ALICE Internal Note/DAQ
ALICE-INT-2010-001

ALICE DAQ and ECS Manual

December 2010

ALICE DAQ Project
ALICE ECS Project

Preface

ALICE [1] is a general-purpose detector designed to study the physics of strongly interacting matter and the quark-gluon plasma in nucleus-nucleus collisions at the CERN Large Hadron Collider (LHC). ALICE will operate in several different running modes with significantly different characteristics. The experiment has been primarly designed to run with heavy ions beams, which are characterized by relatively low rates (interaction rates <= 10 kHz for Pb-Pb beams at design luminosity of $L=10^{27}$ $cm^{-2}s^{-1}$), relatively short running time (order of few weeks per year) but very high multiplicity and correspondingly large event size. The requirements on the low level trigger selectivity are therefore relatively modest, whereas the trigger complexity is considerable and requires partial or full reconstruction in the high-level trigger. In addition, a large bandwidth DAQ together with efficient data selection and/or data compression in the High-Level Trigger (HLT) are required to collect sufficient statistics in the short running time available.

In proton-proton or proton-ion running mode, the interactions rates are much higher than in heavy-ion runs (up to 200 kHz, limited by pile-up in the TPC detector), whereas the event size is small and the running time is typically of several months per year in pp mode. Therefore, the requirements on trigger selectivity is increased while requirements on trigger complexity and bandwidth are much reduced.

The ALICE data-acquisition system has been designed to run efficiently in these different modes and to balance its capacity to record the very large events (several tens of MBytes) resulting from central PbPb collisions with an ability to trigger and acquire rare cross section processes.

These requirements result in a readout capability of up to 40 GByte/s, an aggregate event-building bandwidth above 2.5 GByte/s and a storage capability up to 1.25 GByte/s to mass storage.

The software framework of the ALICE DAQ is called DATE (ALICE Data Acquisition and Test Environment) and consists of a set of software packages described in the Part 1 of this guide.

The global control of the experiment is ensured by the Experiment Control System (ECS) and the ALICE Configuration Tool (ACT) which are described in Part 2.

The standalone software developped for the Detector Data Link (DDL) via the DAQ Read-Out Receiver Card (D-RORC) is documented in Part 3.

The Part 4 describes the Detector Algorithm framework (DA).

The data quality monitoring is performed with the AMORE software package documented in Part 5.

The monitoring of the DAQ system itself is performed by the Lemon package described in the Part 6.

The Part 7 is dedicated to a description of the electronic logbook.

This User's Guide can be found in the ALICE EDMS:

https://edms.cern.ch/document/1056364/

and on the ALICE DAQ web site [19].

The Authors

F. Carena, W. Carena, S. Chapeland, V. Chibante Barroso, F. Costa, E. Dénes, R. Divià, U. Fuchs, G. Simonetti, C. Soós, A. Telesca, P. Vande Vyvre, B. Von Haller.

Important note on software versions

This user's guide describes the behavior of the software versions listed in Table 0.1. When using different versions of the packages, it is recommended to read the associated release notes.

Table 0.1 Software versions corresponding to this guide

Package	Version
DATE	7.00
ECS	4.00
RORC library	5.3.8
AMORE	1.24
LEMON	2.15.0

Contents

Part III
DDL and D-RORC software Reference Manual

Part IV
Detector Algorithms Framework

Part V
Data Quality Monitoring

Part VI
The ALICE electronic logbook

Part I

DATE Reference Manual

November 2010

ALICE DAQ Project

DATE V7

DATE overview

1

This chapter gives an overview of the architecture of the ALICE DAQ system and of its software framework called DATE. The features of the system are described, with the components that implement such features. For each component, a brief explanation of the underlying mechanism is given.

1.1 ALICE data-acquisition architecture

A broad view of the ALICE data–acquisition architecture is illustrated in Figure 1.1.

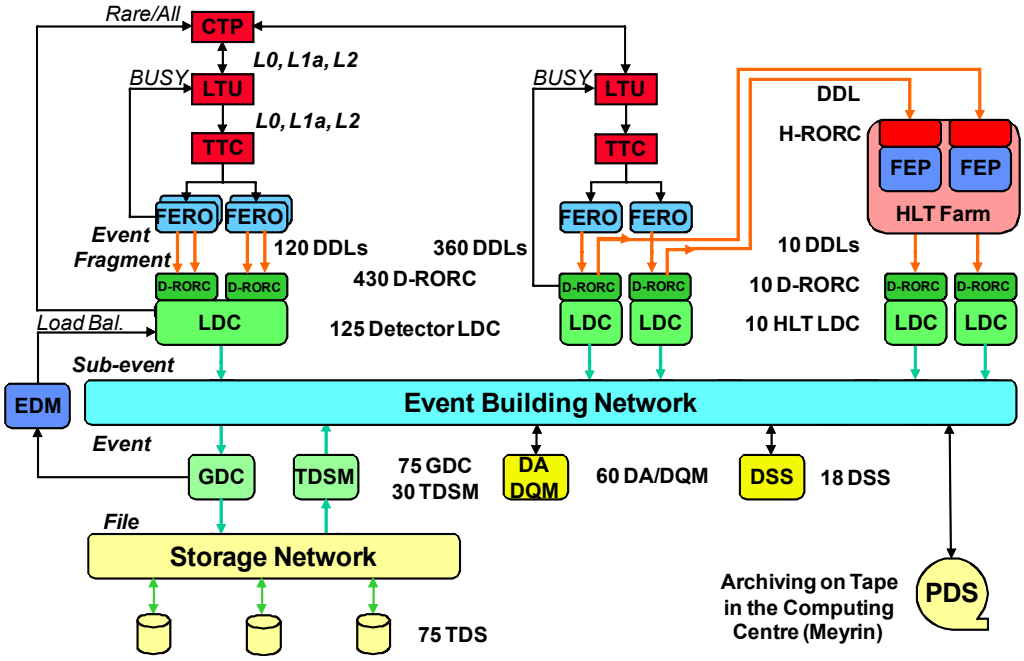

Figure 1.1 DAQ architecture overview.

The detectors receive the trigger signals and the associated information from the Central Trigger Processor (CTP), through a dedicated Local Trigger Unit (LTU) interfaced to a Timing, Trigger and Control (TTC) system. The readout electronics of all the detectors is interfaced to the ALICE standard Detector Data Links (DDL). The data produced by the detectors (event fragments) are injected on the DDLs.

At the receiving side of the DDLs there are PCI-X or PCI-e boards, called DAQ Read-Out Receiver Cards (D-RORC). The D-RORCs are hosted by PCs, the Local Data Concentrators (LDCs). Each LDC can handle one or more D-RORCs. The D-RORCs perform concurrent and autonomous DMA transfers into the PCs memory, with minimal software supervision. In the LDCs, the event fragments originated by the various D-RORCs are logically assembled into sub-events. The role of the LDCs is twofold. Either it can take data isolated from the global system for a test or a calibration run or it can ship the sub-events to a farm of PCs called Global Data Collectors (GDCs), where the whole events are built (from all the sub-events pertaining to the same trigger).

The D-RORCs include 2 DDL channels which can be used in two different ways: either both as input from the detector or one as input and the other one as output to the High-Level Trigger (HLT). In the later case, the data shipped by the detector are copied to the HLT for software triggering or data compression.

Besides having a DDL common to all the sub-detectors, the other major architectural feature of the ALICE data acquisition is the event builder, which is based upon an event building network.

The sub-event distribution is performed by the LDCs, which decide the destination of each sub-event. This decision is taken by each LDC independently from the others (no communication between the LDCs is necessary); the synchronization is obtained using a data-driven algorithm. The algorithm is designed to fairly share the load on the GDCs.

The event–building network does not take part in the decision about the destination; it is a standard communication network supporting the TCP/IP protocol. The event-building network is also used to distribute the HLT decisions from the HLT LDCs to the detector LDCs where the decisions to accept or reject sub-events are applied.

The role of the GDCs is to collect the sub-events, assemble them into whole events, and record them to the Transient Data Storage (TDS) located at the experimental area.

The data files recorded on the TDS are migrated by the TDS Managers (TDSM) onto Permanent Data Storage (PDS) in the computing centre.

The services needed by the DAQ system itself such as the control or the database are performed by the DAQ Services Servers (DSS). Additional servers are used to run the Detector Algorithms (DA) or the Data Quality Monitoring (DQM). All these servers are connected to the event–building network to exchange commands, status and data messages with the other nodes of the system.

1.2 DATE overview

DATE (Data Acquisition and Test Environment) is a software system that performs data-acquisition activities in a multi-processor distributed environment. DATE fulfills the requirements of the ALICE data acquisition, therefore it has been designed with scalability features that make it suitable for large systems, involving hundreds of computers. Nevertheless, DATE can cope with a large variety of configurations; in particular, it is well adapted to small laboratory systems as well, where only few machines are used, or even just one. In that case, the DATE system may be based on one single processor, which will then perform all the functions (LDC, GDC, run control, monitoring, etc.).

The basic dataflow is organized along parallel data streams working independently and concurrently, followed by a stage of event builders where data are merged and eventually recorded as a complete event.

The conditions imposed to the hardware architecture in order to support DATE are minimal:

a. The processors must be of the IA32 or IA64 families.

b. The operating system of all the processors must be Linux.

c. All the processors must be linked to a network supporting the TCP/IP stack

and the socket library.

The readout program contains a piece of code that deals with the devices to be read. This piece of code can be tailored to read any type of devices. ALICE, though, has currently standardized its detector readout channel and uses the DDL and the D-RORC; the software to handle this type of device is available and remains the same for all the detectors using it.

In view of the ALICE upgrade, new types of readout links will be supported. The support for Ethernet coupled with the UDP protocol has for example been added to the DATE readout.

The event triggering is performed via the TTC. The readout program will collect all the data from the DDLs, and the data structure superimposed by the DDL will permit to identify the original blocks belonging to an event.

The DATE system, besides the data-flow function, provides many other features, such as the ones described in the following paragraphs.

1.2.1 Parametrization of the hardware configuration

The hardware configuration of the system is described by declaring all the available machines and assigning a role to them (LDC, GDC, run control, monitoring, etc.). DATE uses a database repository to obtain this information. The repository is made of records in a SQL database containing the description of all the entities and their relationships. The setting up of the hardware configuration is achieved by editing the records of the database.

1.2.2 Interactive setting up of the data-acquisition parameters

The running conditions of the system are described by selecting the machines involved in the data acquisition (which may be a subset of the available machines) and assigning the parameters associated with a given mode of operation. This information is stored on disk and may be changed interactively.

1.2.3 Run control

An interactive program gives to the operator the opportunity to centrally control the operations of all the machines involved in the data acquisition. The activities of all the machines in the system proceed through pre-defined sequences with synchronization check-points. Various hooks are provided to perform calibration procedures and to submit foreign data into the event stream.

1.2.4 Load balancing

Large configurations, involving a farm of many GDCs, may need to smooth the distribution of events to the various machines, in order to avoid that busy machines slow down the system. A module called event-distribution manager (EDM) checks the occupancy of each GDC and instructs the LDCs to dispatch the events to the machines that are not crowded.

1.2.5 Event monitoring

Analysis programs can receive online events, while the data acquisition is active. A monitoring server makes copies of the events requested and dispatches them to the client analysis process. The analysis process does not need to run on the machine where the data are generated. Actually, the analysis can run on any remote non-DATE machine, i.e. not declared as a node of the DATE system.

The same routine calls that provide the online events may be used to read offline events that have been previously recorded.

1.2.6 Information reporting

All the information messages generated by the processes involved in the data acquisition are centrally handled and made available to the operator via an interactive browser.

1.2.7 Electronic Logbook

All the information relevant to the runs (used to keep track run-by-run of the running conditions) may be generated by any process involved in the data acquisition. It is centrally handled and made available to the operator via a Web browser.

The electronic logbook can also be used to archive comments or observations made by the people working at the experimental area.

1.2.8 Performance monitoring system

The performance of large systems should be closely monitored. The ALICE DAQ uses the LEMON package to perform the collection of performance measurements, their centralized handling, and their visualization using a Web browser.

1.2.9 Detector algorithms

A framework has been developed to support in the DAQ system the execution of detector algorithms using data monitored or recorded.

1.2.10 Data Quality Monitoring

The AMORE framework has been developed to allow the execution of Data Quality Monitoring (DQM) programs. These programs monitor physics data during the physics run and accumulate plots that can be inspected asynchronously. The DQM framework also provides an archiving of the plots at various stages of their lifetime in order to ease their inspection and any investigation related to their evolution.

1.3 DATE architectural strategies

Some of the leading ideas that have determined the DATE architecture are described in the next paragraphs.

1.3.1 Protocol-less push-down strategy

Data are pushed down as soon as available. All the actors of the data acquisition, from the detector electronics to the data storage, send the data through open channels to the next processing stage, as soon as they finished their own processing. The DDL and TCP/IP provide the flow control. A back-pressure mechanism (x-on/x-off style) protects the system from congestions. This strategy avoids the synchronization overheads and maximizes the throughput.

1.3.2 Detector readout via a standard handler

The fact of standardizing the transmission medium (presently the DDL) and the data structure allows to provide the same piece of code (***equipment*** code) to handle all the detectors using the same medium. The readout system can be adapted to changes of the hardware configuration without modifying the code.

The addition of a new medium such as Ethernet coupled with the UDP protocol has been made possible by the development of a new readout ***equipment***.

1.3.3 Light-weight multi-process synchronization strategy

Wherever process synchronization is required within a PC, no system services are used, such as semaphores or message queues. An original technology has been developed to be able to use much faster shared-memory mechanisms. This technology is applicable each time the synchronization involves one single data producer and one data consumer.

1.3.4 Common data-acquisition services

DATE provides a set of services, such as run control, event delivery to monitoring programs, message logging, run bookkeeping, load balancing, performance measurement and data quality monitoring. These services are common throughout all the components of the system and are available to any additional piece of software cooperating with DATE.

1.3.5 Detectors integration

The detectors developers usually provide the code dealing with the various operation phases, such as calibration, initialization, run-down and readout. This code can be fully integrated in DATE and makes use of all the services mentioned above.

1.3.6 DATE installation

A user-friendly DATE installation procedure, based on RPMs, produces a turn-key system readily available to the user.

2 DATE configuration parameters

This chapter gives an overview of the configurable parameters for a DATE system.

2.1 DATE site parameters

Since DATE v6 only MySQL is used to store the DATE configuration parameters. As a consequence, a single local configuration file is needed to run DATE (it stores the database access parameters). All the other configuration items are stored in MySQL, and edited with ***editDb*** (see Section 4.5) or some other package-specific human interfaces. The configuration files are retrieved locally when necessary.

The only item that should still be put manually on each host running DATE is the file ***${DATE_SITE_PARAMS}***. It contains a sequence of lines defining environment variables. Every line contains the name of an environment variable followed by the associated value. Lines starting with character **#** (followed by a space) are comments and are not taken into account. The following variables must be defined so that the configuration database is accessible:

- ***DATE_DB_MYSQL_HOST :*** IP name of the host where the MySQL server runs.
- ***DATE_DB_MYSQL_DB :*** MySQL database name.
- ***DATE_DB_MYSQL_USER :*** user name to access MySQL database.
- ***DATE_DB_MYSQL_PWD :*** password to access MySQL database.

2.2 Base configuration

To initially setup a minimal working DATE setup, it is recommended to call the interactive script `newDateSite.sh`.

Answer the questions accordingly to your local system and you will have a basic DATE environment running. This script includes the creation of the ***DATE_SITE_PARAMS*** file described in Section 2.1 and the setup of some services like the ***infoLogger*** system (Chapter 11) and ***logbook*** (Chapter 24). These settings are in principle final and do not need to be changed afterwards. The script also creates a minimal setup with a random software readout in a 1 LDC + 1 GDC configuration on the same machine. It can then be extended according to your needs.

Please note that for an initial DATE installation, some local system services (DIM DNS, firewall, database engine, xinetd, ...) need to be configured. One may run the script `newMySQL.sh` to initially create databases and accounts for DATE. One can also use the script `dateLocalConfig` to configure local services. Finally, some DATE daemons may be started with `dateSiteDaemons` and `runControl/start_daqDomains.sh`. Extensive installation instructions are available in separate guides:

- ***ALICE DAQ and ECS installation and configuration (hardware and software) at Point 2*** (available in the ALICE DAQ WIKI pages);
- ***ALICE DAQ and ECS installation and configuration guide for external sites*** (available on the ALICE DAQ Web server).

Configuration information for roles, detectors, event-building rules, memory banks, triggers and readout equipment is required to operate DATE.

The DATE utility ***editDb*** should be used to populate or edit configurations. Chapter 4 describes the configuration of roles, detectors, banks, triggers, and event building rules. Equipment-specific parameters are described in the corresponding hardware chapters.

Additionnal package-specific configuration files or environment variables may be stored in the database ***FILES*** and ***ENVIRONMENT*** sections. Description of the files or variables is done in the relevant packages documentation.

Some persistent DATE parameters are also stored in the database and not directly accessible by users from ***editDb***. This is for example the case of the runcontrol parameters edited using the ***runControlHI*** human interface, as described in Chapter 14.

After a basic ***DATE_SITE*** system setup, the first parameters usually modified are:

- The ***ROLES*** database in order to add new machines or roles to the DATE system. This is described in Chapter 4.
- The ***EQUIPMENTS*** configuration, to add and configure hardware readout equipment. This is done in ***editDb***, and the parameters are described in the relevant hardware chapters, e.g. Section 7.2.4.4 for the RORC parameters.
- The ***runControl*** parameters, which define the behavior of the DATE processes at run time: e.g. maximum number of events, enabling CDH checks, enabling monitoring, etc. Note that the ***LOGLEVEL*** controlling the verbosity of some DATE processes is also one of these settings. The global run options (e.g. recording mode) may also be saved. This is described in Chapter 14.
- The ***monitoring*** configuration, e.g. to define adequate buffer sizes. These settings are commented in Section 5.8.
- The ***mStreamRecorder*** configuration to record ROOT files, as described in Section 10.5.

Other features, like the Transient Data Storage (Chapter 26) and the Detector Algorithms (Chapter 22) are usually deployed only at the production area, and therefore rarely need to be configured by end users.

2.3 Use of hostnames vs. IP addresses.

During the configuration of DATE, it is necessary to identify several hosts (DIM DNS, database server, infoLogger host). These can be specified either by hostname or by their IP address. The two methods are in principle equivalent. However, they offer different runtime features that may have an impact on the operation of a DATE site.

When hosts are specified by their hostname, this means that one or more calls to the Internet Name Domain server (named) are done at run-time in order to associate the name to the IP address of the machine. We recommend - whenever possible - to use IP addresses rather than hostnames during the configuration of a DATE site to

minimize queries to the IP name server, and avoid problems if it is unavailable or slow.

Data format

3

This chapter describes the different event types used in DATE, the format of the data produced in the LDCs (events and sub-events), and the format of the super event which are built in the GDCs.

3.1 Conventions

All symbols referred in the following pages are defined in the DATE include file ***${DATE_COMMON_DEFS}/event.h***. Programs using this file are also supposed to be compiled using the DATE makefile rules appropriate to the host architecture. The shell command `date-config` can be used - on all architecture compatible with the full DATE kit - to get a list of the options required to compile programs making use of the ***event.h*** definition file.

Several of the macros defined in the DATE environment perform some simple run-time checks. These checks can be disabled by defining the compilation symbol ***NDEBUG***. When the checks are enabled, they may cause an early termination of the process with an appropriate error message and the creation (if possible) of a core dump file as soon as the basic correctness conditions are not met.

Three concepts used in this chapter are those of **ID**s, **pattern**s and **mask**s. An **ID** is a number, belonging to a fixed range, that identifies one and only one entity in a given set (e.g. a trigger class or a detector). A **pattern** is a sequence of bits with one bit for each **ID** of a given set: it can have zero or more bits asserted. A **mask** is a **pattern** with one and only one bit asserted. These concepts are, for example, used in the definition of the ***eventTriggerPattern*** given in Section 3.5.7.

All sizes given in this chapter are expressed in bytes.

3.2 Base header and header extension

All DATE events are prefixed by an event header. This header is made of a first mandatory part (the base header) and of an optional header extension.

The base header is completely defined by the ***eventHeaderStruct*** structure. All the fields of the base header are initialized by DATE to predefined values. The base header includes the size of the complete event, a unique pattern (the DATE event magic number), the version of the ***eventHeaderStruct*** structure, all the fields needed to identify the event (in type, origin, trigger set and detector set) and other fields used for rule-driven criteria. A 64-bit pattern is also available to specify user-defined attributes associated to the event.

A header extension can be appended to the base header: the size and the format of the header extension is left to the data-acquisition system responsible.

3.3 Streamlined and paged events

LDCs must be able to handle synchronous and asynchronous equipments, with either fixed or variable event sizes and with segmented or paged payloads. Two different schemes have been made available: the streamlined events scheme and the paged events scheme. Streamlined events support mainly serial synchronous

channels to be read in strict sequence. Paged events are better suited for asynchronous equipments to be read in parallel. One should select the scheme that better fulfills the requirements of the DATE site.

Within the same run, each machine will handle events of the same type, e.g. only streamlined or only paged events. It is forbidden to switch between modes within one stream of events (online or offline). However, it is possible to have a DAQ system where some LDCs handle paged events and some other LDCs handle streamlined events.

DATE events at LDC level always have their payload partitioned into equipments. The equipment is a logical entity controlling a (set of) physical input channel(s). All equipments prefix their data with a standard ***equipmentHeaderStruct*** structure designed to identify the equipment itself, to provide some standard description of the channel and to associate some attributes to the payload.

GDCs do not produce paged events. Their format is based on UNIX I/O vectors (as in the `writev` system call). GDCs do not support the concept of equipment. All they do is to receive sub-events from the LDCs, eventually perform event building functions (according to the event-building rules defined in the configuration of the DATE site), add a ***super event header*** and send the resulting event to the recording stage (see Section 3.6 for more details on this process). The format of the events at the level of the GDCs is not described here.

3.3.1 Streamlined events

Streamlined events are made of a consecutive sequence of bytes, starting from the base event header followed by the header extension (if present) and by the equipments, one after the other. These events are designed to be read sequentially, typically equipment after equipment. This is the natural approach towards network channels or non-shared channels (such as RS232).

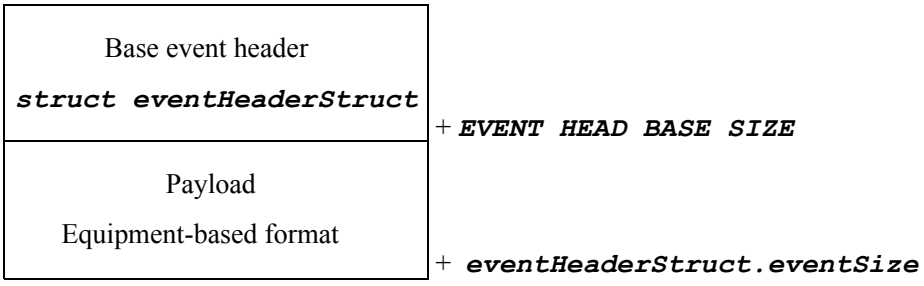

Figure 3.1 Streamlined unextended event format

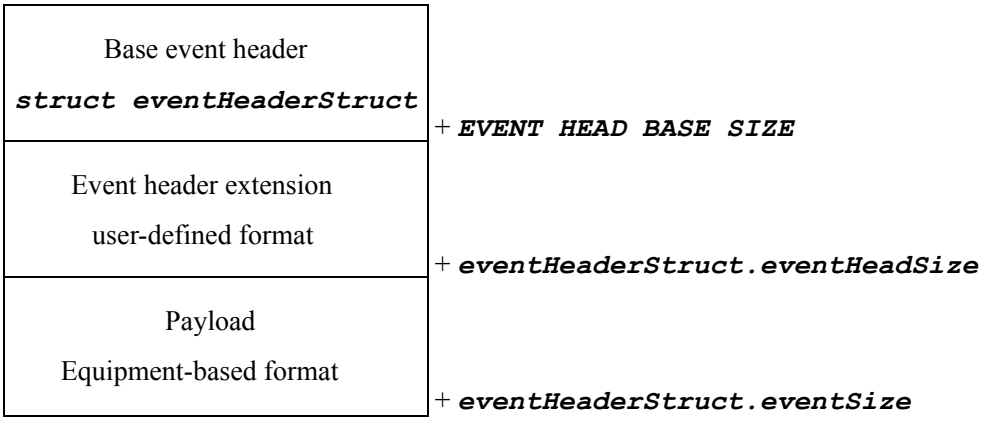

Figure 3.2 Streamlined extended event format

Multiple sequential channels can also be read in pure streamlined mode. The easier approach is to read the channels one after the other, in strict synchronous sequence. This requires the pre-allocation of an event buffer big enough to store the data coming from all the equipments, to be eventually resized at the end of the readout procedure. However, if the amount of data sent over each channel is known in advance, it is possible to allocate the buffer needed for the full event at once, calculate the offset of each equipment's payload and start a parallel, asynchronous readout from all the channels into the right location. Another option is to read all channels in parallel in separate buffers and them combine then into one single streamlined events using a follow-and-copy process. Streamlined events may be quite difficult to modify.

3.3.2 Paged events

Paged events are made of multiple data segments or pages, containing (part of) payloads coming from the input channel(s). This class of events are very efficient for data-driven input channels (such as the DDL) and for parallel, asynchronous input channels schemes, e.g. multiple serial lines. The logical organization of a paged event, starting from the first-level vector used to represent it, is described in Figure 3.3.

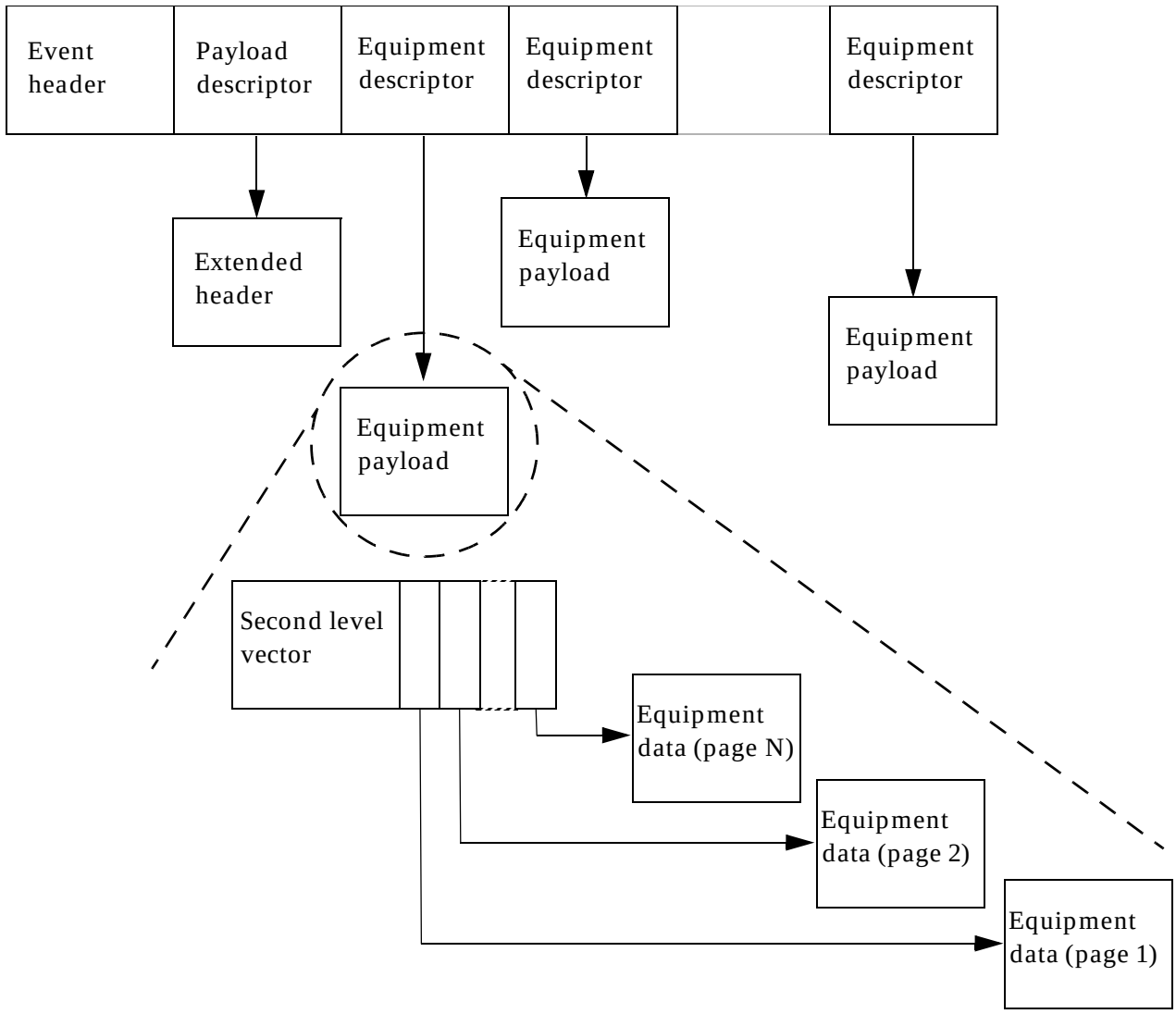

Figure 3.3 Paged event logical format

Paged events are made of a first-level vector including the event header, a payload descriptor and a set of one or more equipment descriptors. The pages with the actual data (extended header and payload) are described by various fields of the first level vector. Note that both the extended header and the equipment payloads are optional and may not be present in all paged events. A first level vector must have at least one equipment descriptor.

The first-level vector has a number of components (and therefore a size) function of the number of the equipments instantiated on the LDC. Its format is described in Figure 3.4.

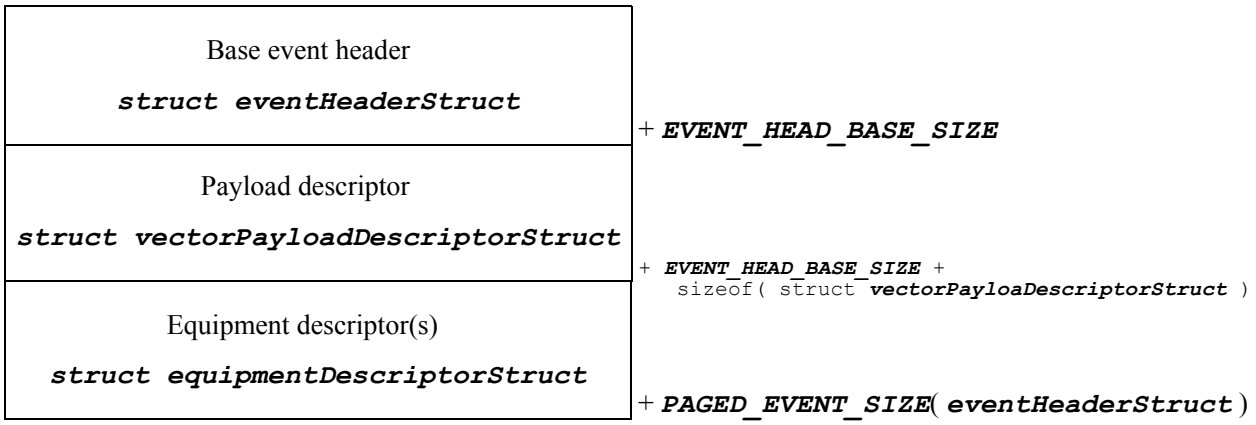

Figure 3.4 Paged event first-level vector format

The actual payload (extended header and equipment payload) is only described by the first-level vector. As a matter of fact, the entity pointed by the first-level vector can itself be a vector called second-level vector, used to represent paged payloads (such as the output from the DDL). In the example given in Figure 3.3, the first equipment has created a paged event made of N pages and described by a second level vector. Equipments creating segmented payloads can avoid the use of the second level vector and let the first level vector point directly to the payload.

The representation could be extended beyond two levels of vectors (if this need arises).

Paged events can be converted into streamlined events by algorithm of follow-and-copy. This is the approach followed by the DATE recording library where paged events are recorded (to file, pipes or over the network) in a strict sequential manner (although data is not explicitly copied in the process).

