. CERN.

[12] Object Management Group. http://www.omg.org/.

[13] ILU documentation. Copyright (c) 1991-1996 Xerox Corporation. ftp://ftp.parc.xerox.com/pub/ilu/ilu.html.

[14] CHSM: A Language System Extending C++ for Implementing Reactive Systems. P. J. Lucas, F. Riccardi. http://www.best.com/~pjl/software.html.

[15] X-Designer Release 4.7. User's Guide, Imperial Software Technology Limited, 1997. http://www.ist.co.uk/.

[16] L.Mapelli et al., A scalable Data Taking System at a Test Beam for LHC, CERN/DRDC 90-64, CERN/DRDC91-23, CERN/DRDC 92-1, CERN/DRDC 93-25, CERN/LHCC 95-47. See also the World Wide Web at URL http://rd13doc.cern.ch/.

[17] D. Francis and T. Wildish, Partitioning issues in DAQ/EF prototype -1, ATLAS DAQ/Event Filter Prototype -1 Project Technical Note #60. http://atddoc.cern.ch/Atlas/Notes/060/Note060-1.html