Paged events allow easy manipulation. It is sufficient to update the pointer(s) to the appropriate page(s) to modify selected portions of the payload. This avoid a lengthy follow-and-copy process as in the case of streamlined events.

3.4 Collider and fixed target modes

DATE events can be identified in two different modes: COLLIDER and FIXED TARGET. The main difference between the two modes is the way the event ID field of the base event header is loaded and handled.

In COLLIDER mode, events are identified as described in Figure 3.5. The components of the event ID are the period counter (software controlled), the orbit counter (from the machine / trigger systems) and the bunch crossing number (from the machine / trigger systems). The bunch crossing number directly comes from the particle accelerator while the orbit counter is an entity still driven by the machine that can be - from time to time - reset under software control. When this happens, a new run period is started: this is identified by the period counter.

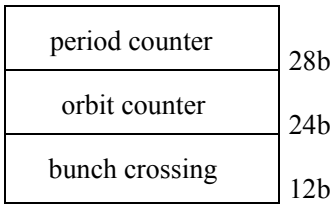

Figure 3.5 Collider mode event identification

In FIXED TARGET mode the event ID is under full software control. The format is described in Figure 3.6. This mode is compatible with both fixed-target and stand-alone setups. In the first case, burst number and number in burst can be included in the event ID. For stand-alone setups, only the number in run field should be set: the burst number and number in burst fields can be zero.

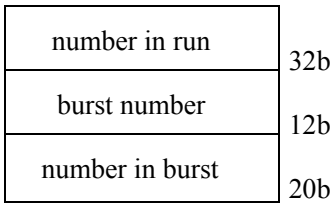

Figure 3.6 Fixed target mode event identification

Within the same stream of events it is not allowed to switch between different modes. The ***ATTR_ORBIT_BC*** system attribute bit can be used to select the encoding (collider if set, fixed target if unset) of the ***eventId***. Different macros are available to manipulate (initialize, load, compare, increment) event IDs of equivalent type.

3.5 The base event header

A DATE event is always prefixed by a base event header, described by the ***eventHeaderStruct*** structure. This structure include several fields that are standard to all events - such as IDs, event type, system attributes - plus some more static information used to identify the base header itself and its representation.

The structure of the base event header is described in Table 3.1.

Table 3.1 Base event header structure

Name	Type	Content	Set by
eventSize	***eventSizeType***	total size of the event	***readout eventBuilder***
eventMagic	***eventMagicType***	unique DATE event signature	***readout eventBuilder***

Table 3.1 Base event header structure

Name	Type	Content	Set by
eventHeadSize	***eventHeadSizeType***	size of the header (base + extension)	***readout eventBuilder***
eventVersion	***eventVersionType***	base event header structure version	***readout eventBuilder***
eventType	***eventTypeType***	type of event	***readout eventBuilder***
eventRunNb	***eventRunNbType***	number of the run associated to the event	***readout eventBuilder***
eventId	***eventIdType***	unique event identification	***readout eventBuilder***
eventTriggerPattern	***eventTriggerPatternType***	level 2 trigger pattern associated to the event	***readout eventBuilder***
eventDetectorPattern	***eventDetectorPatternType***	detector pattern associated to the event	***readout eventBuilder***
eventTypeAttribute	***eventTypeAttributeType***	attributes associated to the event	***readout eventBuilder***
eventLdcId	***eventLdcIdType***	ID of the LDC source of the event	***readout eventBuilder***
eventGdcId	***eventGdcIdType***	ID of the GDC source or destination of the event	***readout eventBuilder***
eventTimestampSec	***eventTimestampSecType***	Timestamp at the creation of the event (seconds)	***readout eventBuilder monitoring***
eventTimestampUsec	***eventTimestampUsecType***	Timestamp at the creation of the event (microseconds)	***readout eventBuilder monitoring***

A program is included in the DATE distribution kit to dump the base header of any event written in DATE format. This tool is available in the monitoring package and it is called `eventDump` (see Section 5.6).

We will now describe the fields of the base event header and their associated symbols and macros.

3.5.1 eventSize

It contains the total size of the event (base header, extended header, equipment header(s) and payload(s)) in bytes. The size must be a multiple of 32 bits. For paged

events, this field shall contain the same value as for the ***eventSize*** field of the streamlined version of the same event.

3.5.2 eventMagic

The ***eventMagic*** field contains a “magic” signature used for two purposes:

1. establish the correctness of the ***eventHeaderStruct***, eventually re-synchronizing a corrupted data stream,
2. determine the endianness of the event (header and payload) when this is received over a binary data channel, possibly originating from an architecture with different endianness (network, disk).

The two symbols ***EVENT_MAGIC_NUMBER*** and ***EVENT_MAGIC_NUMBER_SWAPPED*** can be used to detect at run-time the need to apply endianness-correction algorithms.

3.5.3 eventHeadSize

The ***eventHeadSize*** field contains the length in bytes of the event header (base + extension). This length should be greater or equal to ***EVENT_HEAD_BASE_SIZE***. For paged events it is always equal to ***EVENT_HEAD_BASE_SIZE*** (paged events’ headers can be extended only via a pointer from the payload descriptor, as shown in Figure 3.3). The size of the event header must be a multiple of 32 bits.

3.5.4 eventVersion

The ***eventVersion*** field provides the version ID of the base event header used to create the event itself. The symbol ***EVENT_CURRENT_VERSION*** is available to identify the event header structure as defined at compile time.

3.5.5 eventType

All DATE events have an associated type used to identify the content of the payload. The possible event types are:

- ***START_OF_RUN***
- ***START_OF_RUN_FILES***
- ***START_OF_BURST***
- ***PHYSICS_EVENT***
- ***CALIBRATION_EVENT***
- ***START_OF_DATA***
- ***END_OF_DATA***
- ***SYSTEM_SOFTWARE_TRIGGER_EVENT***
- ***DETECTOR_SOFTWARE_TRIGGER_EVENT***

- ***END_OF_BURST***
- ***END_OF_RUN_FILES***
- ***END_OF_RUN***
- ***EVENT_FORMAT_ERROR***

The primary use of the event type field is to identify each type of event or record and determine the type of processing to be applied. The event type is used for example by the ***eventBuilder*** to determine whether the policy to be applied on a given event (build, partial build or no-build).

The symbols ***EVENT_TYPE_MIN*** and ***EVENT_TYPE_MAX*** are defined to support arrays and enumerated types. Arrays can be defined with [***EVENT_TYPE_MAX*** - ***EVENT_TYPE_MIN*** + 1] range and addressed as [***eventHeaderStruct***.eventType - 1].

The macro ***EVENT_TYPE_OK*** can be used to test a possible event type, e.g. ***EVENT_TYPE_OK***(***eventHeaderStruct.eventType***) will return ***TRUE*** if the content of the ***eventType*** field can be associated to one of the event types above.

The ***START_OF_RUN*** and ***END_OF_RUN*** events can (and should) have the system attributes (***ATTR_P_START*** and ***ATTR_P_END***) set to point to the start and to the end of each phase (see Section 3.7 for more information on this subject).

3.5.6 eventId

The ***eventId*** field must be handled according to the identification system in use (COLLIDER or FIXED TARGET). The system attribute ***ATTR_ORBIT_BC*** shall be set for COLLIDER mode and cleared for FIXED TARGET mode. Macros are provided to handle the ***eventId*** field. Some macros apply only to one particular mode while other macros can be used for any type of event.

If the ID is encoded in COLLIDER mode (***ATTR_ORBIT_BC*** set), the following macros can be used:

- ***LOAD_EVENT_ID(***
  eventHeaderStruct.eventId,
  period,
  orbit,
  bunchCrossing)
  load the given ***eventId*** with the given period, orbit and bunch crossing
- ***EVENT_ID_SET_BUNCH_CROSSING(***
  eventHeaderStruct.eventId ,
  bunchCrossing)
  set the bunch crossing field of the given ***eventId*** with the given value
- ***EVENT_ID_SET_ORBIT(eventHeaderStruct.eventId, orbit)***
  set the orbit field of the given ***eventId*** with the given value
- ***EVENT_ID_SET_PERIOD(eventHeaderStruct.eventId, period)***
  set the period field of the given ***eventId*** with the given value
- ***EVENT_ID_GET_BUNCH_CROSSING(eventHeaderStruct.eventId)***
  get the bunch crossing field of the given ***eventId***
- ***EVENT_ID_GET_ORBIT(eventHeaderStruct.eventId)***

get the orbit field of the given ***eventId***

- ***EVENT_ID_GET_PERIOD(eventHeaderStruct.eventId)***
  get the period field of the given ***eventId***

If the ID is encoded in FIXED TARGET mode (***ATTR_ORBIT_BC*** cleared), the following macros can be used:

- ***LOAD_RAW_EVENT_ID(***
  eventHeaderStruct.eventId,
  numberInRun,
  burstNumber,
  numberInBurst)
  load the given ***eventId*** with the given number in run, burst number and number in burst
- ***EVENT_ID_SET_NB_IN_RUN(***
  eventHeaderStruct.eventId,
  numberInRun)
  set the number in run field of the given ***eventId*** with the given value
- ***EVENT_ID_SET_BURST_NB(***
  eventHeaderStruct.eventId,
  burstNumber)
  set the burst number field of the given ***eventId*** with the given value
- ***EVENT_ID_SET_NB_IN_BURST(***
  eventHeaderStruct.eventId,
  numberInBurst)
  set the number in burst field of the given ***eventId*** with the given value
- ***EVENT_ID_GET_NB_IN_RUN(eventHeaderStruct.eventId)***
  get the number in run field of the given ***eventId***
- ***EVENT_ID_GET_BURST_NB(eventHeaderStruct.eventId)***
  get the burst number field of the given ***eventId***
- ***EVENT_ID_GET_NB_IN_BURST(eventHeaderStruct.eventId)***
  get the number in burst field of the given ***eventId***

The following macros can be used for all encoding schemes:

- ***EQ_EVENT_ID(***
  eventHeaderStruct.eventIdA,
  eventHeaderStruct.eventIdB)
  TRUE if ***eventIdA*** is equal to ***eventIdB***
- ***LT_EVENT_ID(***
  eventHeaderStruct.eventIdA,
  eventHeaderStruct.eventIdB)
  TRUE if ***eventIdA*** is smaller (older) than ***eventIdB***
- ***GT_EVENT_ID(***
  eventHeaderStruct.eventIdA,
  eventHeaderStruct.eventIdB)
  TRUE if ***eventIdA*** is greater (more recent) than ***eventIdB***
- ***LE_EVENT_ID(***
  eventHeaderStruct.eventIdA,
  eventHeaderStruct.eventIdB)

TRUE if ***eventIdA*** is smaller (older) or equal to ***eventIdB***

- ***GE_EVENT_ID(***
  eventHeaderStruct.eventIdA,
  eventHeaderStruct.eventIdB)
  TRUE if ***eventIdA*** is greater (more recent) or equal to ***eventIdB***
- ***COPY_EVENT_ID(***
  eventHeaderStruct.eventIdFrom,
  eventHeaderStruct.eventIdTo)
  copy ***eventIdFrom*** into ***eventIdTo***
- ***ZERO_EVENT_ID(eventHeaderStruct.eventId)***
  clears the given ***eventId*** by setting all fields to zero
- ***ADD_EVENT_ID(***
  eventHeaderStruct.eventIdA,
  eventHeaderStruct.eventIdB)
  load ***eventIdA*** with the sum of ***eventIdA*** and ***eventIdB***: this macro should be used with a ***eventIdB*** all zero excepted one field (it makes little sense to use it with complex patterns, although the macro will still do the requested operation)
- ***SUB_EVENT_ID(***
  eventHeaderStruct.eventIdA,
  eventHeaderStruct.eventIdB)
  load ***eventIdA*** with the difference between ***eventIdA*** and ***eventIdB***: this macro should be used with a ***eventIdB*** all zero excepted one field (it makes little sense to use it with complex patterns, although the macro will still do the requested operation)

3.5.7 eventTriggerPattern

The ***eventTriggerPattern*** field contains the level 2 trigger pattern as published by the trigger system (referred as **L2Class[50 .. 1]**). Its size is given by the symbols ***EVENT_TRIGGER_PATTERN_BYTES*** (number of 8 bit entities) and ***EVENT_TRIGGER_PATTERN_WORDS*** (number of 32 bit entities).

The trigger pattern can be either validated or invalidated. In the first form, it is assumed to contain a valid pattern and can be used to activate trigger-based rules, such as event building rules or monitoring selection criteria. In the second form it cannot be used to activate trigger-based rules although its content can still be loaded according to requirements. In both cases an arbitrary number of trigger classes (from none to all of them) can be set in a trigger pattern.

The ***eventTriggerPattern*** can be used to store trigger IDs (here referenced as ***triggerId***) in the range
[***EVENT_TRIGGER_ID_MIN*** .. ***EVENT_TRIGGER_ID_MAX***] (currently [0 .. 49] which correspond to the number of trigger classes that can be used in ALICE). For each trigger ID we have one and only one trigger mask (a set of bits with one and only one bit set, where each bit corresponds to one trigger class) that can be used to create, handle and test trigger patterns (sets of bits with zero or more bits set, where each bit corresponds to one trigger class).

The following macros are available to handle trigger patterns:

- ***ZERO_TRIGGER_PATTERN(eventHeaderStruct.eventTriggerPattern)***
  clear and invalidate the given trigger pattern
- ***COPY_TRIGGER_PATTERN(***
  eventHeaderStruct.eventTriggerPatternFrom,
  eventHeaderStruct.eventTriggerPatternTo)
  copy the “from” pattern into the “to” pattern and, if the “from” pattern is validated, validate the “to” pattern
- ***SET_TRIGGER_IN_PATTERN(***
  eventHeaderStruct.eventTriggerPattern,
  triggerId)
  set the bit corresponding to the given ***triggerId*** in the given trigger pattern
- ***CLEAR_TRIGGER_IN_PATTERN(***
  eventHeaderStruct.eventTriggerPattern,
  triggerId)
  clear the bit corresponding to the given ***triggerId*** in the given trigger pattern
- ***FLIP_TRIGGER_IN_PATTERN(***
  eventHeaderStruct.eventTriggerPattern,
  triggerId)
  flip (xor) the status of the bit corresponding to the given ***triggerId*** in the given trigger pattern
- ***TEST_TRIGGER_IN_PATTERN(***
  eventHeaderStruct.eventTriggerPattern,
  triggerId)
  TRUE if the bit corresponding to the given ***triggerId*** is set in the given trigger pattern
- ***VALIDATE_TRIGGER_PATTERN(***
  eventHeaderStruct.eventTriggerPattern)
  validate the given trigger pattern
- ***INVALIDATE_TRIGGER_PATTERN(***
  eventHeaderStruct.eventTriggerPattern)
  invalidate the given trigger pattern
- ***TRIGGER_PATTERN_VALID(***
  eventHeaderStruct.eventTriggerPattern)
  TRUE if the given trigger pattern has been validated
- ***TRIGGER_PATTERN_OK(eventHeaderStruct.eventTriggerPattern)***
  TRUE if the given trigger pattern is syntactically correct

3.5.8 eventDetectorPattern

The ***eventDetectorPattern*** field contains information based upon the ***L2a message***, published by the ALICE trigger system, associated to the given event. For physics events it contains the detector pattern corresponding to the ***L2Cluster[6..1]*** field. For software triggers (calibration, detector software trigger and system software trigger events) it contains the ***L2Detector[24..1]*** field. The size of the ***eventDetectorPattern*** field is

given by the symbols ***EVENT_DETECTOR_PATTERN_BYTES*** (number of 8 bit entities) and ***EVENT_DETECTOR_PATTERN_WORDS*** (number of 32 bit entities).

The detector pattern can be either validated or invalidated. In the first form, it is assumed to contain a valid pattern and can be used to activate detectorId-based rules, such as event building rules. In the second form it cannot be used to activate detectorId-based rules although its content can still be loaded according to requirements. In both cases an arbitrary number of detectors (from none to the whole lot) can be set in a detector pattern.

The pattern is a set of bits, each corresponding to one and only one ***detectorId*** (one detector ID for each detector). Detector IDs belong to range [***EVENT_DETECTOR_ID_MIN*** .. ***EVENT_DETECTOR_ID_MAX***] (currently [0 .. 30]). The range [***EVENT_DETECTOR_ID_MIN*** .. ***EVENT_DETECTOR_HW_ID_MAX***] is reserved for HW detectors (to be specified in the **Common Data Header**) while the range (***EVENT_DETECTOR_HW_ID_MAX***. .. ***EVENT_DETECTOR_ID_MAX***] is reserved to SW detectors. A ***detectorPattern*** is a set of bits with one and only one bit for each ***detectorId***, bit that can be either one (**TRUE**) or zero (**FALSE**). A ***detectorMask*** is a ***detectorPattern*** with one and only one bit set.

The following macros are available to handle detector patterns:

- ***ZERO_DETECTOR_PATTERN(***
  eventHeaderStruct.eventDetectorPattern)
  clear and invalidate the given detector pattern
- ***COPY_DETECTOR_PATTERN(***
  eventHeaderStruct.eventDetectorPatternFrom,
  eventHeaderStruct.eventDetectorPatternTo)
  copy the “from” detector pattern into the “to” detector pattern and - if the “from” detector pattern is validated, validate the “to” detector pattern
- ***SET_DETECTOR_IN_PATTERN(***
  eventHeaderStruct.eventDetectorPattern,
  detectorId)
  set the bit corresponding to the given ***detectorId*** in the given detector pattern
- ***CLEAR_DETECTOR_IN_PATTERN(***
  eventHeaderStruct.eventDetectorPattern,
  detectorId)
  clear the bit corresponding to the given ***detectorId*** in the given detector pattern
- ***FLIP_DETECTOR_IN_PATTERN(***
  eventHeaderStruct.eventDetectorPattern,
  detectorId)
  flip (xor) the status of the bit corresponding to the given ***detectorId*** in the given detector pattern
- ***TEST_DETECTOR_IN_PATTERN(***
  eventHeaderStruct.eventDetectorPattern,
  detectorId
  TRUE if the bit corresponding to the given ***detectorId*** is set in the given detector pattern
- ***VALIDATE_DETECTOR_PATTERN(***
  eventHeaderStruct.eventDetectorPattern)

validate the given detector pattern

- ***INVALIDATE_DETECTOR_PATTERN(eventHeaderStruct.eventDetectorPattern)***
  invalidate the given detector pattern
- ***DETECTOR_PATTERN_VALID(eventHeaderStruct.eventDetectorPattern)***
  TRUE if the given detector pattern has been validated
- ***DETECTOR_PATTERN_OK(eventHeaderStruct.eventDetectorPattern)***
  TRUE if the given detector pattern is syntactically correct

The following compilation symbols are defined:

- ***EVENT_DETECTOR_ID_MIN*** set to 0
- ***EVENT_DETECTOR_ID_MAX*** set to 30

3.5.9 eventTypeAttribute

Every event has two sets of attributes available: the system attributes and the user attributes. The system attributes are common to all events and are usually set by the standard DATE software. The user attributes are specific to a data-acquisition system, to an LDC or to an equipment: their definition is left to the responsible for the DAQ system. All attributes can be used to select events for monitoring purposes.

The standard DATE symbols include three set of macros and symbols. One set is dedicated to system attributes. The second set is for user attributes. A third set manipulates all attributes at once: this can be useful for global operations - such as reset of a pattern - but should be used with care for other types of operations. The third set of macros sees the system attributes as an extension of the user attributes, as if they would physically extend it beyond its physical boundaries.

Every attribute is identified by a ***attributeId***, a unique number defining one of the allowed attributes. An attribute pattern - defined by the ***eventTypeAttribute*** data type - is a set of bits with zero or more bits (each corresponding to one and only one ***attributeId***) asserted. System attributes are defined by special DATE symbols while user attributes are, at the base, defined by their positional number (they can be re-defined as site-dependent symbols is the need arises).

The following symbols and macros are available:

- ***SYSTEM_ATTRIBUTES_BYTES*** / ***SYSTEM_ATTRIBUTES_WORDS***
  number of bytes (8 bits) and words (32 bits) allocated to system attributes
- ***USER_ATTRIBUTES_BYTES*** / ***USER_ATTRIBUTES_WORDS***
  number of bytes (8 bits) and words (32 bits) allocated to user attributes
- ***ALL_ATTRIBUTES_BYTES*** / ***ALL_ATTRIBUTES_WORDS***
  number of bytes (8 bits) and words (32 bits) allocated to all attributes (system and user)
- ***RESET_ATTRIBUTES(eventHeaderStruct.eventTypeAttribute)***
  reset (clear) all attributes (system and user) of the given attribute pattern

- ***SET_ANY_ATTRIBUTE(***
  eventHeaderStruct.eventTypeAttribute,
  attributeId)
  set the bit corresponding to the given ***attributeId*** (system or user) in the given attribute pattern
- ***CLEAR_ANY_ATTRIBUTE(***
  eventHeaderStruct.eventTypeAttribute, attributeId)
  clear the bit corresponding to the given ***attributeId*** (system or user) in the given attribute attribute pattern
- ***FLIP_ANY_ATTRIBUTE(***
  eventHeaderStruct.eventTypeAttribute, attributeId)
  flip (xor) the bit corresponding to the given ***attributeId*** (system or user) of the given attribute pattern
- ***TEST_ANY_ATTRIBUTE(***
  eventHeaderStruct.eventTypeAttribute,
  attributeId)
  return ***TRUE*** is the bit corresponding to the given ***attributeId*** (system or user) of the given attribute pattern is set
- ***COPY_ALL_ATTRIBUTES(***
  eventHeaderStruct.eventTypeAttributeFrom,
  eventHeaderStruct.eventTypeAttributeTo)
  copy the “from” attribute pattern to the “to” attribute pattern
- ***RESET_SYSTEM_ATTRIBUTES(***
  eventHeaderStruct.eventTypeAttribute)
  reset (clear) the system attributes of the given attribute pattern, leaving the user attributes unmodified
- ***SET_SYSTEM_ATTRIBUTE(***
  eventHeaderStruct.eventTypeAttribute,
  attributeId)
  set the bit corresponding to the given ***attributeId*** (system) in the given attribute pattern
- ***CLEAR_SYSTEM_ATTRIBUTE(***
  eventHeaderStruct.eventTypeAttribute,
  attributeId)
  clear the bit corresponding to the given ***attributeId*** (system) in the given attribute pattern
- ***FLIP_SYSTEM_ATTRIBUTE(***
  eventHeaderStruct.eventTypeAttribute,
  attributeId)
  flip (xor) the bit corresponding to the given ***attributeId*** (system) of the given attribute pattern
- ***TEST_SYSTEM_ATTRIBUTE(***
  eventHeaderStruct.eventTypeAttribute,
  attributeId)
  return ***TRUE*** if the bit corresponding to the given ***attributeId*** (system) is set in the given attribute pattern
- ***COPY_SYSTEM_ATTRIBUTES(***
  eventHeaderStruct.eventTypeAttributeFrom,
  eventHeaderStruct.eventTypeAttributeTo)

copy the “from” system attributes to the “to” system attributes leaving the user attributes unmodified

- ***SYSTEM_ATTRIBUTES_OK(eventHeaderStruct.eventTypeAttribute)***
  check the validity of the system attributes of the given attribute pattern
- ***RESET_USER_ATTRIBUTES(eventHeaderStruct.eventTypeAttribute)***
  reset (clear) the user attributes of the given attribute pattern, leaving the system attributes untouched

- ***SET_USER_ATTRIBUTE(***
  eventHeaderStruct.eventTypeAttribute,
  attributeId)
  set the bit corresponding to the given ***attributeId*** (user) in the given attribute pattern
- ***CLEAR_USER_ATTRIBUTE(***
  eventHeaderStruct.eventTypeAttribute,
  attributeId)
  clear the bit corresponding to the given ***attributeId*** (user) in the given attribute pattern
- ***FLIP_USER_ATTRIBUTE(***
  eventHeaderStruct.eventTypeAttribute,
  attributeId)
  flip (xor) the bit corresponding to the given ***attributeId*** (user) in the given attribute pattern
- ***TEST_USER_ATTRIBUTE(***
  eventHeaderStruct.eventTypeAttribute,
  attributeId)
  return ***TRUE*** if the bit corresponding to the given ***attributeId*** (user) is set in the given attribute pattern
- ***COPY_USER_ATTRIBUTES(***
  eventHeaderStruct.eventTypeAttributeFrom,
  eventHeaderStruct.eventTypeAttributeTo)
  copy the user field of the “from” attribute pattern into the user field of the “to” attribute pattern leaving the system attributes unmodified

The following system attributes are currently defined at the DATE level:

- ***ATTR_P_START***
  phase start, used for ***START_OF_RUN*** and ***END_OF_RUN*** events
- ***ATTR_P_END***
  phase end, used for ***START_OF_RUN*** and ***END_OF_RUN*** events
- ***ATTR_START_OF_RUN_START***
  synonym for ***ATTR_P_START***
- ***ATTR_START_OF_RUN_END***
  synonym for ***ATTR_P_END***
- ***ATTR_END_OF_RUN_START***
  synonym for ***ATTR_P_START***
- ***ATTR_END_OF_RUN_END***
  synonym for ***ATTR_P_END***

- ***ATTR_EVENT_SWAPPED***
  set when the base header of the given event has been swapped (different endianness). The header extension and payload of the events have not been swapped
- ***ATTR_EVENT_PAGED***
  set for paged event, unset for streamlined events
- ***ATTR_SUPER_EVENT***
  set for events created on GDCs
- ***ATTR_ORBIT_BC***
  set when the eventId follows the ***COLLIDER*** mode encoding, not set for ***FIXED TARGET*** mode encoding
- ***ATTR_KEEP_PAGES***
  set when the data pages (carrying the payload) of the event are not to be disposed after the event is recorded
- ***ATTR_HLT_DECISION***
  set when the payload of the event starts with an HLT Decision record
- ***ATTR_BY_DETECTOR_EVENT***
  set when the event has been created via a "monitoring by detector" scheme
- ***ATTR_EVENT_DATA_TRUNCATED***
  set when the payload of the event has been truncated due to insufficient buffer space
- ***ATTR_EVENT_ERROR***
  set if the base header of the given event is syntactically incorrect

3.5.10 eventLdcId and eventGdcId

The ***eventLdcId*** field contains the ID (according to the DATE role database) of the LDC source of the event. The field is loaded with ***VOID_ID*** if the event has not been created on a LDC.

The ***eventGdcId*** contains the ID (according to the DATE role database) of the GDC source of the event or of the GDC destination of the event.

The symbols ***HOST_ID_MIN*** and ***HOST_ID_MAX*** are available, as well as the symbol ***VOID_ID***. No LDC or GDC can be assigned the ID ***VOID_ID***.

3.5.11 eventTimestampSec and eventTimestampUsec

The ***eventTimestampSec*** and ***eventTimeStampUsec*** fields contain the host system time taken the moment the event is created (trigger arrived on the LDC, first sub-event received on the GDC, event ready for monitoring by detector) split in two 32-bit parts: seconds (***eventTimestampSec***) and milliseconds (***eventTimeStampUsec***). The ***eventTimestampSec*** field may eventually have been truncated to 32 bit (if the size of the "`time_t`" unit on the generating host is > 32 bit) and must be assigned to a native `time_t` entity prior of using it (see below). For more system-specific details concerning these two fields, check the definition of the system call "`gettimeofday`" and of the system structure "`timeval`". These

fields can also be used with the standard Unix system library for printing and for handling (see the definition of the system call "`time`").

For portability issues across different platforms, this field must be copied into a variable of type "`time_t`" - as defined by the `<time.h>` system include file - before using it. Failure to do so may give unexpected results and may terminate the calling process. This procedure takes care of issues such as sizing and signess of the two fields.

The content of these fields may be inaccurate due to clock drifts, system clock adjustment, and latencies (hardware and software), both within the same machine and across different machines. If a more accurate timestamp is required, we recommend to use the LHC clock instead (as available in the ***eventId*** field).

3.6 The super event format

The output of a DATE system is a stream of events. These can be created either by a LDC or by a GDC. In the second case, the events are marked as **`super events`**. Super events have the same structure as events or sub-events: their payload however is guaranteed to contain a series of one or more sub-events.

The data format structure described before applies to sub-events and to super events. Each event will include a header and a data block. In the cases of a super event assembled by the ***eventBuilder***, the data block is itself subdivided into sub-events. Each sub-event will include a header and a data block. The ***eventBuilder*** assembles the sub-events pertaining to the same event and prepends one header relative to the complete event. An example of this representation, with two LDCs (IDs 5 and 7) merging on one GDC (ID 1) is shown in Figure 3.7.

The sub-event refers here to the data read-out by one LDC and assembled later on by one GDC. The super event refers here to the full set of data collected by a DAQ system for an event uniquely identified by a ***eventType-eventId*** pair.

Events can be decoded using the same algorithm. Super events, however, can have the same algorithm applied to their payload, where the payloads split into blocks of one sub-event each.

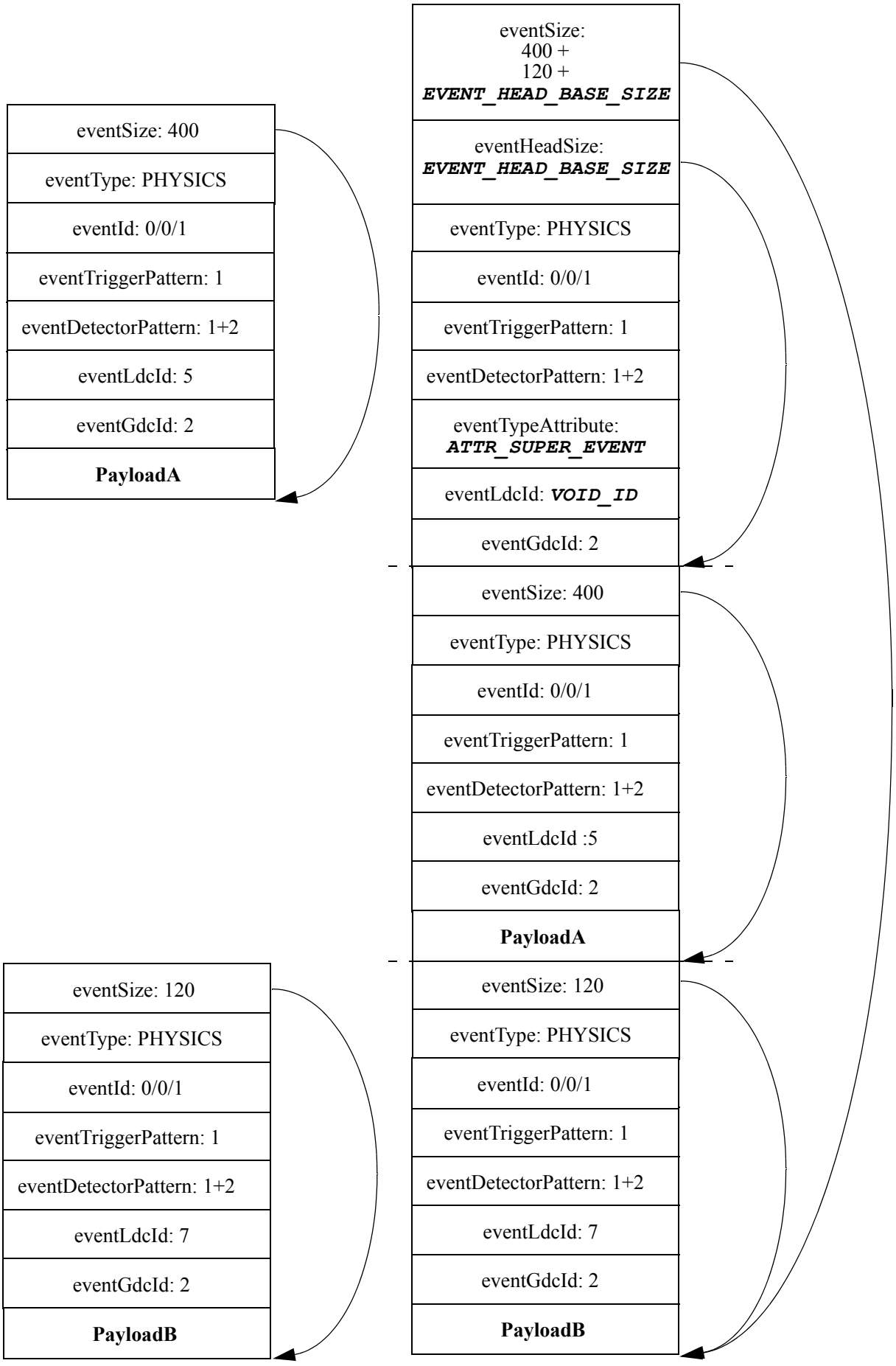

Figure 3.7 The full event format

3.7 The complete file format

The Table 3.2 shows the sequence of records constituting a complete DATE raw data file.

Table 3.2 The successive list of records in a data file generated by DATE

Event type	Event attribute	Comments
START_OF_RUN	***ATTR_P_START***	Unique
START_OF_RUN_FILES		Zero or more records
START_OF_RUN	***ATTR_P_END***	Unique
START_OF_BURST		Optional, unique per each burst
START_OF_DATA		Zero or one record
PHYSICS_EVENT		Zero or more records
CALIBRATION_EVENT		Zero or more records
SYSTEM_SOFTWARE_ TRIGGER_EVENT		Zero or more records
DETECTOR_SOFTWARE_ TRIGGER_EVENT		Zero or more records
END_OF_DATA		Zero or one record
END_OF_BURST		Optional, unique per each burst
END_OF_RUN	***ATTR_P_START***	Unique
END_OF_RUN_FILES		Zero or more records
END_OF_RUN	***ATTR_P_END***	Unique

Of the above events, only the ***START_OF_RUN**** [1] and ***END_OF_RUN**** [2] are to be found in all runs. It is possible to have empty runs (without ***START_OF_DATA, END_OF_DATA, PHYSICS***, ***CALIBRATION***, ***SYSTEM_SOFTWARE_TRIGGER*** or***DETECTOR_SOFTWARE_TRIGGER*** events). ***START_OF_BURST*** and ***END_OF_BURST*** events shall be used only when burst-like beam structure is available (typical of fixed target installations).

1. ***START_OF_RUN*** with ***ATTR_P_START*** and ***ATTR_P_END*** and ***START_OF_RUN_FILES***
2. ***END_OF_RUN*** with ***ATTR_P_START*** and ***ATTR_P_END*** and ***END_OF_RUN_FILES***

3.8 Decoding and monitoring on different platforms

Information of all kind is exchanged between computers' memories. The way these computers order their memory may differ. Most of the times, they will follow either the Little-Endian scheme or the Big-Endian scheme [13]. Little-Endian (LE) computers assign bit 0 to the Least Significant Byte (LSB) of their word and the top bit to the Most Significant Byte (MSB) of their word. Big-Endian (BE) computers do just the opposite: bit 0 is in the MSB and the top bit is in the LSB.

It is evident that exchanging data between LE and BE computer (either via network channels or through files - shared or on permanent media) can create several problems when memory ordering becomes important. Due to efficiency and practical constraints, data acquisition systems based on DATE will have to handle transfer of data between LE and BE computers on their own and on a case-by-case basis.

The conversion is necessary every time an event is decoded on a platform of different endianness from the platform where the event was created. A short, non-exhaustive list of platforms that could take part in such a process is given in Table 3.3.

Table 3.3 Commonly used platforms and their endianness

Platform	Endianness type
Intel (x86, Pentium)	Little-Endian
COMPAQ (HP) ALPHA	Little-Endian
Motorola PPC	Big-Endian
Sun SPARC	Big-Endian
HP PA-RISC	Big-Endian
SGI IRIX	Big-Endian
IBM RS6000	Big-Endian

Programs decoding raw events may have to detect the need for swapping. If portability is not an issue (programs will always run on the same type of platform and data will always be generated on the same type of platform) a rigid swapping policy - if needed - can be systematically applied (always swap events without checking). Programs that may run on different platforms (e.g. generic monitoring programs or roaming programs who may migrate from computer to computer) will have to check on the fly for the appropriate swapping policy.

When data is transferred via a DATE library (***monitoring*** or ***eventBuilder***), a check is always performed on the header of all the events. If the need for swapping is detected, the DATE library adjusts the event header and sets the ***ATTR_EVENT_SWAPPED*** bit of the type field accordingly. Please note that only the event header is "adjusted": the data portion of the event remains in its original

status. Monitoring programs can use the ***ATTR_EVENT_SWAPPED*** bit to trigger their internal swapping algorithm and correct the data portion of the event.

When data is fetched directly from a DATE stream (pipe, file or socket), then programs should check the magic field of the event header. When this field is equal to ***EVENT_MAGIC_NUMBER*** no swapping is needed. When this field is set to ***EVENT_MAGIC_NUMBER_SWAPPED*** swapping of the event header as well as of the event data will be needed.

Examples of these two checks can be found in Listing 3.1.

Listing 3.1 Detecting swapping of the event data

```
 1:  /*** Examples of detection of different endianness data ***/
 2:  struct eventHeaderStruct header;
 3:
 4:  /* Load the header structure (not shown) */
 5:
 6:  if (TEST_SYSTEM_ATTRIBUTE(header.eventTypeAttribute,
                                  ATTR_EVENT_SWAPPED))
 7:    printf( "Swapping needed (SWAPPED bit)\n" );
 8:
 9:  if (header.eventMagic == EVENT_MAGIC_NUMBER_SWAPPED)
10:    printf("Swapping needed (MAGIC_SWAPPED)\n");
```

Special situations may arise when LDCs and GDCs are of different endianness. In this case, the ***eventBuilder*** process detects the need for swapping and adjusts the sub-event header (setting the ***ATTR_EVENT_SWAPPED*** bit of the sub-event header type field). As not all the LDCs may be of the same type, a different swapping policy may have to be applied on a sub-event by sub-event basis.

How to cope with the swapping of data depends on the content of the event itself. Due to the way events are transferred (via network channels, files or on permanent data storage), we must apply different treatment for 8 bit entities (characters, **RS-232**, small I/O channels), 16 bit entities (such as **CAMAC** data), 32 bit entities (DATE event headers, **VMEbus**, **PCI**, wide I/O channels) and 64 bit entities (wide **PCI**, very wide I/O channels). In our experience, 8 bit entities do not need any swapping; bigger data entities (16, 32, 64 bit) need some sort of conversion that depends on the data internal structure.

To facilitate the swapping process, the DATE monitoring library provides the `monitorSetSwap` entry. This entry will apply (if needed) the given swap policy to entire events, assuming they contain only and always 8, 16 or 32 bit entities. Events with non-uniform data content must be swapped with ad-hoc algorithms. We suggest to write a small data file with a couple of good examples of events and debug the decoding/monitoring swapping scheme using this file, then move to the production platform. We also strongly recommend to always check for the need of the swapping of data (using one of - or both - the methods illustrated in Listing 3.1), as the same stream may take different routes and therefore undergo to the swapping process more than once.

3.9 The Common Data Header

All events sent over the ALICE DDL must be prepended by a Common Data Header as defined in [2] and refined in [15].

The main component of the definitions dedicated to the Common Data Header is the structure ***commonDataHeaderStruct***, defined in Table 3.4.

Table 3.4 Common data header structure

Name	Type	Content
cdhBlockLength	unsigned:32	Length of the block
cdhVersion	unsigned:8	Version ID of the CDH
cdhL1TriggerMessage	unsigned:10	Level 1 trigger message
cdhEventId1	unsigned:12	Bunch-crossing field of the event ID
cdhEventId2	unsigned:24	Orbit-number field of the event ID
cdhMiniEventId	unsigned:12	BC counter at the moment of the L1 trigger signal
cdhBlockAttributes	unsigned:8	Attributes of the block
cdhParticipatingSubDetectors	unsigned:24	Pattern of sub-detectors
cdhStatusAndErrorBits	unsigned:16	Status and error bits
cdhTriggerClassesHigh	unsigned:18	Trigger classes (high 18 bits)
cdhTriggerClassesLow	unsigned:32	Trigger classes (low 32 bits)
cdhRoiHigh	unsigned:32	Region Of Interest (high 32 bits)
cdhRoiLow	unsigned:4	Region Of Interest (low 4 bits)

All the definitions given in the table are relative for Little-Endian architectures. All fields can be directly handled by 32- and 64-bit CPUs, including handling of bit patterns and bit masks. All symbols ending with a ***_BIT*** suffix refer to a bit number (LSB:0).

The size of the common data header structure is defined in the compilation constant ***CDH_SIZE***.

The structure contains several fields that must be set to zero. Those fields are not specified in the above table but can be found in the definitions given by the DATE ***event.h*** include file. For compatibility with future versions of the Common Data Header, we recommend - for newly allocated structures - to zero the whole structure and then set / update the fields that have to be set / updated. Using this method, the Must Be Zero fields and the not handled fields will always be zeroed, independently from their location and from their definition.

3.9.1 Common Data Header version

The version of the Common Data Header as defined during the compilation of the handling module is given in the constant ***CDH_VERSION***. This constant is an incremental number and can be used for arithmetic comparisons. New versions of the Common Data Header will be marked with newer version IDs.

Code setting and / or using the common data header should always check the version ID found in a common data header vs. the version ID defined during the compilation stage. When a mismatch is found, this must trigger either an error condition or (if possible) a translation between the two versions.

3.9.2 Status and Error bits

The status and error bits - given in the ***cdhStatusAndErrorBits*** field of the ***commonDataHeaderStruct*** structure - can - for the Common Data Header version 1 - carry the information as given in Table 3.5.

Table 3.5 Common Data Header Status and Error bits

Name	Status/ Error	Content
CDH_TRIGGER_OVERLAP_ERROR_BIT	Error	L1 received while processing another L1
CDH_TRIGGER_MISSING_ERROR_BIT	Error	L1 received when no L0 has been received
CDH_CONTROL_PARITY_ERROR_BIT	Error	Control parity error (instruction and / or address)
CDH_DATA_PARITY_ERROR_BIT	Error	Data parity error
CDH_FEE_ERROR_BIT	Error	Front-end electronics error
CDH_TRIGGER_INFORMATION_UNAVAILABLE_BIT	Status	Trigger information unavailable
CDH_HLT_DECISION_BIT	Status	HLT decision available in payload
CDH_HLT_PAYLOAD_BIT	Status	HLT payload follows
CDH_DDG_PAYLOAD_BIT	Status	DDG payload follows

The assertion of the ***CDH_HLT_DECISION_BIT*** implies the assertion of the ***CDH_HLT_PAYLOAD_BIT*** bit. Events whose Common Data Header ***CDH_HLT_DECISION_BIT*** status bit is set while the ***CDH_HLT_PAYLOAD_BIT*** is not set are considered wrong and must be rejected.

3.10 The equipment header

An LDC can include one or more equipments. Each equipment is associated to one logical input channel, usually paired with one physical channel. All DATE events

must describe the equipments contributing to the payload. This is done using the ***equipment header*** structure.

The structure of the ***equipmentHeaderStruct*** structure is described in Table 3.6.

Table 3.6 Equipment header structure

Name	Type	Content
equipmentSize	***equipmentSizeType***	total size of the payload
equipmentType	***equipmentTypeType***	type of the equipment
equipmentId	***equipmentIdType***	ID of the equipment
equipmentTypeAttribute	***equipmentTypeAttributeType***	attributes of the payload
equipmentBasicElementSize	***equipmentBasicElementSize-Type***	size of the basic element for the equipment

We will now review the individual fields of the ***equipment header*** structure.

3.10.1 equipmentSize

This field contains the combined size of the payload created by the equipment. This size does not include the equipment header whose size is fixed. The value should be aligned to a 32 bits boundary.

3.10.2 equipmentType/equipmentId

The type and ID of the equipment as defined in the DATE site configuration.

3.10.3 equipmentTypeAttribute

The type attributes associated to the equipment. The same rules, symbols and macros as for the ***eventTypeAttribute*** are applicable.

3.10.4 equipmentBasicElementSize

The size of the basic element accepted by the equipment itself. This field is mainly used to adjust the content of the payload when crossing endianness boundaries.

3.11 Paged events and DATE vectors

DATE paged events must provide two main capabilities:

a. support for multi-page payloads with multiple data pools

b. support for efficient exchange of events between different processes

To achieve these capabilities the ***eventVectorStruct*** entity has been defined as by Table 3.7.

Table 3.7 Event vector structure

Name	Type	Content
eventVectorBankId	***eventVectorBankIdType***	ID of the bank supporting the pointed entity
eventVectorPointsToVector	***unsigned***	type of the pointed entity
eventVectorSize	***eventVectorSizeType***	size of the pointed entity
eventVectorStartOffset	***eventVectorOffsetType***	start offset of the pointed entity

The eventVectorStruct is used to point to an entity, vector or payload. It fully describes the pointed entity. A ***NULL*** vector has ***eventVectorSize*** set to zero.

Entities are pointed by a bankId-startOffset pair: the bank ID is a unique identifier defined by the DATE ***database***/***banksManager*** packages according to the run-time configuration of the DATE site.

When the pointed entity is a vector, the ***eventVectorPointsToVector*** field of the pointer is ***TRUE*** and the ***eventVectorSize*** contains the number of entries of the pointed vector.

When the pointed entity is a data page, the ***eventVectorPointsToVector*** of the pointer is ***FALSE*** and the ***eventVectorSize*** contains the size of the data page.

An example of use of the above structure is given inFigure 3.8 where an event with 6 payloads spreading over two banks is described. The event is made of PayloadA through PayloadF for a total of 1652 bytes.

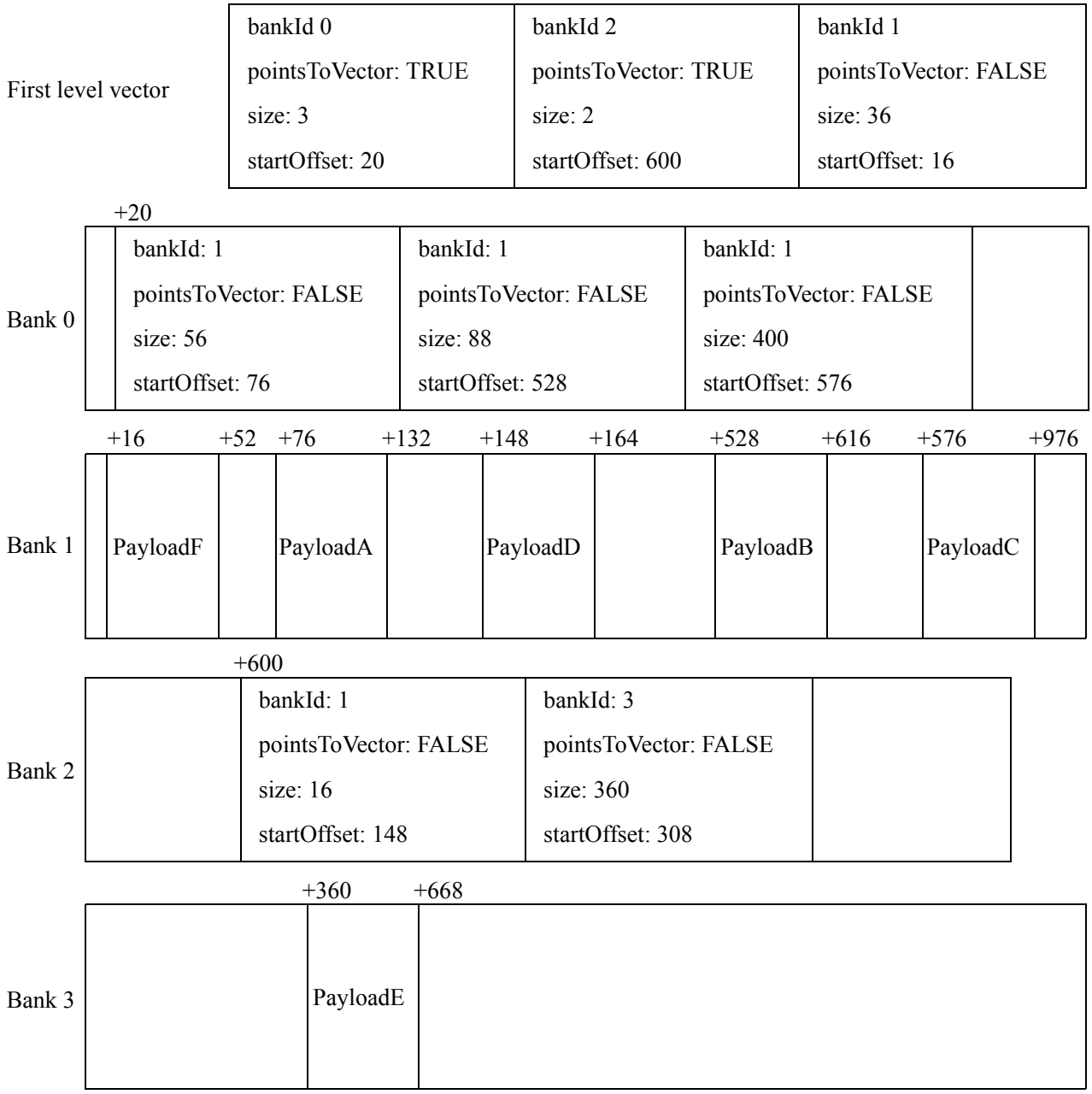

Figure 3.8 Example of use of DATE event vectors

To complete the definition of paged event we must define the ***vectorPayloadDescriptorStructure***. This structure defines all components of paged events following the base event header. Its format described in Table 3.8.

Table 3.8 Payload descriptor structure

Name	Type	Content
eventNumEquipments	***eventNumEquipmentsType***	Number of equipments contributing to the payload
eventExtensionVector	Single entry ***eventVectorStruct***	Pointer to the header extension

Including the above definitions, a complete paged event looks as follows:

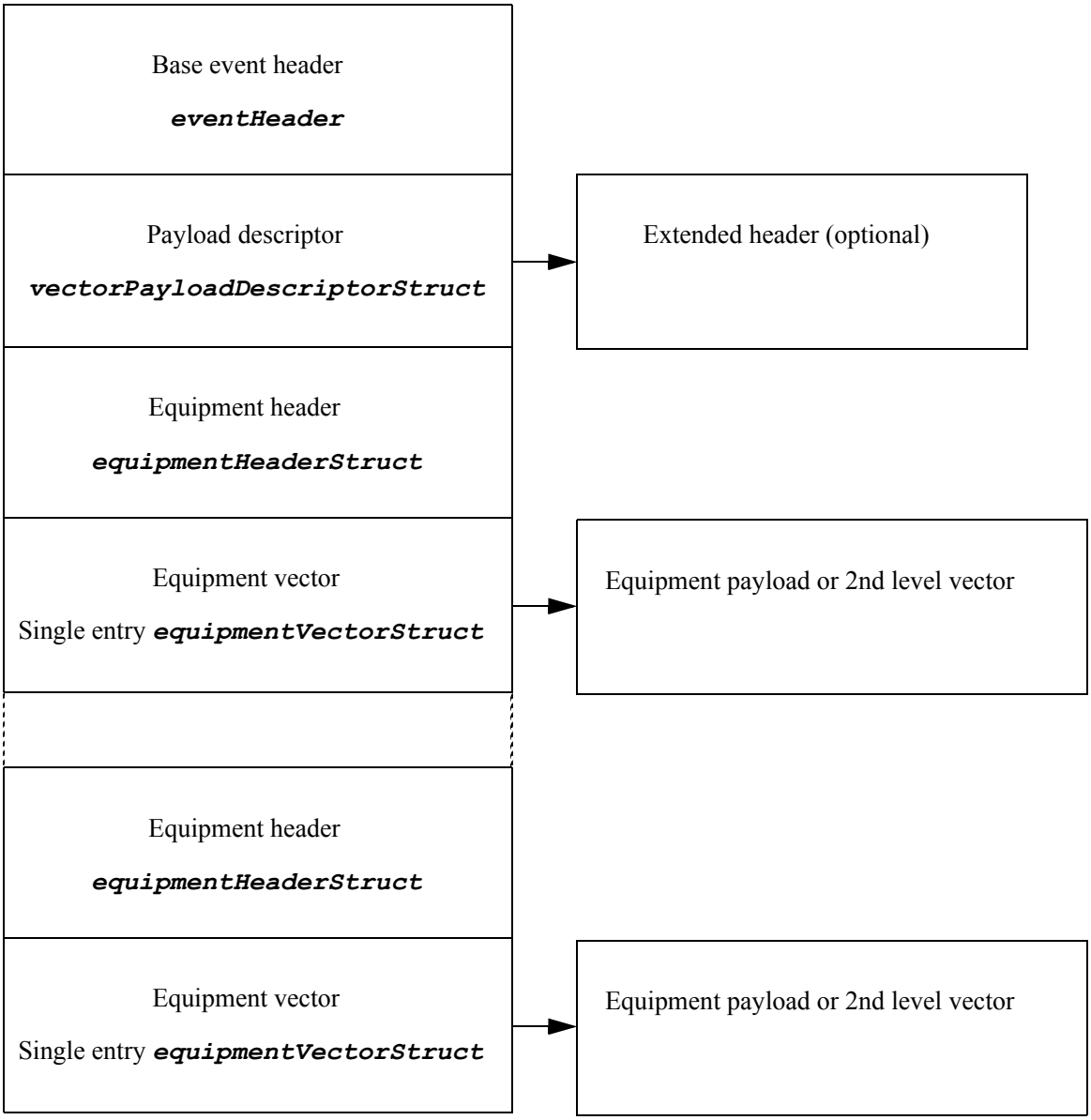

Figure 3.9 Example of complete paged event

3.12 Data pools

A data pool is a contiguous block of memory reserved to a particular function, e.g. the data pages available to the ***readout*** process. Each pool ***must*** be used exclusively for one function. If paged events mode is adopted, separate pools must be allocated for first level vectors, for second level vectors and for data pages. DATE systems can (and usually do) have multiple data pools.

4 Configuration databases

All actors belonging to a DATE system need access to configuration parameters. The DATE databases package, described in this chapter, provides the relevant interfaces to the static DATE configuration information.

4.1 Overview

Every DATE system can be fully defined by several pieces of information, consisting of a static part (described in a MySQL database and editable with ***editDb***, see Section 4.5) and a dynamic part (for run-specific configuration parameters, accessible via the ***runControl Human Interface***, see Section 14.5). Static information is mostly hardware related and valid across many consecutive runs: it includes definitions for available hosts (LDCs, GDCs, EDMs, etc.), readout links, detectors and triggers setup, and DATE components parameters. Based on these static definitions, the operator can then choose a specific dynamic configuration (set of hosts and their run-time parameters) for a given run.

The DATE database package (***dateDb***) provides an access layer to the static configuration, whereas the dynamic part is handled internally by the relevant DATE packages.

4.2 Information schema

The DATE static configuration is stored in a MySQL database. A graphical editor is provided to enter data.

To create the required database structure in MySQL, in a database named ***DATE_CONFIG*** by default, you need to define the database access parameters (see Section 2.1 and Section 4.4.4), execute the DATE setup procedure, and then type:

```
> ${DATE_DB_BIN}/createtables
```

This utility creates the tables structure. The existing configuration stored in MySQL is destroyed. It is recommended to rather use the `newDateSite.sh` which creates a working ***DATE_SITE*** directory and the corresponding ***DATE_CONFIG*** database from scratch, populating it with a minimal running set of local roles. It is then easy to augment the configuration with more roles.

For convenience, the configuration database can be backed up using the command `${DATE_DB_DIR}/dbBackup.sh`. It creates a SQL dump of the full content, which is handy in case one needs to recover from a hardware failure or a wrong operation on the data. It can easily be reloaded in an empty database using the `source filename.sql` syntax from the mysql client command line.

Figure 4.1 describes the current database structure and the relations between the main tables. Details about the semantics of each table are given in the following sections.

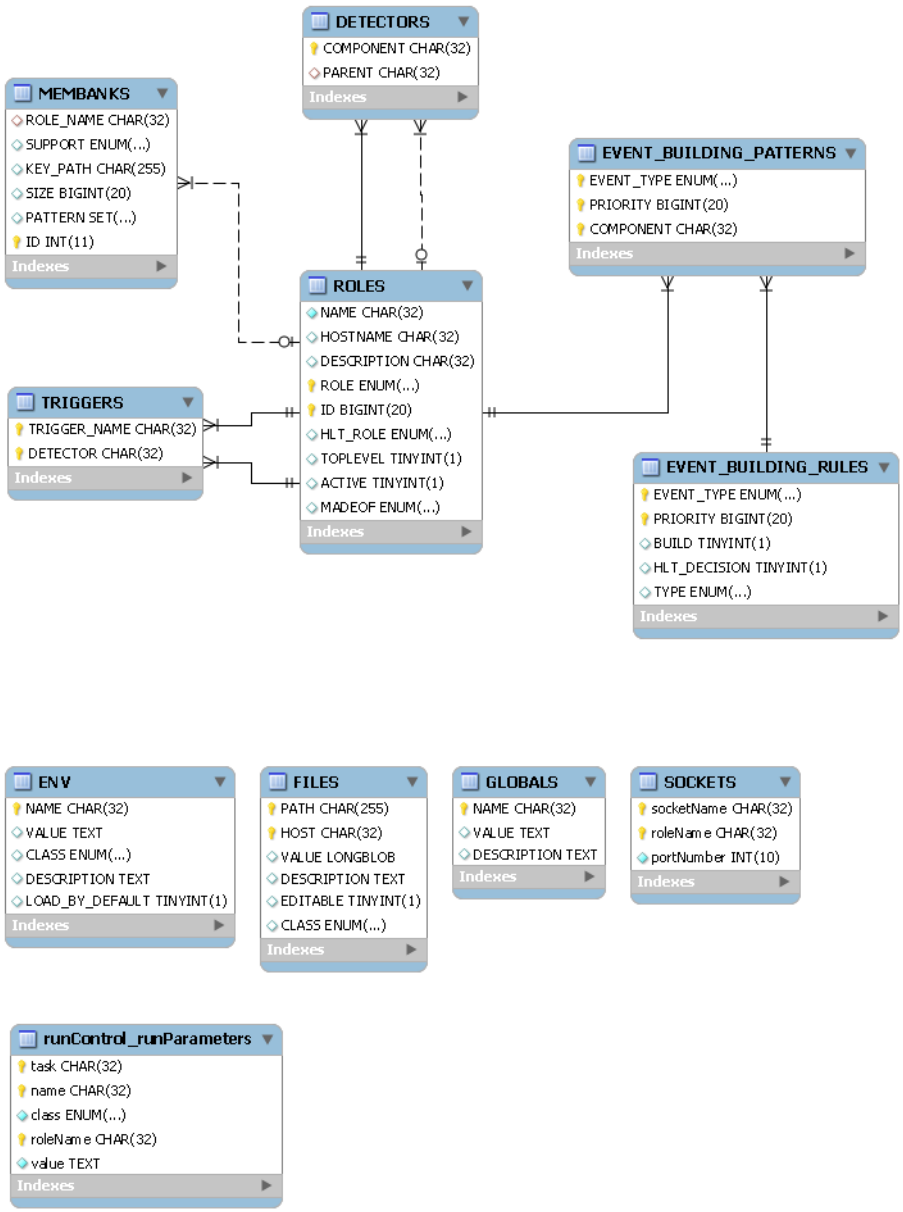

Figure 4.1 DATE configuration database structure - main tables.

4.3 The static databases

The DATE static configuration items are grouped in different families, historically named ‘databases’. Information was originally stored in flat ASCII files, and it was moved to a MySQL database storage starting from DATE V5. To follow the historical design conventions (and the vocabulary still in use in the older part of the DATE source code), what we describe below as ‘databases’ are the original categories of static parameters used to describe a DATE setup, despite the fact that all of them are now stored in the same and unique ***DATE_CONFIG*** database hosted on a MySQL server, mapping the information in the same way it was before in flat files. Newer categories of configurable items are described in Section 4.4.

The static databases include:

- the ***roles*** database: it contains the definitions of all the logical entities part of a given ***DATE_SITE***: LDCs, GDCs, EDM hosts, detectors and trigger masks;
- the ***triggers*** database: it defines the detectors involved in each trigger mask;
- the ***detectors*** database: it defines the composition of each detector and / or sub-detector in terms of sub-detectors and / or LDCs;
- the ***event building control*** database: it defines the event-building strategies to be applied by the ***eventBuilder*** process;
- the ***banks*** database: it defines the memory banks to be provided on the hosts defined in the ***roles*** database.

The information stored in each database is, in principle, static, i.e. it evolves slowly according to hardware or experimental conditions changes.

The static databases always describe a superset of the actual run-time configuration. An entity must be defined in the appropriate database to be able to participate in a run. The actual list of run-time actors is selected from these databases.

The current content of the static databases can be retrieved using the command `${DATE_DB_BIN}/dumpDbs`. The output of this utility looks as shown in Section 4.6. This tool also verifies the consistency of the database information (in particular, the references between tables) and may be useful for debugging purposes.

4.3.1 Terminology and assumptions

The following terms are used throughout this document:

- ***Role***: role of the entity (LDC, GDC, EDM, etc.).
- ***Name***: unique name of the entity defined by the given record. This name must be unique across all roles of the database. It is not case sensitive.
- ***ID***: identifier associated to the entity. Defined in the ***roles*** database, it can be associated to the corresponding bit of a ***bit mask*** or of a ***bit pattern***. Each ***ID*** is unique within its ***role***. Entities of different ***roles*** may share the same ***ID***. An ***ID*** can assume any value between its associated ***min ID*** (included) and its associated ***max ID*** (included). Not all the values in this range need to correspond to an entity: it is possible to have ***ID***s with no associated entity.
- ***Bit mask***: a set of bits with ***one and only one*** bit set: this bit corresponds to a given ***ID*** and it can be tested using the `DB_TEST_BIT` macro. A ***bit mask*** can be described by the same type and size of storage as the corresponding ***bit pattern***.
- ***Bit pattern***: a set of bits with any number of bits (including none) set. Each bit of a ***bit pattern*** correspond to a given ***ID*** and can be tested using the `DB_TEST_BIT` macro. A ***bit pattern*** can be considered like the combination of zero or more ***bit masks*** (of equal semantic) and it is described by the same type and size of storage as the equivalent ***bit mask***.
- ***Min ID*** and ***Max ID***: minimum and maximum value assumed by ***ID***s associated to a given role. For any given role, an entity exists with ID equal to

min ID and ***max ID*** and no entities exist with ID smaller than ***min ID*** or greater than ***max ID***. The ***max ID*** definition is dynamic and is - in principle - unlimited. However, three implicit limitations are given by:

1. the architecture where the code is executed (see the constant ***INT_MAX*** defined in ***limits.h***);
2. the corresponding (if any) entity part of the event header (***eventTriggerPattern, eventDetectorPattern, eventLdcId, eventGdcId***);
3. for all hosts, the ***HOST_ID_MIN*** and ***HOST_ID_MAX*** definitions given in ***${DATE_COMMON_DEFS}/event.h*** (currently equal to 0 and 511).

- ***maskElementType***: the basic type used to describe a ***bit mask*** or a ***bit pattern***.

The DATE event header imposes the following rules on the IDs ranges:

- ***trigger patterns*** must correspond to ID values in the [***EVENT_TRIGGER_ID_MIN***..***EVENT_TRIGGER_ID_MAX***] range;
- ***detector patterns*** must correspond to ID values in the [***EVENT_DETECTOR_ID_MIN..EVENT_DETECTOR_ID_MAX***] range;
- ***LDC ID***s and ***GDC ID***s must be in the [***HOST_ID_MIN..HOST_ID_MAX***] range.

The corresponding constants are defined in ***${DATE_COMMON_DEFS}/event.h***

Each DATE site may define its IDs within the limits imposed by the DATE event header and in respect of the rules described above. IDs are usually assigned automatically by ***editDb***.

4.3.2 The roles database

The ***roles*** database is used to define all entities part of a DATE system. The definition is given using a unique ***role-ID*** scheme, where ***role*** can be one of ***LDC, GDC, EDM, TRIGGER_MASK, TRIGGER_HOST, DETECTOR, SUBDETECTOR, DDG, FILTER, MON*** and ***ID*** is a positive integer (corresponding to a bit in all the relative ***masks*** and ***patterns***). Once the definition is complete, each component can be uniquely identified either by its ***role*** plus ***ID*** or by the presence of the associated bit in a related ***mask*** or ***pattern***.

The ***DDG***,***FILTER*** and ***MON*** roles are used by ***runControl*** to start some extra processes at run-time. See Chapter 14 for more information on these roles.

All the records of this database include a symbolic name, the associated ID and a textual description. Other optional or role-specific parameters are available.

All the other DATE databases use the definitions given in the ***roles*** database to reference DATE components by name rather than by absolute value. Therefore, to fully decode a record belonging to another database, a scan of the ***roles*** database is implicitly required. This is also needed to size the variable-size entities, such as ***bit masks*** and ***bit patterns***.

Entities that should be directly selectable from the ***runControl Human Interface*** can be given a ***topLevel*** attribute. When this attribute is set to "*Y*"

the entity is under direct control of the DAQ operator. On the other hand, when the attribute is set to "*N*" (or not given at all) the operator has no direct control and can only indirectly act on the given entity. Let's take an example of a DAQ system made of two detectors D1 and D2 and six LDCs L1 to L6. D1 is made of L1, L2, and L3, D2 is made of L4 and L5; L6 doesn't belong to any detector. A possible definition of this DAQ system gives to D1, D2 and L6 the attribute ***topLevel*** to "*Y*" and to the L1, L2, L3, L4, and L5 the value "*N*".

Entities that correspond to physical hosts (e.g. LDCs, GDCs, EDMs and TRIGGER_HOSTs) may be given an optional TCP/IP hostname. This will associate the TCP/IP host to the appropriate DATE entity. We can therefore implement "virtual hosts" (e.g. ***detectorOneLdcOne***) and associate them to the actual machine via the database. We can also have machines with multiple roles, e.g. ***GDC*** and ***EDM*** : same TCP/IP hostname and different role names. One can even have several LDC roles on the same machine if resources and readout equipment allows (e.g. Rand equipment for test setups).

LDCs may be assigned a HLT role, see Section 17.2 for details.

4.3.3 The trigger database

The ***trigger*** database is used to define the detectors active for each possible trigger mask. For each ***trigger mask***, the list of ***detectors*** to be read out is given. This should be repeated for all the trigger masks defined in the ***roles*** database.

Each event should have at least one trigger mask active. The information contained in this database, combined with the ***runControl*** dynamic information, gives an exact description of all the possible triggering scenarios. This corresponds to an exact list of all the detectors that transfer data over their DDL(s) for each level-2 accepted trigger.

When running with a real LTU, it is recommended to define a trigger role for all possible class bits (usually 50), and associate each mask with all detectors. One can use the command `${DATE_DB_DIR}/daqDB_createAllTriggerClassMasks` for this purpose. Note that the role ID of a DATE trigger mask corresponds to the class number in the trigger class mask sent by the LTU.

4.3.4 The detectors database

The ***detectors*** database defines - on a detector by detector basis - the connections between the front-end equipment and the LDCs.

The list of LDCs or sub-detectors belonging to a each detector and sub-detector is stored there.

A host can be statically defined as part of a detector even if it is physically disconnected from it. This is the case - for example - of progressive installations or run-time disconnections/replacements. The information contained in this database, combined with the ***runControl*** dynamic information, shall return the exact set of LDCs connected to each detector during any given run.

The tree-like structure needs to be browsed recursively to know the top detector of a given role (if any). One can use the command `${DATE_DB_DIR}/getTopDetector.tcl` to retrieve it.

4.3.5 The event-building control database

For each event, the trigger system activates all the front-end equipments involved. This may or may not correspond to all the detectors/sub-detectors which are part of the DAQ system. The data coming from the "active" detectors is then collected by the LDCs who ship their events to one of the available GDCs. Here the ***event builder*** receives the sub-event(s) and acts on them following the directives given in the ***event building control*** database. At this moment the DATE ***event builder*** can follow three different policies:

a. ***build***: all the LDCs in the ***runControl*** dynamic database must contribute to a given event.

b. ***no-build***: LDCs can create sub-events independently from the rest of the DAQ system and these sub-events will be recorded individually.

c. ***partial build/no-build***: a well-defined set of LDCs will contribute to a given event that will be recorded either as a unique event or sub-event by sub-event.

The first case requires a sub-event from all LDCs participating in the run before building the event and delivering it to the recording channel. Typical use of this policy could be start-of-run or physics records. Please note that whenever the information stored in the event header, namely the trigger pattern and the detector pattern, is valid, the ***event builder*** will use it to establish the list of contributors to the event. A "build" may become a "partial build" whenever the detector pattern contains a subset of the LDCs which are active in the system.

The second case is very simple: LDCs may or may not create sub-events and these are recorded whenever the recording channel is ready. This policy could usually be applied to start-of-run-files and end-of-run-files.

The last case is the more complex and needs great care. Partial event building can be driven in three ways: by source, by detector set and by trigger mask. By source means that the list of expected sub-events for a given event can be derived by the source of the sub-event itself. For example, a calibration event coming from a given detector most likely covers only that detector. In the case of ALICE, all events have an associated detector mask and trigger mask specifying (indirectly) the LDCs who are expected to provide sub-events for the given event. Therefore it is possible to declare policies solely on a "detector set by detector set" or "trigger by trigger" basis. In all cases (by source and by trigger mask) the associated policy can instructs either to build or not to build the given event, according to the requirement dictated by the DAQ system.

All the rules are driven by an associated event type. It is possible to have different rules for the same event type, i.e. calibration events coming from one detector must be built while calibration events coming from another detector shall not be built.

Each record in this database should specify the event type (***SOR***, ***EOR***, etc.), the optional list of trigger masks, detectors or sub-detectors and the action (***BUILD*** or ***NOBUILD***). As many rules as needed can be given. If multiple rules can be activated

by the same event, the first one in order of the associated ***PRIORITY*** value is used. A rule accepts only specifiers of the same type, i.e. only trigger masks or only detectors.

4.3.6 The banks database

The ***banks*** database is used to define the memory banks required by each DATE host/entity, their sizes and the support mechanism(s) with their details.

Each record of the ***banks*** database contains a description of the banks to be implemented and their characteristics: type, name, size and content. The same host can implement multiple banks: one set per role and several subsets for each set. A proper definition of the memory banks should define an optimal and safe usage of the memory resources for each node of a DATE system.

The available supports are:

- ***IPC***: the key is the full path of a file to be used to map to the memory segment. This file is used to create a system-wide unique key used to identify the memory segment (see "`man ftok`" for more details). It is not necessary to specify it: when this field is empty, a unique name is assigned. The file is then created automatically at run-time in the ***${DATE_SITE_CONFIG}*** directory, with access permissions allowing read operation from everybody and write operation from ***DATE_USER_ID***. It is not recommended to manually specify a key file: great care must be taken so that the key is unique for each IPC bank, in case they are used on the same machine.
- ***PHYSMEM***: the key is the device used by the physmem driver.
- ***BIGPHYS***: the key is the device used by the bigphys driver.
- ***HEAP***: the process heap will be used (no multi-process sharing is possible). The key is dummy and not used.

The size gives the amount of memory to be allocated in bytes. It may also be given in kilobytes or megabytes (e.g. 10K, 1M).

If the block has to be used exclusively for the DATE control block, the size can be specified as "-1": this creates a block of the exact size needed to store the DATE control block.

The elements that can be allocated are:

- ***control***: the DATE control block (needed on all hosts part of the DAQ system).
- ***readout***: all the resources needed by ***readout***.
- ***readoutReadyFifo***: the FIFO used to transfer events out of the ***readout*** process.
- ***readoutFirstLevelVectors***: pool of first level vectors used to describe paged events.
- ***readoutSecondLevelVectors***: pool of second level vectors used to describe the data pages of paged events.
- ***readoutDataPages***: the pool for the payload of all types of events.

- ***edmReadyFifo***: the FIFO used to transfer events out of the ***edmAgent***.
- ***hltAgent***: all the resources needed by the ***hltAgent***.
- ***hltReadyFifo***: the FIFO used to transfer events out of the ***hltAgent***.
- ***hltSecondLevelVectors***: the pool of second level vectors available to the ***hltAgent***.
- ***hltDataPages***: the pool for payloads created by the ***hltAgent***.
- ***eventBuilder***: all the resources needed by the ***eventBuilder***.
- ***eventBuilderReadyFifo***: the FIFO used to store events out of the ***eventBuilder***.
- ***eventBuilderDataPages***: the pool used by the ***eventBuilder*** to store the events received from the LDCs.

The ***readout***, ***hltAgent*** and ***eventBuilder*** processes allocate all the resources they need (e.g. ***readout*** allocates ***readoutReadyFifo***, ***readoutFirstLevelVectors***, ***readoutSecondLevelVectors***, ***readoutDataPages*** and ***edmReadyFifo***). If needed, the specific resources can be tuned using the appropriate keyword, e.g.:

```
ROLE_NAME=myldc,SUPPORT==physmem1,KEY_PATH=/dev/physmem1,SIZE
=100M,PATTERN=readoutDataPages

ROLE_NAME=myldc,SUPPORT=ipc,KEY_PATH=,PATTERN=readout,SIZE=1M
```

This allocates 100 MB using ***PHYSMEM*** (device /dev/physmem1) to store the ***readout*** data pages and 1MByte using ***IPC*** for all other resources needed by the LDC role named `myldc`.

If the same memory block has to be used for multiple purposes (e.g. ready FIFO and first level vectors), the block is split evenly between the two resources. If the same block has to be used also for data pages (***readoutDataPages, hltDataPages, eventBuilderDataPages***) an heuristic algorithm is used to distribute the memory for the non-data and data blocks. Data blocks are given much more space than non-data blocks. As this algorithm may not result in an appropriate partitioning, we suggest to separate data from non-data blocks and to explicitly size the two banks separately.

If the same DATE host is used - under different role names - for different roles (e.g. LDC and EDM), different keys must be used, one for each role. Failure to do so may produce unpredictable results. This is done automatically for the ***IPC*** type if no key is provided.

When the database defines resources that are not active at run-time (e.g. EDM when the EDM checkbutton in the runControl window is not selected), these are not allocated. However, when certain resources are first required and then not needed, DATE will not remove them. For example, if a DATE run at one given moment includes the EDM and a memory bank is allocated for the ***edmAgent*** ready FIFO, the block will remain available (unused), even if the EDM is subsequently disabled. If the removal of the block is needed, this must be done via external methods (Operating System reboot or specific procedures).

The run-time configuration of the banks allocated by DATE can be dumped using the utility:

```
${DATE_BANKS_MANAGER_BIN}/dumpBanks
```

This tool can only run on hosts where DATE is currently running (or has run) and dumps the status of the various banks, their size and their addresses. The output of this utility reflects the run-time allocation of blocks and fifos according to the combination of static and dynamic information. The output from the utility is shown in Listing 4.1.

.

Listing 4.1 Example DATE banks dump

```
 1:  > ${DATE_BANKS_MANAGER_BIN}/dumpBanks ldc
 2:  rcShm: @0x40195000 offset:0 size:11088 bank:0
 3:  readoutReady: @0x40197b50 offset:11088 size:1037488 bank:0
 4:  readoutFirstLevel: @0x40295000 offset:0 size:1048576 bank:1
 5:  readoutSecondLevel: @0x40395000 offset:0 size:1048576 bank:2
 6:  readoutData: @0x40495000 offset:0 size:262144000 bank:3
 7:  edmReady: NOT AVAILABLE
 8:  hltReady: NOT AVAILABLE
 9:  hltSecondLevel: NOT AVAILABLE
10:  hltData: NOT AVAILABLE
11:  eventBuilderReady: NOT AVAILABLE
12:  eventBuilderData: NOT AVAILABLE
13:  physmem: @0x10060000 bank:3
14:  FIFOs: readoutReadyFifo == recorderInputFifo
15:  edmAgent: disabled (0) hltAgent: disabled (0)
```

This example describes an LDC where the DDL is active. The first bank (bank number 0) is used for the DATE control block (line 2) and the readout ready FIFO (line 3). One bank is allocated for the first level vectors (line 4) and another bank for the second level vectors (line 5). Finally the PHYSMEM - declared as bank 3 (line 13) - is used to store the ***readout*** data pages (line 6). EDM, HLT and event builder are not active on this node (lines 7-12 and 15). Finally there is one FIFO connecting the output of readout to the input of recorder (line 14).

4.4 Other centrally stored parameters

In addition to the above ***'static databases'*** describing the architecture and relations between the run-time entitites, the numerous DATE distributed processes also need some common parameters centrally defined and accessible to all of them

Items that can be modified by users are grouped in two families of DATE information: the ***Environment*** parameters and the ***Files***. The details about the package-specific configuration items is not described here but in the corresponding packages chapters. A third category, ***Detector Files***, is devoted to store files handled by the detector software (electronics initialization scripts, calibration procedures), but not used by DATE packages.

There are also a few tables meant to store internal DATE persistent parameters, not supposed to be modified directly. This is the case of the ***GLOBALS***, ***SOCKETS***, and ***DETECTOR_CODES*** tables.

Finally, the information related to readout equipments is saved in a set of tables specific to each kind of equipments: ***DDLin***, ***DDLout***, ***EQUIP_PARAM_...***(one

table per type of equipment), ***EQUIP_TYPES***, ***EQUIP_TYPES_FORMAT***. This information is accessible with ***editDb***.

The runControl also uses a dedicated table to store its run parameters: this is the ***runControl_runParameters*** table, editable through the ***runControl Human Interface***.

4.4.1 DATE globals

This table stores only a few hidden (i.e. not supposed to be modified) values:

- ***DB version*** : tags the database structure version. It is used to check that the installed version of DATE can run with this database. A mismatch will prevent DATE to run. Either the database should be updated (check the release notes, this is done with `${DATE_DB_DIR}/upgrade.tcl`), or the correct version of DATE should be installed.
- ***LHC period*** : used by ***mStreamRecorder*** to store and register the files in the correct location to be retrieved by offline analysis. One can get the current value with `${DATE_DB_BIN}/getLHCperiod.sh`
- ***Run Number*** : the latest run number used, incremented at each start of run.

There is no API to write to this table. Modifications are done directly with MySQL commands.

4.4.2 DATE sockets

TCP/IP sockets and ports are used to communicate between the DATE processes. To allow having several roles on the same machine, the port number used for each type of service are not fixed but dynamically assigned at the time of the definition of the ROLES table. This table is modified automatically by ***editDb*** when new roles are added or removed. It calls `${DATE_DB_BIN}/daqDB_fillSocketTable`, which assigns a port number for each service provided by a given DATE role on each machine. The port numbers are allocated within a fixed range of port numbers (***DATE_PORT_MIN*** and ***DATE_PORT_MAX*** defined in ***daqDB_fillSocketTable.c***, typically between 6001 and 6100). There is in principle no need to access (read or write) this information. The DATE services retrieve the information at run-time using the `dbGetPort()` function defined in ***${DATE_DB_DIR}/dateDbFile.h*** or directly from the shell utility `${DATE_DB_BIN}/daqDB_getPort`.

4.4.3 DATE detector codes

Depending on the context, a detector can be identified by a number (e.g. in a bit mask, or for for a DATE role ID), by a name (for human interfaces, or for a DATE role name) or by a 3 letter code. The table to convert between one form and the other is fixed, and defined in ***${DATE_DB_DIR}/detCodes.h***

This header file also provides means to retrieve quickly the information from a in-memory table. However, for convenience in SQL queries, the same information is also stored in the database ***DETECTOR_CODES*** table.

Consistency between the two is ensured with the utility `${DATE_DB_BIN}/daqDB_fillDetectorCodes` producing the corresponding statements to populate the DB at creation time, and `${DATE_DB_BIN}/daqDB_fillDetectorRoles`, which outputs the statements needed to create the corresponding DATE roles.

The output of both utilities in included in the DB creation script, which should be updated accordingly whenever the hardcoded list is modified.

It is very important that the detector code matches the role id in the ***roles*** database (this is ensured by the initial DB populating script). When adding a detector manually, ***editDb*** tries using the detector ID as role ID, if not already used elsewhere.

4.4.4 DATE Environment

The ***Environment*** table stores system environment variables that may be loaded at run-time. Each record consists of a name (the name of the environment variable), a value, a class telling in which context it is used (one of ***General Database Infologger User***), and a flag ***LOAD_BY_DEFAULT*** specifying if it should be loaded by the global DATE setup procedure. Default entries are populated when the configuration database is created. Further entries can be added manually, under the ***User*** class only.

Example variables include access parameters to the databases (configuration, logging, logbook, AMORE), the DIM DNS node, the path to the File Exchange Server, etc.

For performance reasons, care should be taken not to extend the run-time environment with unnecessary variables loaded by default and used only by a few processes. Other methods exist to store configuration information related to a limited number of processes (see next section).

All variables needed in the DATE environment should be defined here. The only exception being the initial access parameters to the database which need to be put in the file ***${DATE_SITE_PARAMS}***. These parameters are needed by the DATE setup script to load all other environment variables (and are overwritten in this process by the ones defined in the database). It is the only information that needs to be distributed manually on all the DATE hosts, everything else being then available from the central database.

${DATE_SITE_PARAMS} should contain the definition (on each line, one variable name and its value separated by a space) of ***DATE_DB_MYSQL_USER***, ***DATE_DB_MYSQL_PWD***, ***DATE_DB_MYSQL_HOST***, ***DATE_DB_MYSQL_DB***. It is populated automatically by the script creating a new DATE site.

The variables with the ***LOAD_BY_DEFAULT*** flag set are loaded in the environment by the DATE setup, using the ***loadEnvDB.tcl*** tool. This script prints the commands necessary to load the corresponding variables in the environment for ***bash*** and ***csh***, and allows to filter them by class. Use `${DATE_DB_DIR}/loadEnvDB.tcl -h` for details on the available options.

4.4.5 DATE Files

The ***Files*** table stores any kind of binary content. It can be seen as a shared filesystem, available from all DATE components.

Each entry is made of a ***PATH*** to identify the file (usually with a directory-like structure to sort the information), an optionnal ***HOST*** (in case of a file specific to a given host or role; this can be empty if it is of general use), a ***VALUE*** (it can be binary, but is usually textual for configuration files), a ***DESCRIPTION*** of the content, and a ***CLASS*** (***General*** for default resources, or ***User*** for the ones added later). The unique key to access the data is the couple ***PATH - HOST***.

There are two ways to access (read or write) the content of a file from a DATE process. The first involves direct access to the MySQL table and issues SQL queries loading the file in memory. The second is done with the shell utility `${DATE_DB_DIR}/copyFileDB` that allows to copy a file from the database to the local disk. Its content can then be read by classical means. Files with a relative path (not starting with '***/***') are loaded to / from ***${DATE_SITE}*** (or ***${DATE_SITE}/${DATE_HOSTNAME}*** for host specific files). The script either takes a local file and stores it in the database, or copies to the local disk a file from the database. The ***-help*** command line option gives an exhaustive list of possible options. This tool is mostly used to retrieve files from the database, whereas ***editDb*** (Section 4.5) provides a user friendly way to store and edit files in the database.

In the case of a file storing key / value pairs, the API provided by the header ***${DATE_DB_DIR}/dateDbFile.h*** offers and easy way to load the file and access the values. A command line tool, ***${DATE_DB_BIN}/dumpDbFile***, is based on this interface and gives a listing of the parsed file contents.

4.4.6 DATE Detector Files

A table named ***DETECTOR_CFG_FILES*** stores all the files for each detector defined in the detector codes table. A view ***DETECTOR_CFG_XXX*** is also created to selectively access the files of a given detector, ***XXX*** being the detector code.

This table structure is optional and not needed to run DATE. To create and remove it, the following utilities may be used: ***${DATE_DB_DIR}/daqDetDB_create*** and ***${DATE_DB_DIR}/daqDetDB_destroy***. A set of shell-like tools are provided to list / get / store / remove the files available: ***${DATE_DB_DIR}/daqDetDB_ls, ${DATE_DB_DIR}/daqDetDB_get, ${DATE_DB_DIR}/daqDetDB_store, ${DATE_DB_DIR}/daqDetDB_remove***. A graphical interface, ***${DATE_DB_DIR}/daqDetDB_browser***, is also provided to edit the files.

To use some of the commands above, the environment variable ***$DATE_DETECTOR_CODE*** must be defined in order to access the files of the given detector.

4.4.7 DATE readout equipment tables

The ***readout equipment*** configuration defines the readout system on the LDCs. This item is not in the ***dateDb*** package, but is part of the static configuration stored

in the database tables. The details about the various parameters are described in the relevant hardware chapters.

The ***DDLin*** table holds the mapping between DDL ids (e.g. used offline) and their space-optimized numbering used for the HLT to DAQ protocol (single bit mask reporting all the links).The static mapping used for the decoding of HLT decisions is defined in ***${DATE_DB_DIR}/dbHLTmask.c*** and the corresponding SQL statements to populate the database created by ***${DATE_DB_BIN}/dbHLTmask***. These statements are again included in the DB creation script every time the static definition changes (e.g. new detector or links). It usually goes together with an update in the ***HLTagent*** protocol.

In order to verify the consistency of the readout links and the correct cabling, the ***DDLin*** table has a field to register the remote SIU IDs. This is not used at run-time by DATE, but provides a convenient way to track changes in the cabling or hardware configuration, if needed. The command `${DATE_DB_DIR}/checkSIUs.tcl` allows to take a snapshot of the current hardware setup, and then check for changes. This is especially useful to notice cabling errors after detectors shutdown periods. Note that for this procedure all the SIUs should be up and running, and the DDL not used by other processes (in particular, it would not work during a run or an electronics configuration via DDL in progress).

To communicate with the electronics through the DDL, it is needed to know what RORC should be used on the client to access a given detector equipment ID. This information can be read from the database with shell commands `${DATE_DB_DIR}/daqDB_getRorcFromEqId`, `${DATE_DB_DIR}/daqDB_getRorcsFromLDC` and `${DATE_DB_DIR}/getDdlLinks.sh`. The necessary details to open the link (e.g. RORC serial number and channel) are returned by these tools, with different flavours of queries and filters.

4.5 The database editor

The configuration can be edited with the graphical user interface named ***editDb***. It is a Tcl/Tk application using SQL transactions to display and update the DATE database content (actually, only the subset accessible to users and directly related to the DATE static information). This tool relies on the tables definition and semantics of the database structure at the time it was developped, in order to provide high-level consistency checks and simple editing.

This chapter describes the features of the human interface. For a description of the configuration parameters, please consult Section 4.3.

To launch ***editDb***, type:

```
> editDb
```

All the menus have a ***Commit*** and a ***Rollback*** button. The configuration database is actually changed only after you click on the ***Commit*** button. You can edit the parameters, and then undo all the modifications made since last ***Commit*** with the ***Rollback*** button.

The ***editDb*** interface starts with the roles configuration display, as shown in Figure Figure 4.2. Use the buttons at the top of it to select a configuration item. The current one is highlighted in red. All the configuration displays share the function buttons at the bottom of the display, however, some configuration displays have extra controls.

To exit ***editDb***, click on the ***Quit*** button. You may quit only if all the changes to the database have been applied or canceled.

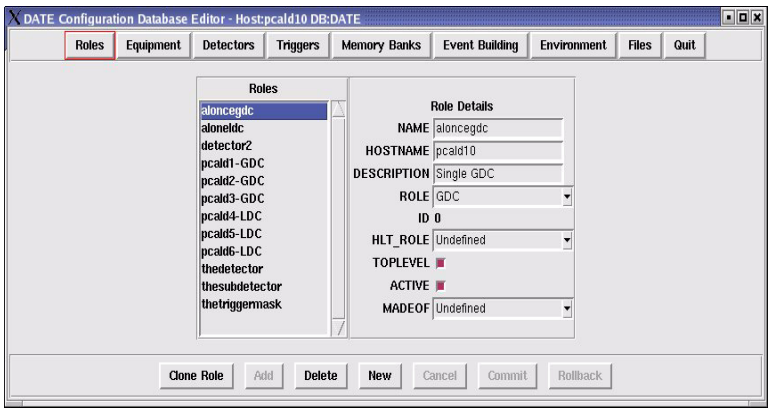

Figure 4.2 The initial editDb view.

To create a new role, click on the ***New*** button. This allows you to enter the details in the entry fields on the right hand side of the display. Once you are finished entering the details for the new role, click on the ***Add*** button. This will cause the new role to be added to the database. It will appear in the roles list on the left of the display. Now you are able to click on the ***Commit*** or ***Rollback*** button, to apply or to undo the changes in the database. To delete a role, first select it in the roles list on the left of the display, then click on the ***Delete*** button, finally click on the ***Commit*** button to accept the changes.

You can clone LDC and GDC roles. From the roles configuration display, first select the GDC or LDC you want to clone, then click on the ***Clone Role*** button. A window will pop up with some options. For cloning a GDC role you need to enter a space separated list of hostnames, as shown in Figure 4.3.

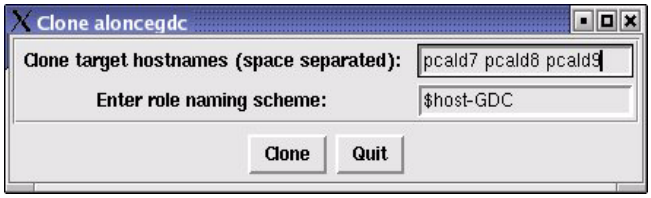

Figure 4.3 GDC cloning window.

For cloning an LDC role you need to enter the hostnames, choose whether to clone the LDC's equipment and choose a detector if you want to add the cloned LDCs to it, as shown in Figure 4.4. For both LDC and GDC roles you can change the naming schema: occurences of ***$host*** in the character string are replaced by the hostname.

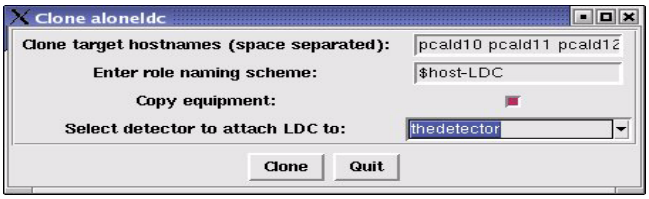

Figure 4.4 LDC cloning window.

After creating a new LDC role you can add an equipment to it, either by selecting it in the roles list and clicking on the extra button ***View Equipment*** or by clicking on the ***Equipment*** configuration display button and then selecting the LDC in the list on the left of the display. Clicking on the ***Add*** button lets you first choose what type of equipment to create by selecting it in the drop down box as shown in Figure 4.5. After selecting an equipment type click on the ***Create*** button. Now you are able to enter the equipment details. Once finished you need to click on the ***Add*** button to add the equipment to the database. Click on the ***Commit*** or ***Rollback*** button to accept or to undo the addition.

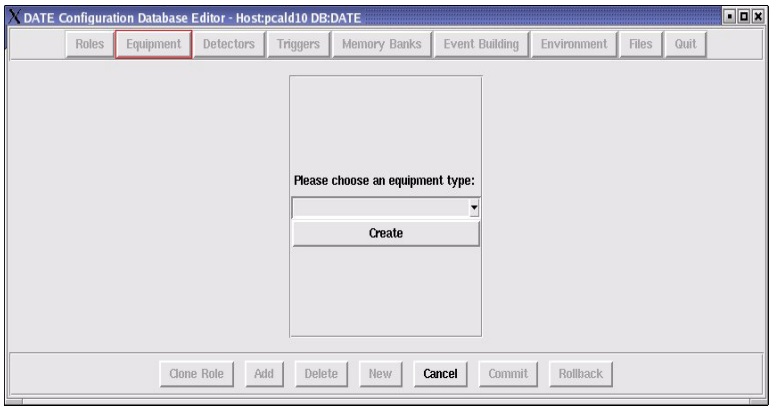

Figure 4.5 New equipment creation display.

Once you have added some equipments to a LDC, you will be able to see their details when you select them in the equipment list as shown in Figure 4.6. You can now edit any of the equipment fields. If you edit a field you will have to click on the ***Commit*** or ***Rollback*** button before you can change to a different configuration display. Inactive equipments appear in red in the equipment listbox.

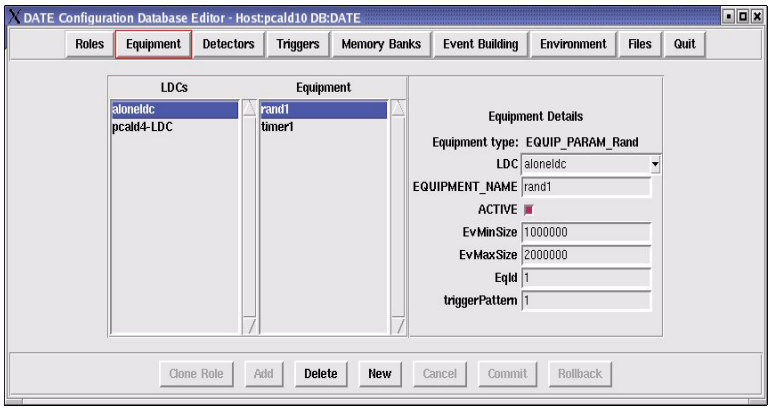

Figure 4.6 Equipment configuration display.

Once you have a Detector or Sub-detector role you can add components to it. Either select the detector in the roles configuration display and press on the ***View Components*** button, or click on the detectors configuration display button, then select the detector in the detector list. You should see a list of components belonging to the detector in the ***Made Of*** list, and a list of available components in the ***Available Components*** list, as shown in Figure 4.7.

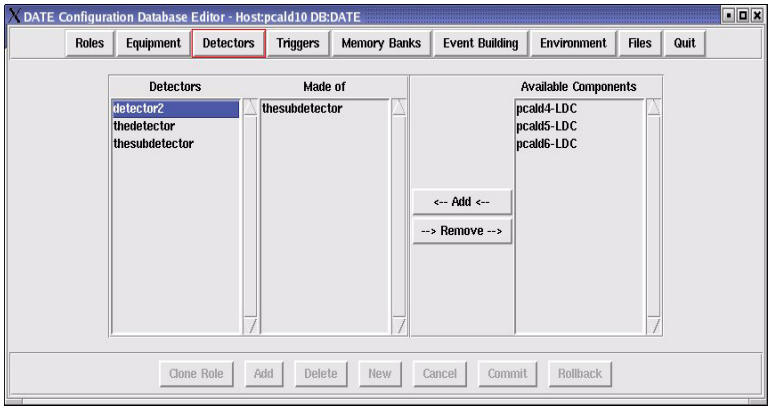

Figure 4.7 Detectors configuration display.

Now you are able to add components by selecting them in the ***Available Components*** list and then clicking on the ***Add*** button. You can remove components by selecting them in the ***Made Of*** list and clicking on the ***Remove*** button. If you make any change, then you need to click on the ***Commit*** or ***Rollback*** button.

Once you have a TriggerMask or TriggerHost role you can add components to it. Either select the trigger in the roles configuration display and press on ***View Components***, or click on the triggers configuration display button, then select the detector in the trigger list. You should see a list of components belonging to the trigger in the ***Made Of*** list, and a list of available components in the ***Available Detectors*** list, as shown in Figure 4.8.

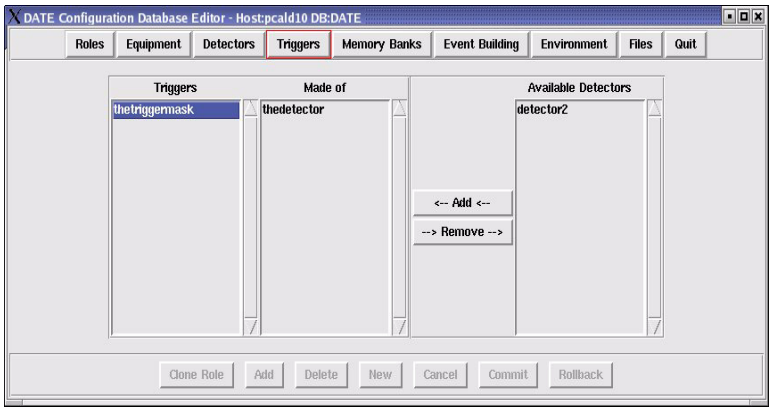

Figure 4.8 Triggers configuration display.

Now you are able to add components by selecting them in the ***Available Detectors*** list and clicking on the ***Add*** button. You can also remove components by selecting them in the ***Made Of*** list and clicking on the ***Remove*** button. If you make any changes, then you need to click on the ***Commit*** or ***Rollback*** button.

Clicking on the ***Membanks*** configuration display button shows the memory banks that are defined in the ***Membanks*** list, as shown in Figure 4.9. The details for the currently selected memory bank are displayed on the right. To add a new membank click on the ***New*** button. This clears the entry fields under the ***Membank Details*** label. Enter the details of the new memory bank, and then click on the ***Add*** button. You can cancel the addition of a new memory bank by clicking on the ***Cancel*** button. You can also edit the details of the currently selected memory bank. You need to click on the ***Commit*** or ***Rollback*** button when you have finished the changes.

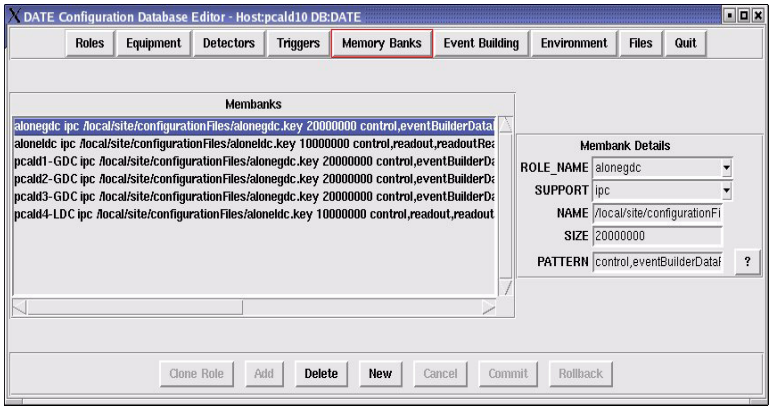

Figure 4.9 Membanks configuration display.

Clicking on the ***Event Building*** configuration display button shows the event building rules that are defined in the ***Rules*** list, as shown in Figure 4.10. The details for the currently selected rule are displayed on the right. To add a new rule click on the ***New*** button. This clears the entry fields under the ***Rule Details*** label. Enter the details of the new rule, and then click on the ***Add*** button. You can cancel

the addition of a new rule by clicking on the ***Cancel*** button. You can also edit the details of the currently selected rule. Click on the ***Commit*** or ***Rollback*** button when you have finished the changes.

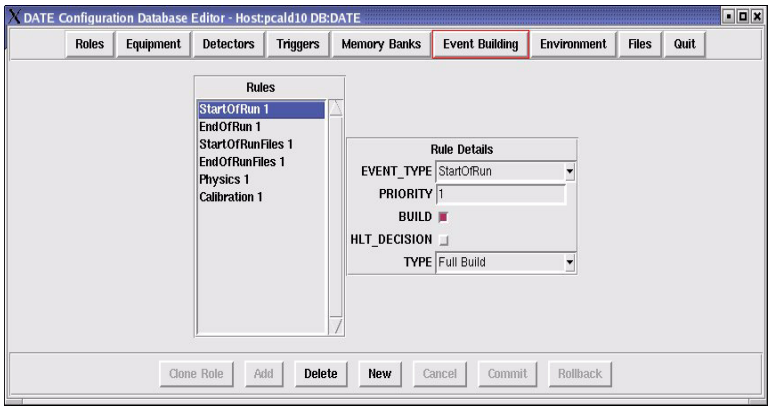

Figure 4.10 Event building rules configuration display.

Clicking on the ***Environment*** configuration display button shows you a dropdown list where you can choose the class of variables you want to see. In the list below you will see the variables defined for the current class. On the right of the display you see the details for the currently selected variable from the listbox. For all the classes except the ***User*** class you can only change the ***value*** field, as shown in Figure 4.11.

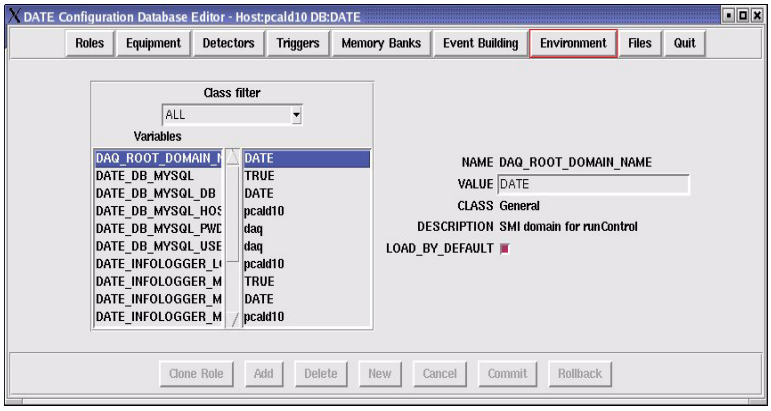

Figure 4.11 Environment variables configuration display.

If you select ***User*** from the dropdown list you are able to add and delete user defined variables. You can also edit the value and description fields for each variable, as shown in Figure 4.12.

If the LOAD_BY_DEFAULT flag is set, the environment variable is loaded into the environment when calling the DATE setup procedure.

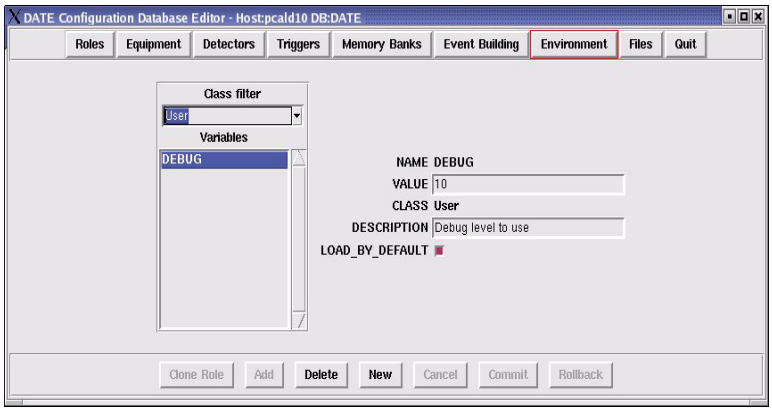

Figure 4.12 Environment variables configuration display showing user defined variables.

The DATE MySQL configuration system allows to store files (ASCII or binary). This is convenient to avoid deploying a shared file system to distribute files on DATE hosts. Clicking on the ***Files*** configuration display button shows a list of file paths with the host they will be placed on, or a path alone which means the file will be placed on all DATE hosts. On the right of the display are the details for the currently selected item in the listbox, as shown in Figure 4.13. To create a new file entry click on the ***New*** button and fill in the entry fields, clicking on the ***Get file*** button brings up a file selection display which lets you select the file you want to upload. Leaving the ***Host*** entry field blank is interpreted to mean all hosts. Click on the ***Add*** button to add the details and the file to the database. Click on the ***Commit*** or ***Rollback*** button when you have finished the changes.

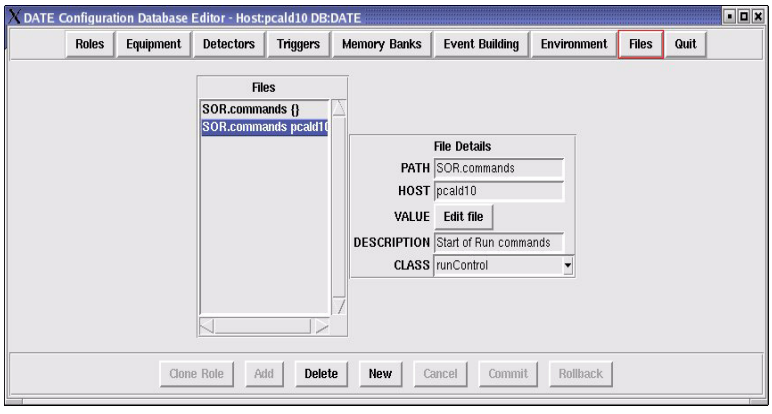

Figure 4.13 Files configuration display.

To edit a file click on the entry in the Files list then click on the ***Edit file*** button. Tthis will launch an editor where you may make changes to the file. Once you have made the changes, click save then exit the editor. Clicking on the ***Commit*** button will apply the changes to the database.

SOR/EOR commands/files are automatically copied on to the target host when ***readout*** or the ***eventBuilder*** starts.

Other files can be copied locally with the ***copyFileDB.tcl*** script (see Section 4.4.5 for details).

4.6 Example of a DAQ system

We will now see how to define an example DAQ system, as described in Figure 4.14.

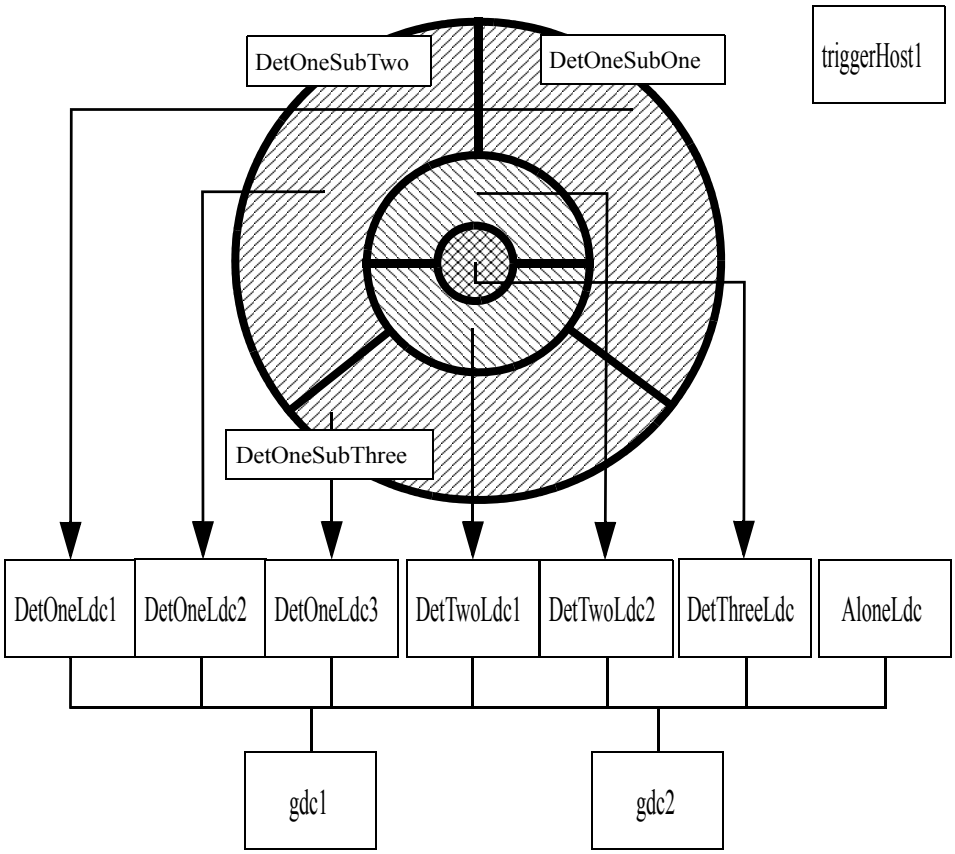

Figure 4.14 Example of a DAQ system.

This DAQ system is made of three detectors, attached to six LDCs. One extra LDC has been allocated for non-detector related tasks (e.g. Trigger System). The three detectors are called ***DetOne***, ***DetTwo*** and ***DetThree***. The first of the three detectors (***DetOne***) has been partitioned into three sub-detectors: ***DetOneSubOne***, ***DetOneSubTwo*** and ***DetOneSubThree***. The experiment allows two triggers: one that activates all the three detectors and a second trigger that activates ***DetOne*** alone. Similarly, for calibration events we want ***DetOne*** to receive a stand-alone calibration and a second calibration to go to all the detectors. The event builders must build all ***PHYSICS*** and ***CALIBRATION*** events.

This example is described in ASCII files format in order to give an idea of what parameters should be stored in the database with ***editDb***. The files can not be used directly, but give a realistic dump of the parameters to be defined in the database to

implement such a DAQ system. These example files are available in the directory ***${DATE_DB_DIR}/testConfig***, and are shown in Listing 4.2.

Listing 4.2 Example of configuration files

```
 1: > cd ${DATE_DB_DIR}
 2: > ls -1 testConfig
 3: dateBanks.config
 4: dateRoles.config
 5: detectors.config
 6: eventBuildingControl.config
 7: triggers.config
```

The first database we examine is the **roles** database, stored in ***${DATE_DB_DIR}/testConfig/dateRoles.config***.

Listing 4.3 Example of roles database

```
 1: > cat ${DATE_DB_DIR}/testConfig/dateRoles.config
 2: >LDC
 3:  DetOneLdc1   1 "DetOne LDC #1" hostname=host1
 4:  DetOneLdc2   2 "DetOne LDC #2" hostname=host2
 5:  DetOneLdc3   3 "DetOne LDC #3" hostname=host3
 6:
 7:  DetTwoLdc1  10 "DetTwo LDC #1" hostname=host4
 8:  DetTwoLdc2  11 "DetTwo LDC #2" hostname=host5
 9:
10:  DetThreeLdc 20 "DetThree LDC"  hostname=host6
11:
12:  AloneLdc    30 "Single LDC"    hostname=host7 topLevel=Y
13:
14: >GDC
15:  gdc1 1 "GDC #1" hostname=host8 topLevel=Y
16:  gdc2 2 "GDC #2" hostname=host9 topLevel=Y
17:
18: >TRIGGER_HOST
19:  triggerHost1 1 "Trigger host 1" hostname=hostT
20:
21: >DETECTORS
22:  DetOne   1 "Detector 1" topLevel=Y
23:  DetTwo   2 "Detector 2" topLevel=Y
24:  DetThree 3 "Detector 3" topLevel=Y
25:
26: >SUBDETECTORS
27:  DetOneSubOne   1 "Detector 1 Sub-detector 1"
28:  DetOneSubTwo   2 "Detector 1 Sub-detector 2"
29:  DetOneSubThree 3 "Detector 1 Sub-detector 3"
30:
31: >TRIGGER_MASK
32:  TriggerMask1 1 "Trigger mask 1"
33:  TriggerMask2 2 "Trigger mask 2"
```

The LDCs declaration (lines 2-12) concerns the seven LDCs. For each machine we have a DATE name, the identifier, a short description and the hostname. Please note that most of the LDCs are not marked as “topLevel” and therefore cannot be directly selected from the ***runControl Human Interface***. Only AloneLdc can be directly selected or deselected.

The GDCs are declared in lines 14-16. All GDCs can be directly selected or deselected via the ***runControl Human Interface***.

Lines 18-19 declare the trigger host.

The three detectors and the three sub detectors of the first detector are declared in lines 21-29. The three detectors can be (de)selected via the ***runControl Human Interface***.

Finally lines 31-33 declare the two trigger masks available in this DAQ system.

The two trigger masks declarations are illustrated below:

Listing 4.4 Example of trigger configuration

```
 1:  > cat ${DATE_DB_DIR}/testConfig/triggers.config
 2:  >TRIGGER_MASK
 3:   TriggerMask1 DetOne DetTwo DetThree
 4:   TriggerMask2 DetOne
```

The first trigger (triggerMask1) activates all the detectors while the second trigger mask (triggerMask2) activates the first detector only.

The detectors are defined as follows:

Listing 4.5 Example of detectors configuration

```
 1:  > cat ${DATE_DB_DIR}/testConfig/detectors.config
 2:  >DETECTORS
 3:   DetOne   DetOneSubOne DetOneSubTwo DetOneSubThree
 4:   DetTwo   DetTwoLdc1 DetTwoLdc2
 5:   DetThree DetThreeLdc
 6:
 7:  >SUBDETECTORS
 8:   DetOneSubOne   DetOneLdc1
 9:   DetOneSubTwo   DetOneLdc2
10:   DetOneSubThree DetOneLdc3
```

The three detectors are defined in lines 2-5. The sub-detectors of the first detector are defined in lines 7-10.

The event building policies are defined as:

Listing 4.6 Example of event-building configuration

```
 1:  > cat ${DATE_DB_DIR}/testConfig/eventBuildingControl.config
 2:  >EVENT_BUILDING_CONTROL
 3:  SOR  nobuild
 4:  SORF nobuild
 5:  EOR  nobuild
 6:  EORF nobuild
 7:
 8:  PHY TriggerMask1 build
 9:  PHY TriggerMask2 build
10:  PHY build
11:
12:  CAL DetOne build
13:
14:  CAL DetOne DetTwo DetThree build
15:
```

The above configuration example drives the event builder not to build SOR and EOR events (lines 3-6). Physics events triggered by **`TriggerMask1`** will be built as well as events triggered by **`TriggerMask2`**. Physics events without trigger mask will be built from all the LDCs. Calibration events involving **`DetOne`** alone will be

built using this detector, while calibration events involving all detectors will be built using data coming from all LDCs. The detectors involved in each calibration trigger are extracted from the **eventDetectorPattern** field of the event header.

The DATE banks are defined as:

Listing 4.7 Example of banks configuration

```
 1:  > cat ${DATE_DB_DIR}/testConfig/dateBanks.config
 2:  >BANKS
 3:
 4:   # --- LDCs ---
 5:   DetOneLdc1 IPC ${DATE_SITE_CONFIG}/LDC.key 150000 control readout
 6:   DetOneLdc2 IPC ${DATE_SITE_CONFIG}/LDC.key 150000 control readout
 7:   DetOneLdc3 IPC ${DATE_SITE_CONFIG}/LDC.key 300000 control readout
 8:
 9:   DetTwoLdc1 \
10:    PHYSMEM /dev/physmem1          5M   readoutDataPages \
11:    IPC     ${DATE_SITE_CONFIG}/LDC.key  1.5M readout \
12:    IPC     ${DATE_SITE_CONFIG}/LDC.key1 *    control
13:   DetTwoLdc2 \
14:    PHYSMEM /dev/physmem1          5M   readoutDataPages \
15:    IPC     ${DATE_SITE_CONFIG}/LDC.key  1.5M readout \
16:    IPC     ${DATE_SITE_CONFIG}/LDC.key1 *    control
17:
18:   DetThreeLdc IPC ${DATE_SITE_CONFIG}/LDC.key 100K control readout
19:
20:   AloneLdc IPC ${DATE_SITE_CONFIG}/LDC.key 10000 control readout
21:
22:   # --- GDCs ---
23:   gdc1 IPC ${DATE_SITE_CONFIG}/GDC.key 10M control eventBuilder
24:   gdc2 IPC ${DATE_SITE_CONFIG}/GDC.key 10M control eventBuilder
```

The banks to be implemented in the three LDCs of the first detector are defined in lines 5-7. They all contain the resources needed for control and data flow. They are all implemented using IPC shared memory and their sizes are 150000 (LDCs 1 and 2) and 300000 (LDC 3) bytes. These block are partitioned into the separate regions needed to store the DATE control block, the data pages, the FIFOs and all other resources needed by ***readout***.

The two LDCs of the second detector use PHYSMEM to allocate their ***readout*** data pages. This is the typical case for a DDL-based DAQ system. The other resources needed by the ***readout*** process are allocated using IPC via the given keys for a size of 1.5 MB. The DATE control segment is handled via IPC, using a separate block whose size is equal to the size of the DATE control block itself.

The LDC of the third detector has 100 KB handled via IPC. The same mechanism is used for the stand-alone LDC, with a size of 10000 bytes.

The two GDCs use an identical configuration of a single 10 MB IPC segment for the DATE control block and for all the resources needed by the ***event builder***.

Using the above example configuration to fill the database, here is the report from the ***dumpDbs*** utility:

Listing 4.8 Example of dumpDbs output

```
> dumpDbs
Roles DB:
0) id: 1 LDC DetOneLdc1 hostname:host1 "DetOne LDC #1" madeOf:Undefined bankDescriptor:0
[...]
9) id: 1 Detector DetOne hostname:N/A "Detector 1" madeOf:Subdetector TOP-LEVEL
[...]
12) id: 1 Subdetector DetOneSubOne hostname:N/A "Detector 1 Sub-detector 1" madeOf:LDC
[...]
15) id: 1 Trigger-Host triggerHost1 hostname:hostT "TriggerHost 1" madeOf:Undefined
16) id: 1 Trigger-Mask TriggerMask1 hostname:N/A "Trigger mask 1" madeOf:Undefined
17) id: 2 Trigger-Mask TriggerMask2 hostname:N/A "Trigger mask 2" madeOf:Undefined
Max LDC:30, Max GDC:2, Max Detector:3, Max Subdetector:3, Max Trigger-Host:1, Max Trigger-Mask:2
.....................................................
Trigger DB:
0) id: 1 TriggerMask1
   detectorPattern:0000000e = DetOne+DetTwo+DetThree =>
      ldcPattern:DetOneLdc1+DetOneLdc2+DetOneLdc3+DetTwoLdc1+DetTwoLdc2+DetThreeLdc (6 LDCs)
1) id: 2 TriggerMask2
   detectorPattern:00000002 = DetOne
      => ldcPattern:DetOneLdc1+DetOneLdc2+DetOneLdc3 (3 LDCs)
.....................................................
Detectors DB:
0) Detector id: 1 DetOne (made of:Subdetector)
   subdetectorPattern:DetOneSubOne+DetOneSubTwo+DetOneSubThree (3 subdetectors)
   => ldcPattern:DetOneLdc1+DetOneLdc2+DetOneLdc3 (3 LDCs)
1) Detector id: 2 DetTwo (made of:LDC)
   ldcPattern:DetTwoLdc1+DetTwoLdc2 (2 LDCs)
   => ldcPattern:DetTwoLdc1+DetTwoLdc2 (2 LDCs)
[...]
3) Subdetector id: 1 DetOneSubOne (made of:LDC)
   ldcPattern:DetOneLdc1 (1 LDC)
[...]
.....................................................
Banks DB:
LDC DetOneLdc1 (host1): descriptor:0 1 bank(s)
 ipc "${DATE_SITE_CONFIG}/LDC.key" size:150000 => control readout readoutReadyFifo readoutFirstLevelVectors readoutSecondLevelVector readoutDataPages edmReadyFifo
[...]
LDC DetTwoLdc1 (host4): descriptor:3 3 bank(s)
 physmem "/dev/physmem.device" size:5242880 => readoutDataPages
 ipc "${DATE_SITE_CONFIG}/LDC.key" size:1572864 => readout readoutReadyFifo readoutFirstLevelVectors readoutSecondLevelVectors edmReadyFifo
 ipc "${DATE_SITE_CONFIG}/LDC.key1" size:-1 => control
[...]
GDC gdc1 (host8): descriptor:7 1 bank(s)
 ipc "${DATE_SITE_CONFIG}/GDC.key" size:10485760 => control eventBuilder eventBuilderReadyFifo eventBuilderDataPages
[...]
.....................................................
Event building control DB:
0) eventType:StartOfRun all-events NO-BUILD
[...]
4) eventType:Physics triggerPattern:00000000-00000002=1 BUILD
   1:TriggerMask1
5) eventType:Physics triggerPattern:00000000-00000004=2 BUILD
   2:TriggerMask2
[...]
.....................................................
```

The output of the utility has been edited for brevity and formatting purposes.

The following declared roles are shown:

- LDCs (line 3).
- Detector and SubDetectors (lines 5,7).
- Trigger host (line 9).
- Trigger masks (lines 10-11).

The two elements added dynamically by the database package to each role are the madeOf field (used to specify the components of a role) and the bankDescriptor ID pointing to the specific host role entry in the banks database. Note the TOP-LEVEL attribute that specifies the entities that can be directly selected using the ***runControl Human Interface***.

Line 12 reports the number of entities declared in the database. These values are used by DATE to allocate global structures within a run.

Follows the trigger database (lines 14-20). The dumpDbs utility complements the static information retrieved from the databases with some derived information, such as the detector and LDC pattern corresponding to each trigger mask.

For the detectors database (lines 22-32) the dumpDbs utility appends the derived information of the list of LDCs corresponding to each detector and subdetector.

The banks database is reported (lines 34-45) for each host with the list of all banks, their support, size and entities. The individual sizes are not given, since these can be computed only at run-time according to the actual configuration.

Finally, the event-building control is shown (lines 47-54) with some NO BUILD rules (for start of run records) and some “by-trigger” rules.

4.7 The programming interface

The DATE database package provides a common interface to access some of the database content, in particular the data of the ***‘static databases’*** described in Section 4.3. It is not required to use this interface to operate DATE. Information in this chapter is given for developer information only, since it is mainly used by DATE actors.

The way to access data is the same for all information: the database is opened, loaded, and is mapped onto the process address space. Access is provided via memory-mapped, read-only operations. After a successful mapping the following information is made available to the calling process:

a. pointer to an array describing the database, the size (number of entries) of the database is given by an int and can also be given by the array itself (each DB has the last entry with invalid ID set to ***DB_ID_INVALID***).

b. a set of max*Id variables where the maximum defined ID is given. This value can be used to size at run-time structures with one element for each ID. The value of the maximum ID is guaranteed to be less than or equal to the static

maximum value as defined in ***${DATE_COMMON_DEFS}/event.h***.

Once a database is successful mapped into the process address space, it can be reloaded only explicitly via an ***unload***/***load*** sequence. Consecutive ***load*** calls produce no effect.

All entries require the definitions given by the files ***${DATE_COMMON_DEFS}/event.h*** and ***${DATE_DB_DIR}/dateDb.h***. These files can be included either by specifying the full path or via "**-I**" C include directive.

In this section, a definition is given for the following entities:

- macros used to manipulate and test bit masks and bit patterns.
- base types used to represent the entities defined in the static databases.
- entries used to load, unload and perform other operations through the static databases.
- pointers and variables where the databases and their associated information is made available to the calling process.

Several other access methods to more specific data are available and defined in ***${DATE_DB_DIR}/dateDb.h*** and fully documented in the header file. It includes means to list equipment, retrieve run parameters, parse configuration files, browser detector ID / name / code mapping, etc.

Bit test macro

C Synopsis

```
#include "event.h"
#include "dateDb.h"

#define DB_TEST_BIT( bitMaskOrPattern, id )
```

Description

The **DB_TEST_BIT** macro can be used for all **bit masks** and **bit patterns** to test for the assertion of a given **ID**. It returns the boolean value **TRUE** if the *id* is set, **FALSE** otherwise. The macro can be used directly or indirectly in boolean-driven statements, e.g. the following lines of code:

```
if ( DB_TEST_BIT(mask, id) ) idIsSet();
if ( DB_TEST_BIT(mask, id) == TRUE ) idIsSet();
```

shall execute the function `idIsSet()` if *id* is set in *mask*.

dbIdType
DB_ID_INVALID
dbLdcPatternType
eventDetectorPatternType
eventTriggerPatternType

C Synopsis

```
#include "event.h"
#include "dateDb.h"
```

Description

The basic type ***dbIdType*** defines the data storage element used to represent all IDs used in the DATE static databases, being LDC, GDC, Trigger Host, Trigger Mask, Detector, Subdetector or EDM Host. Within the same role, different entities must have different IDs. The same ID can be used for entities of different roles.

An ID equal to ***DB_ID_INVALID*** has either not been set or it is not applicable to a given record / entity.

DB_WORDS_IN_LDC_MASK

C Synopsis

```
#include "event.h"
#include "dateDb.h"
```

Description

Number of 32-bit words used to store a **dbLdcPatternType**. Can be used to scan and size a LDC pattern.

dbRoleType

C Synopsis

```
#include "event.h"
#include "dateDb.h"

typedef enum {
   dbRoleUndefined,
   dbRoleUnknown,
   dbRoleGdc,
   dbRoleLdc,
   dbRoleEdmHost,
   dbRoleHltProxy,
   dbRoleHltProducer,
   dbRoleHltRoot,
   dbRoleDetector,
   dbRoleSubdetector,
   dbRoleTriggerHost,
   dbRoleTriggerMask,

   dbRoleDdg,
   dbRoleFilter
} dbRoleType;
```

Description The dbRoleType enumeration type is used to define the role of a given record.

dbMemType

C Synopsis

```
#include "event.h"
#include "dateDb.h"

typedef enum {
   dbMemUndefined,
   dbMemIpc,
   dbMemHeap,
   dbMemBigphys,
   dbMemPhysmem
} dbMemType;
```

Description Define the method used to implement the control and (optionally) data buffers. This enumeration applies only to hosts where DATE actors need run-time memory support (LDC, GDC, EDM, Trigger Host). Where this is not applicable or has not been correctly defined, the ***dbMemUndefined*** value is used.

dbMemTypeNames

C Synopsis

```
#include "event.h"
#include "dateDb.h"

const char * const dbMemTypeNames[];
```

Description Maps any memory type to a description string. Use as `dbMemTypeNames[ dbMemType ]`.

dbRoleDescriptor

C Synopsis

```
#include "event.h"
#include "dateDb.h"

typedef struct {
   char *name;
   char *hostname;
   char *description;
   dbIdType id;
   dbRoleType role;
   dbHltRoleType hltRole;
   unsigned topLevel : 1;
   unsigned active : 1;
   dbRoleType madeOf;
   int bankDescriptor;
} dbRoleDescriptor;
```

Description This is the structure used to represent the records of the **roles** database. It includes the following fields:

- name of the entity.
- hostname of the entity (where applicable).
- description of the entity.
- **ID** of the entity.
- **role** of the entity.
- **hltRole** of the entity.
- flag **topLevel** to designate the entity as selectable from the ***runControl Human Interface*** (if ***TRUE***).
- active flag to indicate whether the entity is active or not.
- the **madeOf** field to tell what the role is made of (e.g. a **detector** will be made of either **subDetectors** or **LDCs**).
- index of the **bankDescriptor** that defines the banks and supports to be made available on the given role (where applicable).

dbTriggerDescriptor

C Synopsis

```
#include "event.h"
#include "dateDb.h"

typedef struct {
   dbIdType id;
   eventDetectorPatternType detectorPattern;
} dbTriggerDescriptor;
```

Description Represent a record from the trigger static database. The record includes:

- **ID** of the trigger, as defined in the **roles** database.
- the **detector pattern** associated to the given trigger.

Each record of this type associates a trigger (identified by its **ID**) to the corresponding detector pattern where **TEST_BIT(*detectorPattern, detectorId*)** returns **TRUE** if the detector with the given **ID** is active in the given trigger mask.

dbDetectorDescriptor

C Synopsis

```
#include "event.h"
#include "dateDb.h"

typedef struct {
   dbIdType id;
   dbRoleType role;
```

```
   dbLdcPatternType componentPattern;
} dbDetectorDescriptor;
```

Description Represent a record from the detectors static database. The record includes:

- **`ID`** of the detector, as defined in the **`roles`** database.
- role (**`detector`** or **`subDetector`**) of the described entity.
- the **`ldc pattern`** or **`detector pattern`** associated to the given detector.

The structure ***`dbDetectorDescriptor`*** describes the sub-detectors or the LDCs that belong to the given detector (regardless of the status of their actual connection).

dbEventBuildingRule

C Synopsis

```
#include "event.h"
#include "dateDb.h"

typedef struct {
   eventTypeType eventType;
   unsigned build : 1
   enum {
      fullBuild,
      useDetectorPattern,
      useTriggerPattern
   } type;
   union {
      eventDetectorPatternType detectorPattern;
      eventTriggerPatternType triggerPattern;
   } pattern;
} dbEventBuildingRule;
```

Description Represent a record from the event building database. The record includes:

- the type of the event associated to the rule.
- a **`build`** (***`TRUE`***) / **`no-build`** (***`FALSE`***) flag.
- the type of build: **`full`**, partial by **`detector pattern`** or partial by **`trigger pattern`**.
- the **`detector pattern`** or the **`trigger pattern`** used for partial event building (where applicable).

The structure ***`eventBuildingRule`*** describes the rules followed by the event builder. These rules can specify either a build or a no-build rule on a per-event type basis. Furthermore, the rule can result in a request for partial building.

dbBankType

C Synopsis

```
#include "event.h"
#include "dateDb.h"

typedef enum {
  dbBankControl,

  dbBankReadout,
    dbBankReadoutReadyFifo,
    dbBankReadoutFirstLevelVectors,
    dbBankReadoutSecondLevelVectors,
    dbBankReadoutDataPages,

  dbBankHltAgent,
    dbBankHltReadyFifo,
    dbBankHltSecondLevelVectors,
    dbBankHltDataPages,

  dbBankEventBuilder,
    dbBankEventBuilderReadyFifo,
    dbBankEventBuilderDataPages
} dbBankType;
```

Description

Describes the various banks that can be made available on the hosts where DATE actors can run.

The `dbBankControl` bank contains the control section. It must be present on all DATE hosts.

The `dbBankReadout` bank contains all banks needed by a LDC. This bank can be partitioned into the `dbBankReadoutReadyFifo`, `dbBankReadoutFirstLevelVectors`, `dbBankReadoutSecondLevelVectors` and `dbBankReadoutDataPages` banks.

Similarly, the `dbBankHltAgent` bank contains all banks needed by a HLT agent. This bank can be partitioned into the `dbBankHltReadyFifo`, the `dbBankHltSecondLevelVectors` and the `dbBankHltDataPages` banks.

The `dbBankEventBuilder` bank contains all entities needed on GDCs. It can be partitioned into the sub-entities `dbBankEventBuilderReadyFifo` and `dbBankEventBuilderDataPages`.

dbBankNames

C Synopsis

```
#include "event.h"
#include "dateDb.h"

const char * const dbBankNames[];
```

Description Maps any memory bank to a description string. Use as `dbBankNames[ dbBankType ]`.

dbBankPatternType

C Synopsis
```
#include "event.h"
#include "dateDb.h"
```

Description Basic type used to describe a Memory Bank pattern.

dbBankComponents

C Synopsis
```
#include "event.h"
#include "dateDb.h"

const dbBankPatternType dbBankComponents[];
```

Description Read-only array containing, for each DATE bank, the pattern of its components (if any). The information stored in this array can be used to find out what the sub-entities are. For example, the entry corresponding to the `dbBankEventBuilder` contains `dbBankEventBuilderReadyFifo` and `dbBankEventBuilderDataPages`, therefore the two corresponding bits of `dbBankComponents[ dbBankEventBuilder ]` has the two bits `dbBankEventBuilderReadyFifo` and `dbBankEventBuilderDataPages` set.

dbSingleBankDescriptor
dbBankDescriptor

C Synopsis
```
#include "event.h"
#include "dateDb.h"

typedef struct {
  dbMemType support;
  char *name;
  int size;
  dbBankPatternType pattern;
} dbSingleBankDescriptor;

typedef struct {
  int numBanks;
  dbSingleBankDescriptor *banks;
} dbBankDescriptor;
```

Description Structures used to describe a single bank (within one DATE host) and all the banks (on one DATE host).

The `size` field contains either an explicit total amount of memory (in bytes) to be used to store the bank or the value -1, meaning that the bank will be sized according to its content (if possible).

DB_LOAD_OK
DB_UNLOAD_OK
DB_LOAD_ERROR
DB_PARSE_ERROR
DB_INTERNAL_ERROR
DB_BAD_SIZING
DB_PAR_ERROR
DB_UNKNOWN_ID

C Synopsis

```
#include "event.h"
#include "dateDb.h"
```

Description

The above definitions cover all case of errors during handling of the static databases.

DB_LOAD_OK and ***DB_UNLOAD_OK*** report the successful load / unload of a database.

DB_LOAD_ERROR reports a problem loading a database due to the underlying file system. If this error is received, check for the presence of the file and for the access permissions.

DB_PARSE_ERROR reports a problem parsing a database. The line that generated the error can be retrieved using the ***dbGetLastLine*** call.

DB_INTERNAL_ERROR reports an unexpected error condition. More explanations may be found in the messages stored in ***dateDb*** log facility. This error should be reported to the DATE support team.

DB_BAD_SIZING is the result of a **ID** out of range or a limit out of range.

DB_PAR_ERROR is returned when one or more input parameters have invalid values.

DB_UNKNOWN_ID corresponds to an ID that is within the valid values but has no corresponding DATE role associated.

All the above errors can be decoded and printed using the ***dbDecodeStatus*** call.

dbDecodeStatus

C Synopsis

```
#include "event.h"
#include "dateDb.h"

char *dbDecodeStatus( status );
```

Description This routine will map the given status code - as returned by any of the ***dateDb*** routines - to a description string.

Returns The description string if the error code is amongst the allowed choices, an error message otherwise. The pointer is to a static location, overwritten by subsequent calls to the routine.

dbGetLastLine

C Synopsis
```
#include "event.h"
#include "dateDb.h"

char *dbGetLastLine();
```

Description Get the last line decoded by the last `dbLoad*()` call. This line can be used for diagnostic purposes, e.g. to understand parse errors. This call is relevant only when operating with file-based databases, not in MySQL mode.

Returns Pointer to a char array containing the last line decoded. The pointer is to a static location, overwritten by subsequent parsing of the databases.

dbDecodeRole

C Synopsis
```
#include "event.h"
#include "dateDb.h"

char *dbDecodeRole( dbRoleType role );
```

Description Returns a description string for the given role (or an error message if the input parameter is not correct).

Returns Pointer to a read-only char array.

dbDecodeBankPattern

C Synopsis
```
#include "event.h"
#include "dateDb.h"

char *dbDecodeBankPattern( dbBankPatternType bankPattern );
```

Description Returns a description string for the given bank pattern (or an error message if the input parameter is not correct).

Returns Pointer to a read-only char array.

dbRolesDb
dbSizeRolesDb
dbMaxLdcId
dbMaxGdcId
dbMaxTriggerMaskId
dbMaxDetectorId
dbMaxSubdetectorId
dbMaxHltProxyId
dbMaxHltProducerId
dbMaxHltRootId
dbMaxTriggerHostId
dbMaxEdmHostId
dbMaxDdgId
dbMaxFilterId
dbLoadRoles
dbUnloadRoles

C Synopsis

```
#include "event.h"
#include "dateDb.h"

dbRoleDescriptor *dbRolesDb
int dbSizeRolesDb;
dbIdType dbMaxLdcId;
dbIdType dbMaxGdcId;
dbIdType dbMaxTriggerMaskId;
dbIdType dbMaxDetectorId;
dbIdType dbMaxSubdetectorId;
dbIdType dbMaxHltProxyId;
dbIdType dbMaxHltProducerId;
dbIdType dbMaxHltRootId;
dbIdType dbMaxTriggerHostId;
dbIdType dbMaxEdmHostId;
dbIdType dbMaxDdgId;
dbIdType dbMaxFilterId;

int dbLoadRoles();
int dbUnloadRoles();
```

Description

The **roles** database is fully described by the above entities that can be loaded using the ***dbLoadRoles*** call.

All the **ID**s described by the database are limited by the values given in ***dbMax*Id***.

The entry ***dbLoadRoles*** loads all the above entities when called the first time. Successive calls to ***dbLoadRoles*** do not force a reload of the entries. This can be achieved by using the ***dbUnloadRoles*** call followed by ***dbLoadRoles***.

On failure to load the database, the values given in the associated variables are ***undefined***.

Returns ***DB_LOAD_OK*** / ***DB_UNLOAD_OK*** in case of success, otherwise an error code.

dbRolesFind
dbRolesFindNext

C Synopsis

```
#include "event.h"
#include "dateDb.h"

int dbRolesFind( char *roleName, dbRoleType role );
int dbRolesFindNext();
```

Description These routines implement a fast-find algorithm on the **roles** database. The ***dbRolesFind*** entry initializes a find for the given entity with the given role (***dbRoleUnknown*** will search for any entity with the given name). The search can be continued starting from the point of the last match using the ***dbRolesFindNext*** entry.

For semantically-correct databases, whenever the role is different from ***dbRoleUnknown***, at most one match should be returned and therefore ***dbRolesFindNext*** should never return a match.

Returns -1 for no match, index to **dbRolesDb** is a match is found.

dbTriggersDb
dbSizeTriggersDb
dbLoadTriggers
dbUnloadTriggers

C Synopsis

```
#include "event.h"
#include "dateDb.h"

dbTriggerDescriptor *dbTriggersDb;
int dbSizeTriggersDb;

int dbLoadTriggers();
int dbUnloadTriggers();
```

Description The ***dbTriggersDb*** pointer can be loaded and unloaded using the ***dbLoadTriggers*** and ***dbUnloadTriggers*** routines. Consecutive calls to the ***dbLoadTriggers*** routine do not reload the database. To achieve this, use a ***dbUnloadTriggers*** / ***dbLoadTriggers*** sequence.

Returns ***DB_LOAD_OK*** / ***DB_UNLOAD_OK*** in case of success, otherwise an error code.

dbGetDetectorsInTriggerPattern

C Synopsis

```
#include "event.h"
#include "dateDb.h"

int dbGetDetectorsInTriggerPattern(
  eventTriggerPatternType triggerPat,
  eventDetectorPatternType detectorPat
);
```

Description The ***detectorPat*** is loaded with the detector pattern that corresponds to the given ***triggerPat***. This information is static and needs to be combined with the dynamic run-time mask of active detectors to give the actual run-time pattern.

Returns ***DB_LOAD_OK*** in case of success, otherwise an error code.

dbGetLdcsInTriggerPattern

C Synopsis

```
#include "event.h"
#include "dateDb.h"

int dbGetLdcsInTriggerPattern(
  eventTriggerPatternType triggerPat,
  dbLdcPatternType ldcPat
);
```

Description The ***ldcPat*** is loaded with the LDC pattern that corresponds to the given ***triggerPat***. This information is static and needs to be combined with the dynamic run-time mask of active LDCs to give the actual run-time pattern.

Returns ***DB_LOAD_OK*** in case of success, otherwise an error code.

dbDetectorsDb
dbSizeDetectorsDb
dbLoadDetectors
dbUnloadDetectors

C Synopsis

```
#include "event.h"
#include "dateDb.h"

dbDetectorDescriptor *dbDetectorsDb;
int dbSizeDetectorsDb;

int dbLoadDetectors();
int dbUnloadDetectors();
```

Description The ***dbDetectorsDb*** pointer can be loaded and unloaded using the ***dbLoadDetectors*** and ***dbUnloadDetectors*** routines. Consecutive calls to the ***dbLoadDetectors*** routine do not reload the database. To achieve this, use a ***dbUnloadDetectors***/***dbLoadDetectors*** sequence.

Returns ***DB_LOAD_OK***/***DB_UNLOAD_OK*** in case of success, otherwise an error code.

dbGetLdcsInDetector

C Synopsis

```
#include "event.h"
#include "dateDb.h"

int dbGetLdcsInDetector(
  dbIdType detectorId,
  dbLdcPatternType ldcPat
);
```

Description The ***ldcPat*** is loaded with the LDC pattern that corresponds to the given ***detectorId***. This information is static and needs to be combined with the dynamic run-time mask of active LDCs to give the actual run-time pattern.

Returns ***DB_LOAD_OK*** in case of success, otherwise an error code.

dbGetLdcsInDetectorPattern

C Synopsis

```
#include "event.h"
#include "dateDb.h"

int dbGetLdcsInDetectorPattern(
  eventDetectorPatternType detectorPat,
  dbLdcPatternType ldcPat
);
```

Description The ***ldcPat*** is loaded with the LDC pattern that corresponds to the given ***detectorPat***. This information is static and needs to be combined with the dynamic run-time mask of active LDCs to give the actual run-time pattern.

Returns ***DB_LOAD_OK*** in case of success, otherwise an error code.

dbEventBuildingControlDb
dbSizeEventBuildingControlDb
dbLoadEventBuildingControl
dbUnloadEventBuildingControl

C Synopsis

```
#include "event.h"
#include "dateDb.h"

eventBuildingRule *dbEventBuildingControlDb;
int dbSizeEventBuildingControlDb;

int dbLoadEventBuildingControl();
int dbUnloadEventBuildingControl();
```

Description The ***dbEventBuildingControlDb*** pointer can be loaded and unloaded using the ***dbLoadEventBuildingControl*** and ***dbUnloadEventBuildingControl*** routines. Consecutive calls to the ***dbLoadEventBuildingControl*** routine do not reload the database. To achieve this, use a ***dbUnloadEventBuildingControl*** / ***dbLoadEventBuildingControl*** sequence.

Returns ***DB_LOAD_OK*** / ***DB_UNLOAD_OK*** in case of success, otherwise an error code.

dbBanksDb
dbSizeBanksDb
dbLoadBanks
dbUnloadBanks

C Synopsis

```
#include "event.h"
#include "dateDb.h"

dbBanksDescriptor *dbBanksDb;
int dbSizeBanksDb;

int dbLoadBanks();
int dbUnloadBanks();
```

Description The ***dbBanksDb*** pointer can be loaded and unloaded using the ***dbLoadBanks*** and ***dbUnloadBanks*** routines. Consecutive calls to the ***dbLoadBanks*** routine do not reload the database. To achieve this, use a ***dbUnloadBanks*** / ***dbLoadBanks*** sequence.

Returns ***DB_LOAD_OK*** / ***DB_UNLOAD_OK*** in case of success, otherwise an error code.

dbUnloadAll

C Synopsis

```
#include "event.h"
#include "dateDb.h"

int dbUnloadAll();
```

Description Unload all the static databases from the memory of the calling process. If the operation fails, the final result is ***unpredictable***.

Returns ***DB_UNLOAD_OK*** if the operation succeeds, error from individual unload routines otherwise.

The monitoring package

5

This chapter describes how to write a monitoring program. After a brief introduction to the monitoring in DATE, the monitoring library is explained and its use from all the most commonly used programming languages is shown.

5.1 Monitoring in DATE

A data-acquisition system requires monitoring of experimental data (online and offline data, on online and offline hosts). Some possible applications for monitoring tasks are:

- statistical analysis of the experimental stream to evaluate the quality of the physics conditions.
- detailed analysis of the experimental data.
- occasional checking of the overall status of the data-acquisition system (e.g. operator status panel).

To perform these and other functions, DATE provides the monitoring package, whose objective is to offer a uniform interface for the development and the support of user-written monitoring programs tailored to specific needs. The monitoring interface allows access to events coming from the live experimental stream or from a ***Permanent Data Storage*** (PDS)[1] media, with statistical or strict monitoring purposes, on online (part of the data-acquisition system) or offline (totally detached) hosts.

When monitoring is performed in its full online configuration (see Figure 5.1 top diagram), the monitoring program gets the data from a local monitoring buffer, filled from the online data producer (the ***readout*** process on LDCs and the ***eventBuilder*** process on GDCs). This approach is the most efficient for what concerns the use of system resources but might impose an unacceptable load on the online host, already charged with acquisition and control tasks.

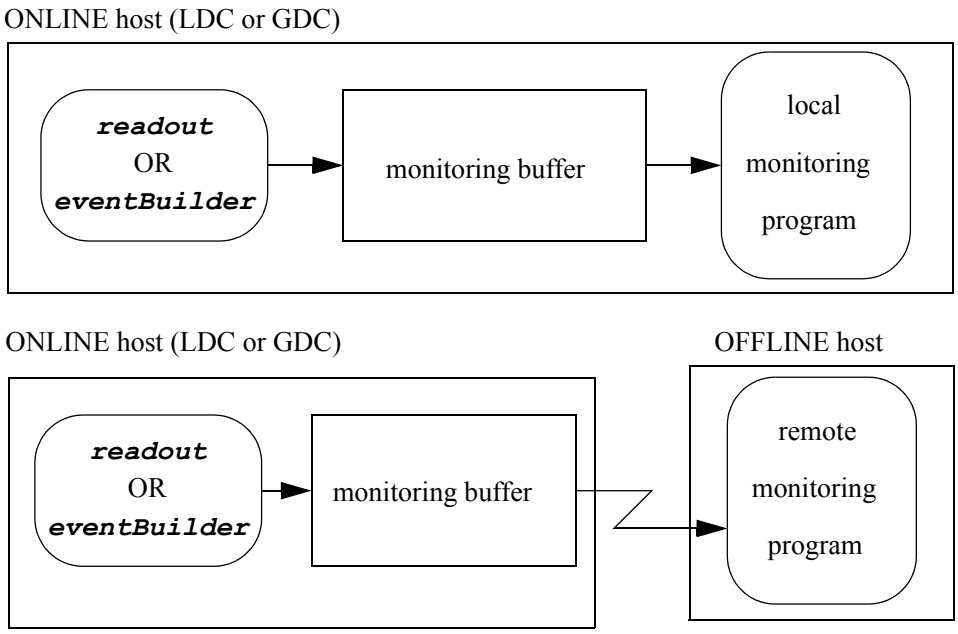

Figure 5.1 The DATE online monitoring, local and remote configurations

1. The term PDS - defined in the ALICE technical proposal [14]- is used here with a wider meaning, also covering permanent, semi-permanent and temporary storage, usually located in the physical path between the data-acquisition system online buffer and the final PDS.

To “off-load” the online environment, it is possible to run the monitoring program on another host, linked to the first via LAN or WAN (see Figure 5.1 bottom diagram). The result is similar to what we achieved in the first configuration, with the advantage of freeing resources on the data-acquisition host, at the price of an increased load on the interconnecting network between the two machines. The same data-acquisition system can have - without reconfiguration - several local and remote monitoring programs, all running simultaneously and getting their data from the same source. However, each monitoring program can receive its data to monitor from one source at a time. It is possible to switch forth and back between different data sources within the same monitoring program, although this practice is not recommended.

The need of monitoring only sub-events coming from selected detector(s) exists in ALICE. A special function has therefore been added to the monitoring library: monitoring by detector. This function extends the remote monitoring scheme, applying it to a set of LDCs, the active hosts attached to one or more detectors. The monitoring library gets the sub-events from all the LDCs, performs a “reduced” event building procedure and delivers the result to the monitoring program. Only events where all the selected LDCs contribute with one sub-event will be selected.

Another operating mode of the monitoring library - shown in Figure 5.2 - allows the same functions on offline streams, usually coming from the Permanent Data Storage. This setup allows direct monitoring from the PDS server or from other hosts (batch server, desktop or workstation) not connected to the PDS media. This configuration can optionally make use of the SHIFT / CASTOR disks servers available at CERN.

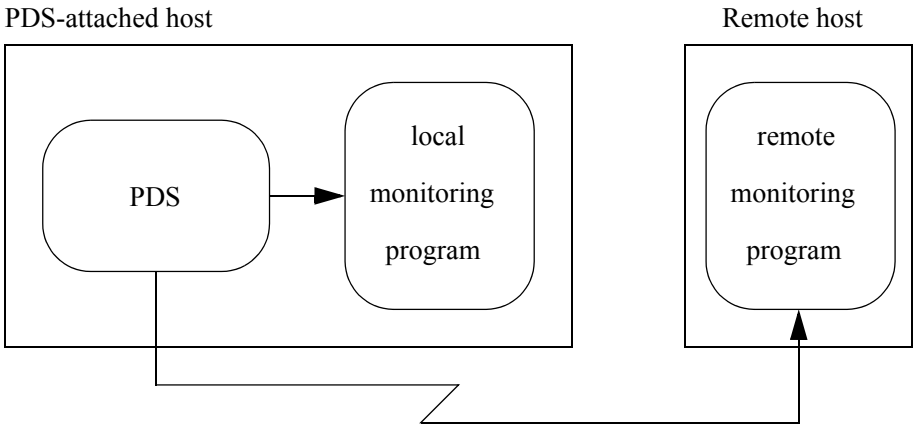

Figure 5.2 The DATE offline monitoring

During the connection phase, monitoring programs can declare themselves to the monitoring scheme. This allows easy tracing of each client and makes it possible to “fine tune” the runtime parameters of the monitoring system.

When a monitoring program connects itself to the experimental stream, it has the capability to declare a monitoring policy for any given event type. This policy can require all events for monitoring (***must*** policy), as many as possible of the events (***most*** policy) a random share of events for monitoring (***yes*** policy), whatever events are available (***few*** policy) or no monitoring at all (***none*** policy). It is important to understand the impact of a given monitoring policy on the data-acquisition system and on the monitoring environment. A monitoring program requesting a “must”

policy must process the information as fast as it will be offered or it might stall the entire data-acquisition stream. On the other hand, the exclusion of certain classes of events - unwanted for a given type of monitoring - will reduce the overhead on the online host and on the interconnecting network, as less data will be stored and transferred between the online producer (***readout*** or ***eventBuilder***) and the consumer (the monitoring program).

Monitoring programs have the choice to stall if no data is available or to continue with their execution (knowing that no data has been received). This allows the implementation of event-driven processes (such as X11 clients) that should not be blocked in absence of data.

Another feature of the monitoring library is to let a monitoring program discard all data eventually stored in the monitoring buffer. This is useful to access only future events at any given point in time.

Some experimental setups might “hide” their data-acquisition hosts behind routers or firewalls, making remote monitoring difficult or impossible. To solve this problem, the DATE monitoring library allows a mechanism called “relayed monitoring”, where the monitoring channel travels through a dedicated relay host (visible from the offline host and with access to the hidden online host). The scheme is described in Figure 5.3. It is possible to filter the access through the relay host only to a restricted set of clients, according to the type of monitoring requested. Relayed monitoring performs worse than direct monitoring and should be used only whenever absolutely unavoidable.

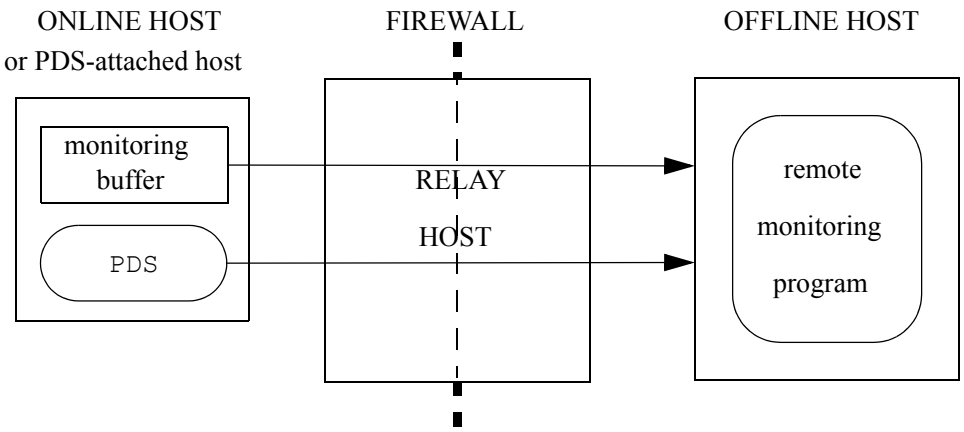

Figure 5.3 The DATE relayed monitoring

5.2 Online monitoring and role name

DATE allows any given host (LDC or GDC) to operate within multiple independent setups (e.g. one setup for production and a second setup for commissioning), also simultaneously. To identify the environment of each setup, DATE assigns to the same host different role names, one for each setup. The monitoring library uses the same mechanism in order to monitor a setup when a choice between multiple data streams is available.

The recommended way to select the appropriate setup is to use role names rather than host names during the declaration of the data source (see the description of the `monitorSetDataSource` routine). When doing so, the monitoring library automatically sets the environment in order to access the appropriate data stream. This mechanism requires at runtime an active and valid connection to the DATE configuration database in order to resolve the role name and the associated host name. This is also the mechanism recommended for local monitoring of a machine that belongs to multiple setups (the monitoring library sets the environment according to the selected role and then proceeds with the same path as for local online monitoring, therefore not using TCP/IP to move the events).

If it is not possible to specify a role name (e.g. for monitoring clients that do not have a connection to the DATE configuration database) it is still possible to select any setup by defining the environment variable DATE_ROLE_NAME before connecting to the remote host (in this case the host name must be used as data source).

If no role name is selected at runtime, the monitoring library chooses the first alphabetical match on the target host name (or for the local host in case of local monitoring). For single-setup environments this solution chooses the only available data stream and therefore always give the expected result. For machines running multiple instances of DATE, an arbitrary selection is implied and this may lead to unexpected behaviors at runtime (according to the content of the DATE configuration database). For this reason we strongly recommend to use the role name whenever this is possible.

To summarize:

1. if the monitoring program has access to the DATE configuration database, always use the role name as data source (also for local monitoring): this procedure gives the maximum flexibility, is fully reconfigurable via the DATE database and always connect to the right data source regardless of the HW/SW configuration in use with no runtime overhead.
2. If the machine being monitored plays a single role, it is still possible to use the anonymous syntax ":" for local monitoring or the TCP/IP hostname for remote monitoring. This scheme is less flexible, is not recommended but still works.
3. If the monitoring program has no access to the DATE configuration database, then it is not possible to use the role name to connect to the (obviously remote) data source. In this case, the data source must contain the TCP/IP hostname and the DATE_ROLE_NAME should be given as environment variable within the monitoring process.

5.3 Monitoring and Analysis in C/C++

A monitoring program should accomplish the following steps in order to perform its function:

1. declare the source providing the data to monitor.
2. declare itself to the monitoring scheme.

3. declare - if necessary - the monitor policies to be followed.
4. declare - if necessary - the wait / nowait policy to be followed.
5. get the available event(s) from the monitoring stream.

The monitoring scheme can be used from programs written in C or C++.

This chapter describes the C / C++ callable interface available within the DATE monitoring package and its characteristics.

5.3.1 Some simple examples

In Listing 5.1 we have a very simple example of a monitoring program written in C.

Listing 5.1 Example of event dump in C:

```
 1: #include <stdio.h>
 2: #include <stdlib.h>
 3: #include "event.h"
 4: #include "monitor.h"
 5:
 6: void printError( char *where, int errorCode ) {
 7:   fprintf( stderr,
 8:            "Error in %s: %s\n",
 9:            where, monitorDecodeError( errorCode ) );
10:   exit( 1 );
11: } /* End of printError */
12:
13: int main() {
14:   int status;
15:
16:   status = monitorSetDataSource( ":" );
17:   if ( status != 0 )
18:     printError( "monitorSetDataSource", status );
19:   status = monitorDeclareMp( "C demo mp" );
20:   if ( status != 0 )
21:     printError( "monitorDeclareMp", status );
22:   for (;;){ /* Start of endless loop */
23:     void *ptr;
24:     struct eventStruct *event;
25:
26:     status = monitorGetEventDynamic( &ptr );
27:     if ( status != 0 )
28:       printError( "monitorGetEventDynamic", status );
29:     event = (struct eventStruct *)ptr;
30:     printf( "Run #:%d, EventId #:%08x%08x, Type:%ld, size:%ld,
    Data size:%d\n",
31:             event->eventHeader.eventRunNb,
32:             event->eventHeader.eventId[0],
33:             event->eventHeader.eventId[1],
34:             event->eventHeader.eventType,
35:             event->eventHeader.eventSize,
36:             event->eventHeader.eventSize -
    event->eventHeader.eventHeadSize );
37:     free( ptr );
38:   } /* End of endless loop */
39: } /* End of main */
```

The program consists of a declaration phase followed by an endless loop where events are fetched from the monitoring stream and their header is printed. Please note that the program never terminates: the process must be killed via an external signal (e.g. `^C` - obtained pressing the "`control`" and the "`C`" keys - via the keyboard for interactive processes).

Looking at the example more in details, we can observe the following features:

Line 3: inclusion of the DATE event declaration module.

Line 4: inclusion of the DATE monitoring declaration module.

Line 16: declaration of the source of monitoring data (in this case, the online local host).

Line 19: declaration of the monitoring program.

Line 26: the next available event is transferred from the monitoring buffer.

Other examples are available in the directory ***${DATE_MONITOR_DIR}***, namely the source code for the `eventDump` utility (described in Section 5.6), named ***eventDump.c***.

5.3.2 The monitoring package files

The distribution point for the monitoring package is ***${DATE_MONITOR_DIR}*** (defined by the DATE setup procedure). In this area it is possible to find the following files:

- ***${DATE_MONITOR_DIR}/monitor.h***: prototypes and definitions for monitoring programs written in C.
- ***${DATE_MONITOR_DIR}/${DATE_SYS}/libmonitor.a***: monitoring library for any language capable of calling C code (e.g. C and C++).
- ***${DATE_MONITOR_DIR}/${DATE_SYS}/libmonitorstdalone.a***: monitoring library with reduced functionality for non-SHIFT hosts (see below).
- ***${DATE_MONITOR_DIR}/${DATE_SYS}/libmonitor.so***: non-SHIFT shareable monitoring library.

C monitoring programs should include the prototypes declaration monitor.h either including in the C compilation statement the output of the command `date-config --cflags` (on machines running DATE or with DATE installed) or copying the prototypes declaration locally (non-DATE machines) and providing the appropriate C compilation directives (specifications vary from architecture to architecture).

Monitoring programs require the libraries specified by the output of the command `date-config --monitorlibs` (with SHIFT access) or `date-config --monitorlibs=noshift` (without SHIFT access).

The ***SHIFT*** library - referenced by the monitoring I/O package - is used to access hosts whose PDS is available on SHIFT servers, e.g. the CERN ALICE WorkGroup server (LXPLUS), the CERN batch processing facility (SHIFT) and the CERN CASTOR servers. If access to any of these facilities is not required, the inclusion of ***SHIFT*** libraries is not necessary. The output image can therefore be used for local file I/O or for remote network monitoring. These libraries are distributed by the CASTOR support team at CERN.

Hardware and software platforms not part of the standard DATE distribution - but possible clients of the DATE monitoring scheme - can still use the monitoring library by copying the necessary files and performing local compilation and link.

5.3.3 Error codes

The entries belonging to the monitoring library may return a monitoring-specific error code. This code can be either zero for success or non-zero for failure. To decode an error code please refer to the ***${DATE_MONITOR_DIR}/monitor.h*** file or call the entry monitorDecodeError described in the next section.

5.3.4 The monitoring callable library

This section describes the entries available in the monitoring library. Each entry is described in the C version. For the decoding of error codes eventually returned by the entries, please refer to Section 5.3.3.

---

monitorSetDataSource

C Synopsis

```
#include "monitor.h"

int monitorSetDataSource( char* source )
```

Description The source of events to monitor is declared. The syntax of the monitor source parameter is given in Table 5.1.

Table 5.1 Monitor source parameter syntax

":"	local online (default)
"file"	local file (both full and relative paths are accepted, full path recommended)
"@target:"	remote online on machine "target"
"file@target"	remote file on machine "target" (the full path to the file should be given)
"@target1@target2:"	remote online on machine "target1" via the relay host "target2"
"file@target1@target2"	remote file on machine "target1" via the relay host "target2" (the full path to the file should be given)
"^det[+det]"	remote online on the LDCs belonging to the detector "det" (plus-separated lists can be used to specify more than one detector) and active in the current run
"@*:"	remote online on the GDCs active in the current run
"=partition"	remote online on the GDCs active in the partition

The parameter "target" can specify either a role name (recommended) or a TCP/IP host name (see Section 5.2). If remote monitoring is used and "target" points to the local host, then local monitoring is assumed and no transfer take place over TCP/IP (not even via local sockets). The monitoring library is able to resolve host aliasing and multi-interface hosts.

For detector and GDCs monitoring, the run number must be specified in the environment variable DATE_RUN_NUMBER. The run number can also be re-defined during the monitoring, but in this case a `monitorLogout` followed by a new call to `monitorSetDataSource` is recommended.

For partition monitoring, the run number is not needed. The monitoring library will reconfigure at each start of run adding or removing GDCs according to the configuration of the partition itself. Monitoring programs must take care in handling the events when these come from consecutive runs.

Returns Zero in case of success, else an error code (see Section 5.3.3 for more details).

monitorDeclareMp

C Synopsis

```
#include "monitor.h"

int monitorDeclareMp( char* mpName )
```

Description The given string is used to declare the monitoring program. This can be used for debugging, for fine tuning and to monitor the online monitoring scheme (see Section 5.7).

Returns Zero in case of success, else an error code (see Section 5.3.3 for more details).

monitorDeclareTable

C Synopsis

```
#include "monitor.h"

int monitorDeclareTable( char** table )
```

Description A table describing the desired monitoring policy is declared within the monitoring scheme. Each monitoring program can declare a monitoring table at any time. This table will be used for all subsequent calls to monitorSetDataSource and will be kept valid in case monitorLogout is called. It is possible to declare a table in the middle of a monitoring stream: this will force a flush of all events eventually available in the monitoring buffer and in the monitoring channel.

The input parameter should have the following C syntax:

```
char *table[] = {
  [ "event type", "monitor type", ]*
  NULL
};
```

where the fields `event type` and `monitor type` can assume one of the values and aliases given in Table 5.2 and in Table 5.3.

Table 5.2 Event types

Event type	Single-word alias	Short alias
"All events"	"All_events"	"ALL"
"Start of run"	"Start_of_run"	"SOR"
"Start of run files"	"Start_of_run_files"	"SORF"
"Start of data"	"Start_of_data"	"SOD"
"Start of burst"	"Start_of_burst"	"SOB"
"End of burst"	"End_of_burst"	"EOB"
"Physics event"	"Physics_event"	"PHY"
"Calibration event"	"Calibration_event"	"CAL"
"System Software Trigger event"	"System_software_ trigger_event"	"SST"
"Detector Software Trigger event"	"Detector_software_ trigger_event"	"DST"
"End of data"	"End_of_data"	"EOD"
"End of run"	"End_of_run"	"EOR"
"End of run files"	"End_of_run_files"	"EORF"
"Event format error"	"Event_format_error"	"FERR"

Table 5.3 Monitor types

Monitor type	Action
"all"	all events of this type are monitored (100%)
"most"	a priority sample of the events of this type is monitored
"yes"	a sample of the events of this type is monitored
"few"	events of this type may be monitored
"no"	no events of this type are monitored

All declarations are case-insensitive and can be shortened to the nearest unique string (watch out for ambiguous shortening, e.g. "end of run" can match either "end of run" or "end of run files").

The features of the various sampling modes are:

– **all**: all the events that match the selection are stored in the monitoring buffer. This mode ***must be used with extreme care*** as the DAQ stops if the monitoring program(s) cannot keep up with the throughput of the data flow.

– **most**: as long as there is buffer space, the events that match the selection are copied in the monitoring buffer. These events may be dropped to make space to "**all**" events. If the monitoring program cannot keep up with the data flow, the overflowing events are dropped. This monitoring mode should be used only to

select rare events, not to disrupt the distribution of events received by other monitoring programs. Although delivery of events is not guaranteed (they may be dropped to make space to "**all**" events and they may not get recorded if the monitoring buffer contains only "**most**" events), it should have a much higher delivery success compared to a "**yes**" policy in case of multiple monitoring programs running with high input event rates.

- **yes**: as long as there is buffer space, the events that match the selection are stored in the monitoring buffer. These events may be dropped if "**all**" or "**most**" events need space to be stored and may drop "**few**" events if they cannot be stored due to lack of space in the monitoring buffer.
- **few**: events matching the selection are monitored as long as they can be stored and may be removed in case events matching other monitoring type criteria need space in the monitoring buffer. This policy may be useful for very slow monitoring programs such as event displays.
- **no**: the events of the given type are not published for monitoring.

Please note that for the case of one monitoring program active, "**most**", "**yes**" and "**few**" will yield the same result.

For setups with multiple monitoring client, events monitored as "**no**" by one client may still be stored in the monitored buffer if other monitoring program(s) requested them.

The default table is:

```
char *defaultTable[] = {
  "All", "yes",
  NULL
};
```

Returns Zero in case of success, otherwise an error code (see Section 5.3.3 for more details).

monitorDeclareTableExtended

C Synopsis

```
#include "monitor.h"

int monitorDeclareTableExtended( char** table )
```

Description The purpose of this entry is to declare a monitoring policy where event attributes and / or trigger classes can be used for the selection method. Functionally, this routine is equivalent to monitorDeclareTable. The input parameter has the following syntax:

```
char *table[] = {
 [ "event type", "monitor type", "attributes", "triggers", ]*
  NULL
};
```

The event type and monitor type fields have the same syntax as for `monitorDeclareTable` (see Table 5.3 and Table 5.2).

The attributes and the triggers fields may contain either one number, a list of numbers separated by "|" (any of the given patterns would select the event) or by "&" (all the bits of the pattern must be asserted to select the event). For example:

- "2" selects events with bit 2 asserted.
- "2|3" selects events with bit 2 or with bit 3 asserted.
- "2&3" selects events with bits 2 and 3 asserted.

It is not possible to mix "|"s and "&"s in the same declaration, e.g. "2&3|4" returns a runtime error.

Empty lists or wildcard "*" lists can be specified to disable the selection criteria. For example:

- "PHY" "Y" "1" "" selects physics events with attribute 1 asserted, regardless of the status of their trigger pattern.
- "PHY" "Y" "*" "1" selects physics events with trigger pattern 1 asserted.
- "PHY" "Y" "1" "2" selects physics events with attribute 1 and trigger pattern 2 asserted.
- "PHY" "Y" selects physics events (same as "PHY" "Y" "*" "*").

Both system and user attributes can be specified: for user attributes, use the attribute number (as used in the *_USER_ATTRIBUTE() macro). System attributes should be specified via the corresponding symbol from the ***${DATE_COMMON_DEFS}/event.h*** definition file.

If a non-empty trigger pattern is declared, events whose trigger pattern has not been validated are NOT selected for monitoring. If the trigger pattern is not specified, all events are potentially selected regardless of the validation status of their trigger pattern.

Returns Zero in case of success, otherwise an error code (see Section 5.3.3 for more details).

monitorDeclareTableWithAttributes

C Synopsis

```
#include "monitor.h"

int monitorDeclareTableWithAttributes( char** table )
```

Description The purpose of this entry is to declare a monitoring policy where event attributes can be used for the selection method. Functionally, this routine is equivalent to monitorDeclareTableExtended.

This entry is deprecated and is left for backward compatibility only. The entry monitorDeclareTableExtended should be used instead (with the triggers.field left empty).

The input parameter has the following syntax:

```
char *table[] = {
  [ "event type", "monitor type", "attributes", ]*
```

```
  NULL
};
```

For the description of the `attributes` parameter, see the description of `monitorDeclareTableExtended`.

Returns Zero in case of success, otherwise an error code (see Section 5.3.3 for more details).

monitorGetEvent

C Synopsis

```
#include "monitor.h"

int monitorGetEvent( void *buffer, long size )
```

Description The next available event (if any) is copied in the region pointed by `buffer` for a maximum length of `size` bytes. In case of failure, a zero-length event is returned.

Returns Zero in case of success, otherwise an error code (see Section 5.3.3 for more details).

monitorGetEventDynamic

C Synopsis

```
#include "monitor.h"

int monitorGetEventDynamic( void **buffer )
```

Description The next available event (if any) is copied on space reserved from the process heap and returned to the caller. The caller ***must take care*** of properly disposing the event via the `free` system call: failure to do so will exhaust the resources associated to the process and can severely degrade the overall system performances. If no data is available and the channel set in ***`noWait`*** mode the pointer returned will be ***`NULL`***; in this case the event does not need to be disposed.

Returns Zero in case of success (also if no event is available), otherwise an error code (see Section 5.3.3 for more details).

monitorFlushEvents

C Synopsis

```
#include "monitor.h"

int monitorFlushEvents( void )
```

Description All the data available in the monitoring buffer is discarded. The next event transferred over the monitoring channel will be injected in the monitoring stream after this call terminates.

Returns Zero in case of success, otherwise an error code (see Section 5.3.3 for more details).

monitorSetWait

C Synopsis

```
#include "monitor.h"

int monitorSetWait( void )
```

Description After this call completes, if the monitoring program requests an event when the monitoring buffer and the monitoring channel are empty, the monitoring program will stop and wait for new events. This is the default behaviour of the monitoring library.

Returns Zero in case of success, otherwise an error code (see Section 5.3.3 for more details).

monitorSetNowait

C Synopsis

```
#include "monitor.h"

int monitorSetNowait( void )
```

Description After this call completes, if the monitoring program requests an event when the monitoring buffer and the monitoring channel are empty, the monitoring program will continue and an empty event will be returned.

Returns Zero in case of success, otherwise an error code (see Section 5.3.3 for more details).

monitorControlWait

C Synopsis

```
#include "monitor.h"

int monitorControlWait( int flag )
```

Description The wait / nowait behavior of the monitoring library is set accordingly to the `flag` parameter:

- true (wait): ***TRUE*** `or (0 == 0)`
- false (nowait): ***FALSE*** `or (0 == 1)`

Returns Zero in case of success, otherwise an error code (see Section 5.3.3 for more details).

monitorSetNoWaitNetworkTimeout

C Synopsis

```
#include "monitor.h"

int monitorSetNoWaitNetworkTimeout( int timeout )
```

Description Set the timeout for nowait reads of events via the network. When the `timeout` parameter is negative or zero, nowait reads of events through the network return immediately. When the `timeout` parameter is > 0, reads of events through the network may wait - if no events are available - up to the given time expressed in milliseconds. The default value of the timeout is -1 (no timeout). This call does not apply to the "monitoring by detector" scheme.

Returns Zero in case of success, otherwise an error code (see Section 5.3.3 for more details).

monitorSetSwap

C Synopsis

```
#include "monitor.h"

int monitorSetSwap( int 32BitWords, int 16BitWords )
```

Description This entry controls the behavior of the monitoring library when a network channel is opened with a host of different endianness (e.g. ***PC***/***ALPHA*** vs. ***Motorola***/***IBM***/***Sun***). The two parameters are used to control the swapping algorithm to be used for the data portion of the incoming events; their possible use depends on the actual content of the data (payload) portion of the event and can be summarized as in Table 5.4.

Table 5.4 Bytes swapping control

Data buffer data type	32BitWords	16BitWords
8-bit entities (signed or unsigned characters)	***FALSE***	***FALSE***
32-bit entities (e.g. ***VMEbus*** data)	***TRUE***	***FALSE***
16-bit entities	***FALSE***	***TRUE***

In case swapping is not known beforehand, monitoring programs should set the two flags to FALSE and swap the data manually once their type is known: this will avoid unnecessary double-swapping at run-time.

The values that can be given to the two flags are:

- true (perform swapping): ***TRUE*** `or (0 == 0)`
- false (do not swap): ***FALSE*** `or (0 == 1)`

Returns Zero in case of success, otherwise an error code (see Section 5.3.3 for more details).

monitorDecodeError

C Synopsis

```
#include "monitor.h"

char *monitorDecodeError( int code )
```

Description The entry returns the pointer to a string describing the given error code.

Returns Pointer to a zero-terminated static, read-only string.

monitorLogout

C Synopsis

```
#include "monitor.h"

int monitorLogout( void )
```

Description The monitoring link is closed and all resources allocated for this monitoring program are freed. The link will be automatically re-opened when the monitoring program requests the next event. This entry can be used whenever the monitoring program expects long pauses, such as operator input. It imposes a certain overhead on the monitoring scheme and therefore should not be used too frequently.

Returns Zero in case of success, otherwise an error code (see Section 5.3.3 for more details).

5.4 Monitoring by detector

Monitoring by detector is the mechanism where events are received from all the active LDCs belonging to a particular detector (or a set of detectors). This is done by opening a set of TCP/IP channels and handling them individually. Events are received on a channel-by-channel basis and a reduced event building procedure is applied. All the LDCs active within the given detectors set must give one sub-event in order to perform a successful event building. Whenever one or more LDCs fail to provide a sub-event for a given event, this will be discarded.

The output of a "monitoring by detector" procedure is a event with the following characteristics:

- ***Event ID***, ***event type***, ***version ID*** and ***run number*** as for the input events.
- ***Trigger pattern***: picked up at random from any of the sub-events.
- ***Detector pattern*** set according to the monitoring data source.
- ***LDC ID*** and ***GDC ID*** set to ***VOID_ID***.
- ***ATTR_SUPER_EVENT*** and ***ATTR_BY_DETECTOR_EVENT*** set.
- ***Time*** set to the moment the event has been delivered to the monitoring

program.

Almost all the calls of the monitoring library are supported by the “monitoring by detector” scheme. Please note that, as the library must handle a set of monitoring channels, if a runtime error is returned by any call the error could come from any of the active channels. More error may occur during such an operation, in which case only the first error is reported. The requested operation is anyway attempted on all the channels, regardless from the status of each individual transactions. If any of the calls returns an error status, the status of the connected channels is unpredictable: a full logout/login procedure is therefore recommended.

5.5 Monitoring from ROOT

5.5.1 The ROOT system

The ROOT [10] system provides a set of Object Oriented frameworks with all the functionality needed to handle and analyze large amounts of data in a very efficient way. As the data is defined as a set of objects, specialized storage methods are used to get direct access to the separate attributes of the selected objects without having to touch the bulk of the data. Included in ROOT are histogramming methods in 1, 2 and 3 dimensions, curve fitting, function evaluation, minimization, graphics and visualization classes to allow the easy setup of an analysis system that can query and process the data interactively or in batch mode.

Thanks to the built-in CINT C++ interpreter the command language, the scripting, or macro, language and the programming language are all C++. The interpreter allows for fast prototyping of the macros since it removes the time consuming compile/link cycle. It also provides a good environment to learn C++. If more performance is needed, the interactively developed macros can be compiled using a C++ compiler.

The system has been designed in such a way that it can query its databases in parallel on MPP machines or on clusters of workstations or high-end PC’s. ROOT is an open system that can be dynamically extended by linking external libraries.

From ROOT, the “standard” C library is loaded inside ROOT and called directly. The features available to the monitoring module are the same offered to any C/C++ program. Special care must be given to all asynchronous events, such as timers, signals and graphics handling.

Monitoring from ROOT can be done either directly - using the libraries and files provided by the basic DATE distribution - or via AMORE (see Chapter 23).

5.5.2 Direct monitoring in ROOT

When the direct monitoring approach is used, the “standard” DATE monitoring library is loaded in the ROOT context and programs run as any other C/C++ applications. In this environment, special care must be given to a proper integration between the monitoring program and the ROOT framework, especially for what concerns handling of signals, timers and graphics events.

The calling sequence for direct monitoring in ROOT is the same as for standard C/C++ programs.

No support is available for ROOT/CINT, only compiled code can be used.

5.6 The "eventDump" utility program

Part of the standard DATE kit is the utility `eventDump`. This program allows easy monitoring of any stream, useful for a quick check or debugging of a running system. The standard DATE kit provides a version of the `eventDump` utility for each architecture fully supported or only with monitoring support.

To run the utility on DATE hosts, execute the standard DATE setup and issue the command

```
> eventDump buffer
```

For non-DATE hosts, copy the utility in your ***${PATH}*** (or declare a proper alias) and then issue the same command as for DATE hosts.

A list of all available options can be shown via the "-?" command-line flag. Some of the parameters are:

- `-b` brief output (does not display event data).
- `-c` check events data against a pre-defined data pattern (test environment only).
- `-s` use static data buffer rather then dynamic memory.
- `-a` use asynchronous reads (nowait mode).
- `-i` interactive: pauses after each event and proposes a mini-menu with several options.
- `-t` allows the declaration of a monitoring table, e.g.:

  ```
  -t "SOR yes EOR y"
  ```

  will show only **`start-of-run`** and **`end-of-run`** events, skipping all events of other types.
- `-T` allows the declaration of an extended monitoring table e.g.:

  ```
  -T "SOR y * * PHY Y 1&3 *"
  ```

  will monitor **`start-of-run`** records and **`physics`** events with attributes 1 and 3 set, skipping all the other events.

The buffer parameter must always be specified. The syntax to be used is the same as for the parameter of the monitorSetDataSource entry (see Table 5.1). The attributes shall be specified according to the SITE-specific conventions (***USER*** attributes) or the central ***${DATE_COMMON_DEFS}/event.h*** definition (e.g. ***ATTR_P_START*** - corresponding to a phase start event - can be monitored specifying the attribute 64).

5.7 Monitoring of the online monitoring scheme

The DATE monitoring environment may influence the performance of the data acquisition system it is connected to, usually reducing its performance. This can be caused either by high-debit monitoring schemes (too many clients and / or too demanding clients) or by "bad" architectural designs. For this main reason, DATE provides a set of tools to monitor the status of the monitoring scheme in a "live" environment. This set includes the following utilities (whose alias is defined by the DATE setup procedure):

- `monitorClients`: live display of the list of all registered clients, eventually with the name of the host they are running on.
- `monitorSpy`: live, highly detailed snapshot of the monitor scheme data structures.

It is not possible - and not logical - to monitor the status of an offline or relayed monitoring host.

For machines belonging to multiple DATE setups, it is mandatory to set the environment variable DATE_ROLE_NAME to its appropriate value prior to run any of the monitoring utilities. If this is not done, the monitoring library arbitrarily chooses the first match in alphabetical order with the host name and uses it, which may lead to incorrect results for multi-role hosts.

5.7.1 The monitorClients utility

The `monitorClients` utility gives a report on the usage of the monitoring scheme local to the host it runs on. Without parameters, it gives the list of monitor programs currently registered and their monitoring policies. If used with the "`-t`" option, it gives a continuous list of the clients and their usage of the monitoring streams (in number of events and number of bytes transferred): the display process can be interrupted using the ^C (obtained pressing the "`control`" and the "`C`" keys) quit signal.

Two examples on the use of the `monitorClients` utility are given in Listing 5.2.

Listing 5.2 Examples of use of the monitorClients utility

```
 1:  > monitorClients
 2:  10 clients max, 1 client declared, 2 processes attached
 3:   PID     SOR   EOR  SORF  EORF   SOB   EOB  PHYS   CAL   EOL
     FERR    Monitoring program:

 4:  53648    yes   yes   yes   yes   yes   yes   yes   yes   yes
     yes    DATE V3 event dump V1.08@host

 5:  > monitorClients -t
 6:  Displaying top clients: ^C to stop
 7:  PID         Bytes/s Mp
 8:  37678        843113 DATE V3 event dump V1.08@host
 9:  PID        Events/s Mp
10:  37678           242 DATE V3 event dump V1.08@host
```

5.7.2 The monitorSpy utility

The `monitorSpy` utility can be used to obtain a snapshot of the entire monitoring scheme in use on the host where the utility is executed. An example of the information that can be obtained with this utility is given in Listing 5.3.

Listing 5.3 Examples of use of the monitorSpy utility

```
 1: > monitorSpy
 2: mbMonitorBufferSize: 0x30000000      13114156 bytes
 3: ------------------------------------------------------------------------
 4: mbMaxClients:        0x30000004            10 clients max
 5: mbMaxEvents:         0x30000008           100 events max
 6: mbEventSize:         0x3000000c        131072 bytes/event
 7: mbEventBufferSize:   0x30000010      13107200 bytes event data
 8: ------------------------------------------------------------------------
 9: mbNumSignIn:         0x30000024             1 clients
10: mbNumSignOut:        0x30000028             0 clients
11: mbForcedExits:       0x3000002c             0 clients
12: mbTooManyClients:    0x30000030             0 clients
13: mbEventsInjected:    0x30000034             0 events
14: mbBytesInjected:     0x30000038             0 bytes
15: ------------------------------------------------------------------------
16: monitorEvents:       0x30001030
17: mbOldestEvent:       0x30000018            -1 (index)
18: mbFirstFreeEvent:    0x3000001c            -1 (index)
19: mbCurrentEventNumber:0x30000020             0 (sequential number)
20: ------------------------------------------------------------------------
21: monitorClients:      0x30000040            10 clients [ 0 .. 9 ]
22: mbNumClients:        0x30000014             1 client(s), 1
    process(es) attached
23:      0@30000040..300001d7: PID:22202, reg#:1, mp:"DATE V3 event
    dump V1.08", last event:0, events:0, bytes:0, WAITING
24:       Monitor policy:
25:          SOR         :monitor -=-
26: [...]
27:          FORMATERROR :monitor -=-
28:      1 .. 9: unused
29: ------------------------------------------------------------------------
30: mbFreeEvents:        0x30001b20, 1 frag, size: 13107188 tot,
    13107188 avg
31:   Free list:
32:     0x30001b2c (13107188, 0x00c7fff4)  0x00000000 0x00000000
    0x00000000 0x00000000
33: monitorEventsData:   0x30001b2c
```

5.8 Monitoring configuration

Functionally, hosts participating in a DATE monitoring scheme can be defined as:

1. monitoring hosts running specific monitoring programs, either part of the standard monitoring package (e.g., the utility eventDump) or written by any DATE user.
2. monitored hosts offering monitoring streams to monitoring programs: these streams can be online streams (from a live data-acquisition system) or offline streams (typically files available from permanent data storage).
3. relaying hosts offering a liaison between monitored and monitoring hosts that cannot establish a direct link due to the presence of network firewalls or

gateways.

It is possible to have any combination of those three functions, e.g. hosts who are monitoring, are monitored and offer relayed monitoring to other hosts.

The DATE monitoring scheme ***needs*** to be configured only for monitored and relaying hosts in the following situations:

1. online monitored hosts (LDCs or GDCs) offering online, offline or relayed monitoring to itself and / or to other hosts.
2. hosts that are part of a DATE system offering offline or relayed monitoring to other hosts.

No setup is required for hosts only wishing to perform monitoring, either on the same or on remote hosts and a complete DATE installation is not required. For the developer of the monitoring program itself, a library is available and can be used in stand-alone mode. Otherwise, monitoring programs can be exported to any type of hosts (within the set of supported architectures) with no need for extra files or special setups. No daemons are necessary and no configuration is required on the monitoring hosts.

We will now review the configuration needed on monitored and relayed hosts to let them perform their function.

5.8.1 Creation of configuration files

The monitoring scheme can be configured using three separate files:

- ***${DATE_SITE_CONFIG}/monitoring.config:*** this file is optional and can be used to control a complete DATE site, all types of hosts.
- ***${DATE_SITE}/${DATE_HOSTNAME}/monitoring.config:*** this file is mandatory for **`online`** hosts and ***must*** be created by the DATE system administrator. It is not required for offline, relayed or monitoring hosts.
- ***/etc/monitoring.config***: this file is optional and can be used to control the behavior of relaying hosts; it is not used by online or offline monitored hosts.

The above files should be created using the following commands:

Listing 5.4 Creation of configuration files

```
1:  > touch file
2:  > chmod u=rw,g=rw,o=r file
```

where "`file`" is the full path of the file to be created. Once created, the configuration files can be edited and parameters can be specified as a list of names followed by their associated values. Comments can be inserted via the "***#***" sign, e.g.:

```
# This is a comment
PARAMETER  VALUE # comment
```

These files can be changed at any time. Some of the parameters (those labelled in Table 5.5 as "Online monitoring only") require the acquisition to be stopped and no

active clients (the command ***monitorClients*** - see Section 5.7.1 - can be used to check for registered clients). All the other parameters can be changed at any time and will become active for all new clients (monitored and monitoring hosts) started after the modification(s).

When the same parameter is defined in multiple files, a “last given” policy is followed, that is:

- parameters defined in ***${DATE_SITE_CONFIG}/monitoring.config*** can be overwritten by equivalent definitions from any of the other files.
- parameters defined in ***${DATE_SITE}/${DATE_HOSTNAME}/monitoring.config*** are final for local monitoring and can be overwritten by equivalent definitions from ***/etc/monitoring.config*** for relayed monitoring.
- parameters defined in ***/etc/monitoring.config*** are final and cannot be overwritten.

Only exception to this scheme is the parameter ***LOGLEVEL***, where the highest given level is used regardless of their definition point (e.g. if the values 0, 20 and 10 are specified, the value used will be 20).

The parameters that can be specified in the configuration files are listed in Table 5.5.

Table 5.5 Monitoring configuration parameters

Parameter name	Used for	Description
LOGLEVEL	All types of monitoring	Level for error, information and debug statements generated by the monitoring scheme
MAX_CLIENTS	Online monitoring only	Maximum number of clients allowed to be registered simultaneously
MAX_EVENTS	Online monitoring only	Maximum number of events available for monitoring
EVENT_SIZE	Online monitoring only	Average event size
EVENT_BUFFER_SIZE	Online monitoring only	Size of buffer used to store events data
EVENTS_MAX_AGE	Online monitoring only	Maximum age (in seconds) of the events available for monitoring
MONITORING_HOSTS	Online monitoring Networked monitoring	Comma-separated list of hosts allowed to perform monitor-when-available from this host
MUST_MONITORING_HOSTS	Online monitoring Networked monitoring	Comma-separated list of hosts allowed to perform all types of monitoring from this host

For the ***MONITORING_HOSTS*** and ***MUST_MONITORING_HOSTS*** parameters, a comma-separated list of hosts should be given, e.g.:

```
MONITORING_HOSTS localhost,pcxy,pcabc01
```

In the above example, the hosts allowed to perform “normal” monitoring are the local host, all hosts whose name begins with pcxy (pcxy01, pcxy02 and so on) plus the host pcabc01.

A host who is defined within the ***MONITORING_HOSTS*** list can only perform monitoring-when-available. To be able to perform 100% monitoring, a host must be in the ***MUST_MONITORING_HOST*** list.

If the parameter ***MUST_MONITORING_HOSTS*** is not specified, all hosts can perform 100% monitoring on the monitored machine. Conversely, if the parameter ***MONITORING_HOSTS*** is not specified, all hosts can perform monitoring functions on the given machine.

If processes on the local host are allowed to perform monitoring, the ***MONITORING_HOSTS*** and ***MUST_MONITORING_HOSTS*** lists ***must*** contain the `localhost` keyword. If the keyword `localhost` is not present, local monitoring ***will not*** be allowed. This keyword is needed as local monitoring “escapes” from the network protocol and is instead performed via memory-mapped direct access.

The readout program

6

This chapter describes the DATE software running on the LDC that manages the data stream. There are two processes in an LDC, namely ***readout*** and ***recorder***. The ***readout*** process waits for a trigger, reads out the front-end electronics, and fills a FIFO with the sub-event data. The ***recorder*** process off-loads this FIFO and sends the sub-event data to a local disk, to a named pipe, or to a GDC over the network. In particular this chapter explains how to build and how to customize a ***readout*** program. By using the generic ***readList*** library, the ***readout*** program is organized as a collection of ***equipments***. Each equipment can be programmed independently, can be selected (activated and deactivated) and parameterized before the run starts without changing the code. All the software is contained in the DATE packages ***readout*** and ***readList***.

6.1 The readout process

The ***readout*** process and the ***recorder*** process are running in all the LDCs participating in the data taking. This chapter is devoted to the ***readout*** process, whereas the ***recorder*** process is covered in Section 10.3.

The ***readout*** process executes the suitable code to perform the front-end electronic readout. This code is specified in a separate software module, called ***readList***, which has to be compiled and linked with the readout main program. The ***readlist*** module consists of the following five routines:

- ***ArmHw***, called at each start of run to perform the initialization;
- ***AsynchRead***, called in the main event loop to perform the readout of the hardware that produces an asynchronous flow of data;
- ***EventArrived***, called in the main event loop to discover whether a trigger has occurred;
- ***ReadEvent***, called in the main event loop after the arrival of a trigger to perform the readout of the hardware;
- ***DisArmHw***, called at each end of run to perform the hardware rundown.

Figure 6.1 shows the structure of the readout program and how these routines are called in the main event loop.

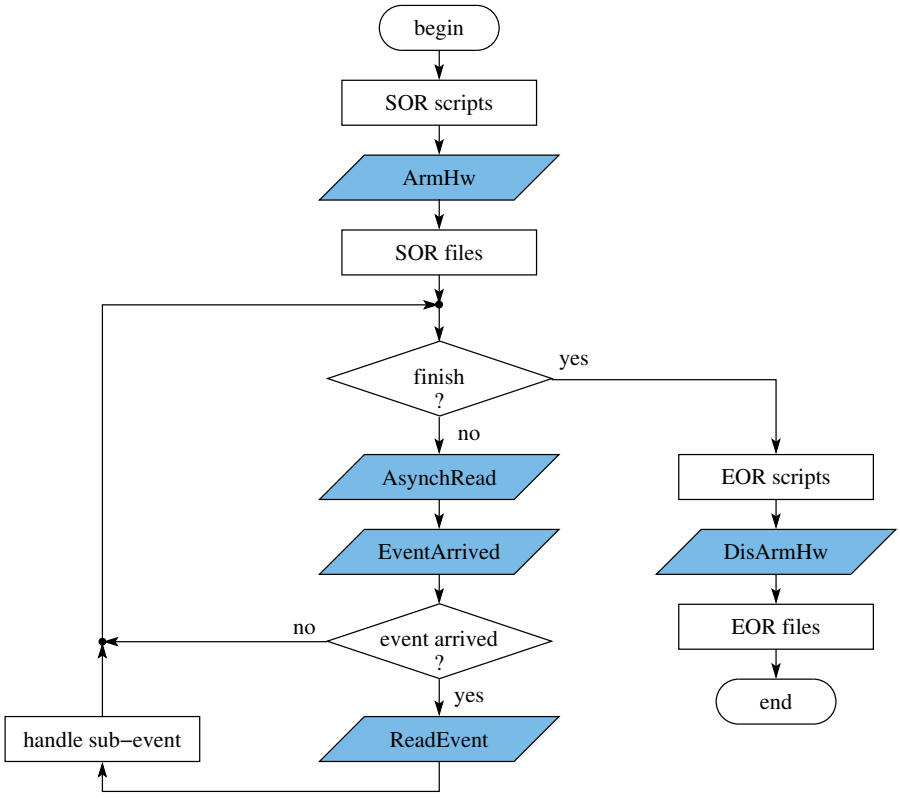

Figure 6.1 Main event loop executed by the readout process.

6.1.1 Start of run phases

At each start of run (SOR), the ***readout*** process performs the following sequence of operations in the order described below:

1. maps to all memory banks that are configured for this LDC in the banks database (see Chapter 4.3.6);
2. executes the common SOR scripts (if any), see "***SOR.commands***" in the ALICE DAQ WIKI;
3. executes the detector SOR scripts (if any);
4. executes the specific SOR scripts (if any);
5. calls the routine ***ArmHw***;
6. produces a SOR record: the ***eventType*** field of the base event header is set to ***START_OF_RUN***, the ***eventTypeAttribute*** field of the base event header is set to ***START_OF_RUN_START*** and the payload is empty;
7. prepares the common SOR files (if any, one record per file);
8. prepares the detector SOR files (if any, one record per file);
9. prepares the specific SOR files (if any, one record per file);
10. produces a SOR record: the ***eventType*** field of the base event header is set to ***START_OF_RUN***, the ***eventTypeAttribute*** field of the base event header is set to ***START_OF_RUN_END*** and the payload is empty.

The SOR sequences have been split into phases, corresponding to the points enumerated above. At each time the timeout (given by the LDC run parameter ***Max. time for SOR/EOR phases***) is restarted to also allow longer initialization or ending phases.

Once the SOR phase has been completed ***readout*** reads from the DATE data base two configuration files:

1. ***readout.config***, to select the GDC selection algorithm, the possibility to dump the event payload during the run and activate the back-pressure monitor, (see "***readout.config***" in the ALICE DAQ WIKI);
2. ***CDH.config*** that contains the readout instructions on how to print the detector specific information stored in the CDH, (see "**CDH.config**" in the ALICE DAQ WIKI).

If these two files don't exist, ***readout*** uses the default configuration, described in the ALICE DAQ WIKI:

- **GDC_SELECTION_ALGORITHM = ORBIT_RAND**
- **READOUT_DUMP_PAYLOD = NO**
- **BCKP_MONITOR = NO**
- **CDH.print = FALSE**

6.1.2 Main event loop

After the start of run phases, the ***readout*** process enters in the main event loop (see Figure 6.1). It allocates memory for one sub-event, which depends on the value of the LDC run parameter ***Paged data flag*** (see Chapter 3):

- for streamlined data (***Paged data flag*** is 0) ***readout*** allocates the following: an entry for an event descriptor in the ***readoutReadyFifo*** and memory from the ***readoutData*** memory bank for the payload, whose size is given by the LDC run parameter ***Max. event size***.;
- for paged data (***Paged data flag*** is 1) ***readout*** allocates the following: an entry for an event descriptor in the ***readoutReadyFifo*** and memory for the first level vector from the ***readoutFirstLevel*** memory bank.

The ***readout*** process calls the routine ***AsynchRead*** to activate the readout of hardware that produces an asynchronous flow of data and then waits for a trigger by calling the routine ***EventArrived***: the arrival of a trigger can signal for example a physics event or a start of burst (SOB) or an end of burst (EOB). If no events are arriving (routine ***EventArrived*** returns 0), the innermost main event loop is executed at maximum speed as long as there is no “end of run” request.

If the LDC run parameter ***startOfData/endOfData event enabled*** is set, the ***readout*** process also exits the main event loop, when the timeout (given by LDC run parameter ***startOfData timeout*** or ***endOfData timeout***) to wait for events of type ***START_OF_RUN*** or ***END_OF_RUN*** (see “**DATA FORMAT**” in the ALICE DAQ WIKI) has expired.

Each time an event has been arrived, the ***readout*** process fills the base event header fields for which it is responsible (including the field ***eventTimestamp*** to tag the sub-event with an absolute timestamp), and it increments the run-time variable ***Number of sub-events*** for all event types. Then the routine ***ReadEvent*** is called, which is in charge of transferring the event data and for filling the base event and equipment header fields (see Chapter 3). Afterwards the ***readout*** process performs the following operations in the order described below:

1. checks that the mandatory fields in the base event header have been set by the ***ReadEvent*** routines;
2. checks that the ***eventId*** field in the base event header filled by the routine ***ReadEvent*** is not zero and has an increasing value for events of types ***PHYSICS_EVENT*** and ***CALIBRATION_EVENT***.;
3. checks whether the ***eventType*** field is ***START_OF_RUN*** for the first arrived event. If this event has a different type or if it arrives after the ***startOfData timeout*** period, then an “end of run” request with an error condition is issued. The LDC run parameter ***startOfData/endOfData event enabled*** must be set to enable this check, otherwise it is omitted;
4. checks whether the ***eventType*** field is ***END_OF_RUN*** after having received an “end of run” request. If such an event does not arrive within the ***endOfData timeout*** period, then the end of run phases (see Section 6.1.3) are executed with an error condition. The LDC run parameter ***startOfData/endOfData event enabled*** must be set to enable this check, otherwise an “end of run” request leads directly to the end of run phases;
5. fills the ***eventGdcId*** field in the base event header for ***PHYSICS_EVENT*** and

CALIBRATION_EVENT events with a default value in order to achieve a fair distribution of sub-event to multiple GDCs. The dispatch algorithm uses the "number in run" part of the ***eventId*** field if FIXED TARGET mode (see Section 3.4) or the "orbit counter" part of the ***eventId*** field if COLLIDER mode. The ***eventGdcId*** field can be overwritten by the ***edmAgent*** process (see Chapter 13). By default the destination of special event types (***START_OF_RUN***, ***START_OF_RUN_FILES***, ***END_OF_RUN***, ***END_OF_RUN_FILES***, ***START_OF_BURST***, ***END_OF_BURST***) is the first GDC;

6. if FIXED TARGET mode is selected (see Section 3.4), it fills the "number in run" part within the ***eventId*** field in the base event header for ***END_OF_BURST*** events in such a way that it contains the last event number held in the header of the last physics event in the burst and the number of events in the last burst. These values are used by the event builder to make consistency checks based upon independent criteria and redundant information.;
7. for ***START_OF_BURST*** events, it sets the run-time variable ***inBurst flag*** to 1, and for ***END_OF_BURST*** events sets the run-time variable ***inBurst flag*** to 0;
8. for FIXED TARGET mode it fills the run-time variables ***Number of sub-events in burst*** and ***Number of bursts*** in the shared memory control region;
9. increments the run-time variable ***Number of triggers*** in the shared memory control region for events of type ***PHYSICS_EVENT***;
10. fills the ***eventSize*** field in the base event header by taking into account the event scheme (streamlined or paged).;
11. sets the run-time variable ***Bytes injected*** in the shared memory control region for all event types.

The ***readout*** process exits the main event loop, if one of the following six conditions is met:

- the maximum number of events to be collected (given by the LDC run parameter ***Max. number of sub-events***) has been reached;
- the maximum number of bursts to be collected (given by the LDC run parameter ***Max. number of bursts***) has been reached and there is an ***END_OF_BURST*** type of event;
- the maximum number of bytes to be collected (given by the LDC run parameter ***Max. bytes to record)*** has been reached;
- the arrival of an "end of run" request combined with the following three cases:
  - the parameter ***startOfData/endOfData event enabled*** is not set, hence there is no waiting for an event of type ***END_OF_DATA***;
  - the parameter ***startOfData/endOfData event enabled*** is set and an event of type ***END_OF_DATA*** has been received within the timeout (given by the parameter ***endOfData timeout***);
  - the parameter ***startOfData/endOfData event enabled*** is set and an event of type ***END_OF_DATA*** has not been received within the timeout (given by the parameter ***endOfData timeout***);
- an event of type ***START_OF_DATA*** has not been received within the timeout (given by the parameter ***startOfData timeout***) when the parameter ***startOfData/endOfData event enabled*** is set;

- a fatal error has occurred.

All records are inserted in the ***readoutReadyFifo***. In addition, they are also injected in the buffer reserved for monitoring (see Chapter 5) if the following conditions are met:

- the LDC run parameter ***Monitor enable flag*** is set to 1;
- a monitor program requesting this type of events is running;
- there is enough space in the monitoring buffer.

6.1.3 End of run phases

At each end of run (EOR), the ***readout*** process performs the following sequence of operations in the order described below:

1. executes the common EOR scripts, if any, (see "**EOR.commands**" in the ALICE DAQ WIKI);
2. executes the detector EOR scripts, if any;
3. executes the specific EOR scripts, if any;
4. calls the routine ***DisArmHw***;
5. produces an EOR record: the ***eventType*** field of the base event header is set to ***END_OF_RUN***, the ***eventTypeAttribute*** field of the base event header is set to ***END_OF_RUN_START*** and the payload is empty;
6. prepares the common EOR files, if any (one record per file);
7. prepares the detector EOR files, if any (one record per file);
8. prepares the specific EOR files, if any (one record per file);
9. produces an EOR record: the ***eventType*** field of the base event header is set to ***END_OF_RUN***, the ***eventTypeAttribute*** field of the base event header is set to ***END_OF_RUN_END*** and the payload is empty;
10. updates the bookkeeping information with the run number, the physics events count, the SOB records count, the EOB records count, the trigger count and the burst count.

The EOR sequences have been split into phases, corresponding to the points enumerated above. At each time the timeout (given by the LDC run parameter ***Max. time for SOR/EOR phases***) is restarted to also allow longer initialization or ending phases.

6.1.4 Log messages

It is possible to choose where to direct the output of messages produced by the ***readout*** process. The script ***readout_startup.sh*** located in directory ***${DATE_READOUT_BIN}*** can be edited. The options proposed are the following:

- output via ***infoLogger*** (default);
- creation of an iconized xterm where all the output produced should appear;
- no output;

- output to a file (e.g. ***/tmp/Readout@****hostname****.log***);
- output to a file (e.g. ***${DATE_SITE_TMP}/Readout@****hostname****.log***).

By default the ***readout*** process uses the **infoLogger** package to report and trace errors or abnormal conditions, and to trace state changes. The operator can tailor these features to the required needs by setting the value of the LDC run parameter ***Logging level*** (see the ALICE DAQ WIKI to operate the system).

The readout process also updates the bookkeeping information at the end of the run through the ***LOGBOOK*** facility.

If the ***readout*** process crashes, a core dump for post-mortem analysis is produced in the ***${DATE_SITE_TMP}/${DATE_HOSTNAME}/readout*** directory.

6.2 The generic readList concept

This section describes how the ***readout*** program accesses the hardware by calling the five routines of the ***readList*** module (see Figure 6.1), which contains the code specific to a given electronics setup. Instead of writing several of these modules, the generic ***readList*** concept allows to group the code for all the electronics setups in another library called ***equipmentList*** (see Figure 6.2).

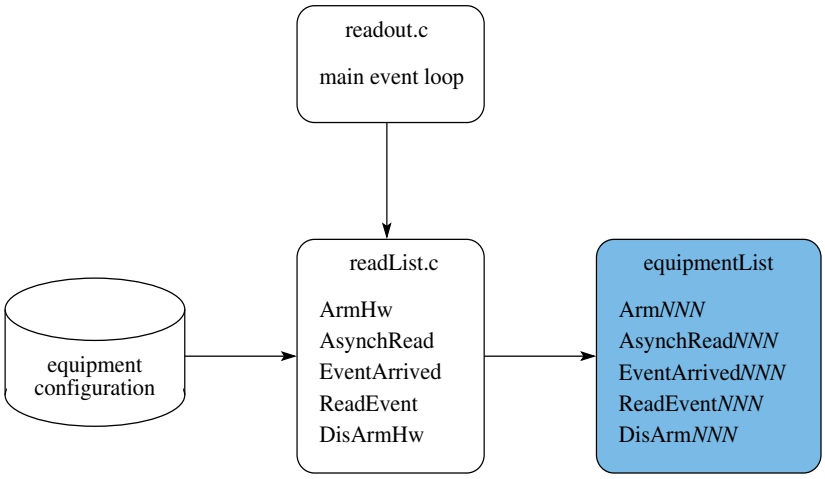

Figure 6.2 The generic readList concept of the readout process.

The readout software specific to an electronics setup can be written separately for a so-called ***equipment***. One equipment is responsible for generating data from an electronics board or a set of electronics boards, depending on how the readout software is structured. A set of equipment-handling routines deals with one single equipment, thus the code is more modular and readable. The routine names are fixed by convention: the name is obtained by concatenating the prefix ***Arm***, ***DisArm***, ***AsynchRead***, ***ReadEvent*** and ***EventArrived*** with the name of the equipment type *NNN* as declared by the equipment configuration. All the five routines must be implemented for an equipment. If one of these functionalities is not required, a dummy routine should be provided. The details of this library is described in Section 6.4.

The equipment configuration defines the equipments used for each LDC. It is done with the equipment databases (see Chapter 4).

An equipment may be repeated several times in a detector; each run-time call will be distinguished by a different set of equipment parameters. The configuration file specifies the selection of the active equipments and the setting of the parameters that will be passed to the readout routines. Therefore, it is possible to modify the readout program behaviour without changing the readout executable code.

As a result of this generic ***readList*** concept, the sub-events of an LDC are divided further into smaller parts, called ***equipment data*** or ***fragments***. Each equipment data block begins with an equipment header, followed by the equipment raw data (see Chapter 3). Hence a fully built event contains the sub-events from the various LDCs, which in turn contain the fragments from the equipments.

In order to realize this concept, the generic ***readList*** module implements the following functions:

- ***ArmHw***
  It loads the equipment configuration, identifies the equipments involved in the readout of the LDCs, saves them in a table and then calls the ***ArmNNN*** routine for each active equipment in the order specified in the configuration file. It also sets the run-time variable ***Number of equipments*** to the number of data generating equipments.
- ***AsynchRead***
  It calls the ***AsynchReadNNN*** routine for each active equipment configured in the equipment database.
- ***EventArrived***
  It calls the ***EventArrivedNNN*** routine of the active equipment configured in the equipment database.
- ***ReadEvent***
  It calls the ***ReadEventNNN*** routine for each active equipment. In addition it generates the equipment header if it is an equipment generating data.
- ***DisArmHw***
  It calls the ***DisarmNNN*** routine for each active equipment.

The handling of the equipment routines must be further configured with the help of the optional attributes ***GENDATA*** and ***TRIGGER***. They are assigned to an equipment type as part of the equipment configuration.

- ***GENDATA***: Usually an equipment generates data, but it may be convenient to isolate some specific processing in an equipment that does not generate any data. Only the equipment routine ***ReadEventNNN*** is able to produce data when the attribute ***GENDATA*** is set. If this attribute is set, then the equipment header is generated and the ***ReadEventNNN*** routine is called with parameters to access the equipment header and the raw data. If this attribute is omitted, then no equipment header is generated and the equipment routine is called with parameters that do not allow to access the equipment header and the raw data.
- ***TRIGGER***: The trigger hardware plays a special role, since it is in charge to indicate if a sub-event has arrived or not. This decision needs to be captured by the ***EventArrivedNNN*** routine of an equipment. However, this equipment routine is only called if the attribute ***TRIGGER*** is set. Several trigger equipments may be declared in the equipment configuration, but only one per LDC should

be active. If there are more of them, no warnings are given and only the first one is used.

6.3 Using the generic readList

The use of the generic ***readList*** (see Section 6.2) requires the preparation of two components:

- A source code file that contains the equipment routines to handle the readout of all the equipments that may be activated in an LDC. A detailed description on how to prepare it is given in Section 6.4.
- An entry in the equipment database describing the configuration of each active equipment.

These two components are strictly correlated and must match one another. There is no mechanism to make sure that this is the case. Error conditions due to a mismatch are discovered at start of run and will immediately stop the run. The conventions tying these files are the following:

- The name of the equipments in the equipment database constrains the name of the equipment routines in the source code file. A prefix is added to the equipment type, as explained in Section 6.2.
- The equipment routines must be declared using macros provided in ***readList_equipment.h***. These macros provide the link between the equipment name (read by ***readList*** from the equipment database) and the address of the readout routines (to be called by ***readList***). The usage of these macros is explained in Section 6.4.3.
- The configuration described in the equipment database is shared by all the LDCs in the system, therefore all the equipments used in each of the LDCs must be declared there. A readout program has to be linked to the ***equipmentList*** library. It is possible to declare in the equipment database equipments, for which the handling code is not provided. The readout program will abort the run during the ***Arm*** phase, only if missing LDCs are selected as active. The advantage is to build a ***readout*** program containing only the equipments that will be used in the target LDC, whereas the equipment configuration file can still contain more equipments than the ones encoded in the ***readout*** program.
- Parameters that can be passed to the equipment routines. The parameter declarations in equipment database and the source code file must match. The handling of these equipment parameters is explained in Section 6.4.2.

The DATE package **readList** contains four suites of equipment software: ***TEST, CTP, DDL*** and ***UDP***. Only one of these suites will be present in any LDC. Users can add their own specific equipment software in one of these suites or create a new one. With the aid of the TEST suite, software simulated events are produced mostly for testing purpose.

The DDL suite contains all the equipment software to handle the RORC readout (see Section 7.1). The CTP suite is an equipment developed for the ALICE trigger system to read out information sent by the Central Trigger System (CTP). The UDP

suite contains all the equipment software to handle the UDP readout (see Section 7.3). To each equipment suite belongs the source code file for the equipment routines and the associated ***GNUmakefile***. A summary is given in Table 6.1.

Table 6.1 Equipment suites in the readList package

equipment suite	components
TEST	***equipmentList_TEST.c*** ***GNUmakefile_TEST***
CTP	***equipmentList_CTP.c*** ***GNUmakefile_CTP***
DDL	***equipmentList_DDL.c/.h*** ***GNUmakefile_DDL***
UDP	***equipmentList_UPD.c/.h*** ***GNUmakefile_UDP***

As an example, to prepare a readout program for the ***TEST*** suite by using the generic ***readList***, see "***How to prepare a readout program***" in the ALICE DAQ WIKI.

The equipment configuration may be changed between runs just by modifying the equipment database. Changes are taken into account only at the next start of run. If the modifications concern only the descriptive parts of the equipment, such as:

- adding or removing equipments assigned to an LDC;
- activating or deactivating equipments assigned to an LDC;
- changing the value of an equipment parameter.

Then the ***readout*** program does not need to be re-built. However, the ***readout*** program must be re-build if the modifications in the equipment configuration (e.g. adding equipment parameters) entail changes in the source code files.

6.4 The equipmentList library

This section provides the synopsis, the parameter handling and the functional references of the five equipment routines that are required for each equipment in the ***equipmentList*** library. The user has to provide these routines that are specific to a readout electronics setup. Examples of these routines can be found in the source code files of the four equipment suites, as shown in Table 6.1.

6.4.1 Synopsis of the equipment routines

The synopsis of the routines ***ArmNNN***, ***AsynchReadNNN***, ***EventArrivedNNN***, ***ReadEventNNN***, and ***DisArmNNN*** for an equipment of type *NNN* is given below. All five routines must be provided for an equipment, even if some of them are empty.

Upon return from these routines the ***readout*** process checks the content of the global variable ***readList_error***, whose value allows signaling error conditions. If its value is different from 0, the readout process logs an error message containing the value of the variable and the name of the routine originating the error through the global variable ***readList_errorSource*** (which is filled by the generic ***readList*** module), and asks to stop the run.

Arm*NNN*

Synopsis

```
#include "rcShm.h"
#include "event.h"
#include "readList_equipment.h"

void ArmNNN( char *par )
```

Description The ***ArmNNN*** routine is called by ***ArmHw*** at each start of run for equipment type *NNN*, after the execution of the start of run scripts and before the transfer of the start of run files on the output medium. This equipment routine should perform all the actions needed at the beginning of the run, e.g. the initialization of the hardware and of the trigger, or the assignment of values to global static variables. The routine cannot generate any data.

Parameters:

- The parameter `par` is a pointer to a memory region containing the sequence of pointers to the values of the parameters of the component being armed; these values are read at run time from the equipment database and are assigned to the equipment before arming the LDC.

Returns The routine does not return any value. The global variable ***readList_error*** should be used to signal error conditions, which will provoke the log of a message and the termination of the run.

AsynchRead*NNN*

Synopsis

```
#include "rcShm.h"
#include "event.h"
#include "readList_equipment.h"

void AsyncReadNNN( char *par )
```

Description The ***AsynchReadNNN*** routine is always called by ***AsynchRead*** in the main event loop (see Figure 6.1) for equipment type *NNN*. It is called in a strictly closed loop without sleeping just before the ***EventArrivedNNN*** routine. Since this equipment routine is invoked before the ***ReadEventNNN*** routine, it offers the possibility to perform the readout of hardware that produces an asynchronous data flow. However, it cannot pass any data to readout. Only the routine ***ReadEventNNN*** is designed for this purpose. If this feature is not needed, this routine should be left empty.

Parameters:

- The parameter `par` is a pointer to a memory region containing the sequence of pointers to the values of the parameters of the component being armed; these values are read at run time from the equipment database and are assigned to the equipment before arming the LDC.

Returns The routine does not return any value. The global variable ***readList_error*** should be used to signal error conditions, which will provoke the log of a message and the termination of the run.

EventArrived*NNN*

Synopsis

```
#include "rcShm.h"
#include "event.h"
#include "readList_equipment.h"

int EventArrivedNNN( char *par )
```

Description The ***EventArrivedNNN*** routine is called by ***EventArrived*** in the main event loop (see Figure 6.1) only if the ***TRIGGER*** attribute is assigned to this equipment type *NNN*. It is called in a strictly closed loop without sleeping just after the ***AsynchReadNNN*** routine. The purpose of this routine is to indicate whether a trigger has occurred or not. This can be done either by polling and returning immediately (with 0 if no trigger has occurred) or by waiting for an interrupt with an appropriate driver call for the hardware.

Parameters:

- The parameter `par` is a pointer to a memory region containing the sequence of pointers to the values of the parameters of the component being armed; these values are read at run time from the equipment database and are assigned to the equipment before arming the LDC.

Returns The function must return the value 1 if a new event has arrived, or 0 otherwise. The global variable ***readList_error*** should be used to signal error conditions, which will provoke the log of a message and the termination of the run.

ReadEvent*NNN*

Synopsis

```
#include "rcShm.h"
#include "event.h"
#include "readList_equipment.h"

int ReadEventNNN( char *par,
                  struct eventHeaderStruct *ev_header,
                  struct equipmentHeaderStruct *eq_header,
                  int *data )
```

Description The ***ReadEventNNN*** routine is always called by ***ReadEvent*** in the main event loop (see Figure 6.1) for equipment type *NNN* after a trigger has arrived. This equipment routine is the place to make the data available to the readout and to fill the fields in the base event and equipment headers.

Parameters:

- The parameter `par` is a pointer to a memory region containing the sequence of pointers to the values of the parameters of the component being armed; these values are read at run time from the equipment database and are assigned to the equipment before arming the LDC.
- The parameter `ev_header` is a pointer to the base event header (see Chapter 3), which is defined in the ***${DATE_COMMON_DEFS}/event.h*** header file.
- The parameter `eq_header` is a pointer to the equipment header (see Chapter 3), which is defined in the ***${DATE_COMMON_DEFS}/event.h*** header file. This pointer is only valid (i.e. different from NULL) if the attribute ***GENDATA*** is assigned to this equipment type in the declaration part of the equipment database.
- The parameter `data` is a pointer to the raw data block to fill in. This pointer is only valid (i.e. different from NULL) if the attribute ***GENDATA*** is assigned to this equipment type in the declaration part of the equipment database. If this equipment is designed for streamlined events (LDC run parameter ***Paged data flag*** is 0), then the data must be copied to the memory where parameter `data` is pointing. If this equipment is designed for paged events (LDC run parameter ***Paged data flag*** is 1), then the equipment vector (see Chapter 3) needs to be placed where parameter `data` is pointing.

The main readout program sets the variable ***readList_bufferSize*** to the value of the available space for the data of the current equipment. The ***ReadEventNNN*** routine is supposed to use this value in order to prevent writing beyond the space available. This variable is accessible by making the following declaration:

```
extern int readList_bufferSize;
```

If streamlined events, the value of the variable ***readList_bufferSize*** is given by the LDC run parameter ***Max. event size*** (see Chapter 2) minus the size of the base event and the equipment header. For paged events, the value of the variable ***readList_bufferSize*** is given by the size of the first level vector minus the size of the equipment header. After calling the ***ReadEventNNN*** routine, the new value of the variable ***readList_bufferSize*** is calculated (both for streamlined or paged events) by the generic ***readList*** module. If the value becomes negative (i.e. overflow) the following error is provoked: the variable ***readList_error*** is set to 15 and the variable ***readList_errorSource*** is set to the string “ReadEvent equipment *N* overflow”, where N is the ordinal number of the faulty equipment. *N* is obtained by counting the active equipments only. The run will then be aborted.

The ***ReadEventNNN*** routine is supposed to fill the fields in the base event and in the equipment headers. Refer to Chapter for a detailed description of them and how to access them with the help of macros. The most important header fields are:

- ***eventId*** in the base header: it is mandatory, except for ***END_OF_BURST*** events. This variable is the event number in the run, where both the COLLIDER mode or the FIXED TARGET mode are encoded, see Section 3.4. It must always

increase during a run, but it allows for gaps. The readout process checks that the value of this field is non-zero and increasing for consecutive events inside a run for events of the type ***PHYSICS_EVENT*** and ***CALIBRATION***, and asks to stop the run if not. This field must be the same in all the sub-events of the same event, since it is used by the event builder to perform consistency checks. This field should be set to 0 for ***START_OF_BURST*** events, while for ***END_OF_BURST*** events this field is overwritten by the readout process which sets it to the last event number held in the header of the last physics event in the burst.

- ***eventType*** in the base header: it is mandatory and initialized to the type ***PHYSICS_EVENT***. This variable marks the type of record. The readout process increments the trigger number only for the ***PHYSICS_EVENT*** type of record and not for other types of records.
- ***eventTypeAttribute*** in the base header: it is optional and initialized to 0. This variable contains the system-defined attributes and the user-defined attribute associated to an event.
- ***eventTriggerPattern*** in the base header: it is optional and initialized to 0. This variable contains the level 2 trigger pattern.
- ***eventDetectorPatttern*** in the base header: it is optional and initialized to 0. This variable contains the level 2 detector pattern.
- ***equipmentId*** in the equipment header: it is optional and initialized to 0. It is set by an equipment parameter to distinguish between equipments of the same type.
- ***equipmentBasicElementSize*** in the equipment header: it is optional and initialized to 0. Usually it is set to 4 bytes.
- ***equipmentTypeAttribute*** in the equipment header: it is optional and initialized to 0. This variable contains user-defined attribute associated to an equipment.

Upon return from this routine, the main readout program checks that the user has filled the mandatory fields in the event header and updates some variables used in the runControl status display.

Returns If the equipment produces data (i.e. the attribute ***GENDATA*** is assigned to this equipment type), the routine must return the number of bytes actually taken. This rule applies to both streamlined and paged events. If the equipment does not produce data (i.e. attribute ***GENDATA*** is omitted to this equipment type), the routine should return 0. The global variable ***readList_error*** should be used to signal error conditions, which will provoke the log of a message and the termination of the run.

DisArm*NNN*

Synopsis

```
#include "rcShm.h"
#include "event.h"
#include "readList_equipment.h"

void DisArmNNN( char *par )
```

Description The ***DisArmNNN*** routine is always called by ***DisArmHw*** at each end of run for equipment type *NNN*, after the execution of the end of run scripts and before the transfer of the end of run files on the output medium. This equipment routine should perform all the actions needed at end of run, such as the release of unused memory, the switching off of high voltages, or the saving of error statistics that may have been collected.

Parameters:

- The parameter `par` is a pointer to a memory region containing the sequence of pointers to the values of the parameters of the component being armed; these values are read at run time from the equipment database and are assigned to the equipment before arming the LDC.

Returns The routine does not return any value. The global variable ***readList_error*** should be used to signal error conditions, which will provoke the log of a message and the termination of the run.

6.4.2 Accessing the parameters

The equipment routines have access to the following parameters:

- Equipment parameters are specific to an equipment type. They are accessible via a pointer received as first parameter in the routine call:

```
char *par;
```

- Global parameters can be used by all equipments. They are accessible via a global pointer:

```
char *globPar;
```

The order, type and format of these parameters is a matter of convention. Coherence must be assured between what is specified in the equipment database and the source code file. No check is performed by ***readList*** before calling the library.

The values of the equipment parameters are copied into memory at run time while reading the equipment database configuration by following this convention on their formats.

To ease the use of the parameters it is suggested to cast their memory pointer into a pointer to a structure with proper fields, according to the declaration in the equipment database.

Listing 6.1 shows a skeleton of a source code file for the routines of equipment type ***Rand*** (lines 1-21). This is a simple equipment for testing purposes in the ***TEST*** suite. Important is the way how the parameters are declared as pointers in a C typedef (lines 2-7), casted to a local pointer (line 10) and eventually accessed (line 11).

6.4.3 The function references

In order to make the functions contained in the library accessible from the generic ***readList***, references to them must be created in the library through an array and a macro defined into the ***readList_equipment.h*** header file. Independent of their equipment attributes, the reference to their routines has to be made as follows:

1. The value returned applying the ***EQUIPMENTDESCR*** macro to the name of each equipment type must be put into the ***equipmentDescrTable*** array. The order is not important.
2. The variable ***nbEquipDescr*** must be set to the number of entries in the ***equipmentDescrTable*** array.

Listing 6.1 (lines 23-27) shows an example on how to make the function references of the sketched above equipment type ***Rand***.

Listing 6.1 Example of an equipment source code file

```
 1: /******************** equipment Rand *********************/
 2: typedef struct {
 3:   long32 *eventMinSizePtr;
 4:   long32 *eventMaxSizePtr;
 5:   short *eqIdPtr;
 6:   short *triggerPattern;
 7: } RandParType;
 8:
 9: void ArmRand(char *parPtr) {
10:   RandParType *randPar = (RandParType *)parPtr
11:   printf("Arming random generator (id = %hd)" \
             with min = %ld max = %ld triggerPattern = %d\n",
             *randPar->eqIdPtr,
             *randPar->eventMinSizePtr,*randPar->eventMaxSizePtr,
             *randPar->triggerPattern);
12: ... }
13:
14: void DisArmRand(char *parPtr) {}
15:
16: void AsynchReadRand(char *parPtr) {}
17:
18: int ReadEventRand(char *parPtr,
                    struct eventHeaderStruct *header_ptr,
                    struct equipmentHeaderStruct *eq_header_ptr,
                    int *data_ptr) {
19: ... }
20:
21: int EventArrivedRand(char *parPtr) {}
22:
23: /****************** table of functions ******************/
24: equipmentDescrType equipmentDescrTable[] = {
25:   EQUIPMENTDESCR( Rand )
26: };
27: int nbEquipDescr =
        sizeof(equipmentDescrTable) / sizeof(equipmentDescrTable[0]);
```

7 The RORC readout software

This chapter describes the DATE readout software for:

- the RORC (Read-Out Receiver Card) which is the interface between the DDL (Detector Data Link) and the LDC.
- The Ethernet port which can be used by DATE as alternative data source.

Information on the implementation of the two equipments can be found in the following Sections:

7.1 Introduction to the RORC equipment

The ***Read-Out Receiver Card*** (RORC) is a PCI master card that provides an interface between the ***Detector Data Link*** (DDL) and the PCI, PCI-X or PCI Express bus of a commodity PC. The DDL consists of a ***Source Interface Unit*** (SIU), which is attached to the front-end electronics inside the detector and a ***Destination Interface Unit*** (DIU). The SIU and the DIU are connected through a pair of optical fibres to transmit data up to a rate of 200 MB/s. Five types of RORCs have been designed:

- The pRORC has a 32 bit/33 MHz PCI bus interface and handles one DDL channel using a piggy-backed DIU.
- The single channel D-RORC has a 64 bit/64 MHz PCI bus interface and handles one DDL channel via a piggy-backed DIU.
- The dual channel D-RORC has a 64 bit/64 MHz PCI bus interface and handles two DDL channels with embedded DIUs.
- The dual channel D-RORC has a 64 bit/100 MHz PCI-X bus interface and handles two DDL channels with embedded DIUs.
- The dual channel D-RORC has a x8 PCI Express (Gen. 2) bus interface and handles two DDL channels with embedded DIUs.

In this chapter, these cards are commonly referred as RORC, since they do not differ from the software point of view. Depending on the acquisition needs and on the number of available PCI bus slots, one PC can be equipped with several RORCs (up to 6). Each RORC has a revision number and a unique serial number in its configuration EPROM. The RORC can generate pre-defined data streams for testing purposes. Dual channel RORC's can be switched to "splitter mode": data arriving in one channel is sent out on the other channel in automatic way.

The data flow from the DIU to the PC memory is driven by the DMA engine of the RORC firmware in co-operation with the RORC readout software. During one DMA, only one ***data page*** can be written. Data pages that belong to the same sub-event are referred as ***fragment***, which is transferred over the DDL by one or more ***DDL blocks***. Each block can be up to 4x(2^19-1) = 2097148 bytes. For a comprehensive description of the DDL and the RORC see the Web site ***http://cern.ch/ddl***. The stand-alone utility programs for the RORC are documented in Chapter 20.

DATE provides all the necessary readout software to operate RORC devices on a PC running Linux via the two following packages:

- Package ***rorc***: it contains the Linux driver module, the library functions, and some utility programs to have an interface to a RORC device. This package is self-contained in the ***${DATE_ROOT}/rorc*** directory.
- Package ***readList***: it contains the equipment software to realize a readout program for an LDC with a RORC device. The software is located in the ***${DATE_ROOT}/readList*** directory and depends on package ***rorc***.

7.2 Internals of the RORC equipment

The goal of RORC readout software is to operate several RORC devices attached to one LDC by considering the asynchronous data flow and the scattered location of data pages in the main memory. Moreover, the RORC readout software has to be structured in equipment routines as explained in Chapter 6. The rest of this chapter presents the internals of the RORC equipment software, which explains in more details the mechanism to transfer data with a RORC device (see Section 7.2.2), the software elements to handle it (see Section 7.2.3), the data flow for multiple RORC devices active in the LDC (see Section 7.2.4), and the pseudo code of the RORC equipment routines (see Section 7.2.6).

7.2.1 Event Identification

The identification of the sub-event is given by the ***eventId*** field in its base event header. The ***RorcData*** equipment writes this field by taking into account the two common run parameters ***Collider mode*** and ***Common Data Header Present*** (see the ALICE DAQ WIKI). Their usage is illustrated in Figure 7.1.

If the raw data contains the Common Data Header (see Section 3.9) with the ***cdhEvent1*** field (12 bit bunch-crossing number) and the ***cdhEvent2*** field (24 bit orbit number), then both ***Collider mode*** and ***Common Data Header Present*** should be set. This case is depicted in the upper half of Figure 7.1. Setting the parameter ***Common Data Header Present*** instructs the software to extract these Common Data Header fields and to use them for filling the ***eventId*** field (***orbit counter*** and ***bunch-crossing counter*** part). Setting the parameter ***Collider mode*** ensures that the ***eventId*** field is decoded in COLLIDER mode (see Section 3.4). The 28 bit ***period counter*** of the COLLIDER mode identification is incremented by software, whenever the orbit number wraps. Not setting the parameter ***Collider mode*** in this scenario leads to an unsuitable event identification. If several fragments need to be assembled to one sub-event, then the ***RorcData*** equipment executes consistency checks among the Common Data Header fields of their fragments.

If the raw data does not contain the Common Data Header, then none of the common run parameters ***Collider mode*** and ***Common Data Header Present*** should be set. This case is depicted in the lower half of Figure 7.1. When the parameter ***Common Data Header Present*** is not set, then the identification is done by the run-time variable ***Number of triggers***. This is a 32 bit software counter, which is incremented for each arrived sub-event. It is used to set the ***eventId*** field (***number in run part***) when decoded in FIXED TARGET mode (see Section 3.4). Therefore parameter ***Collider mode*** should be not set in this scenario to avoid an incorrect event identification.

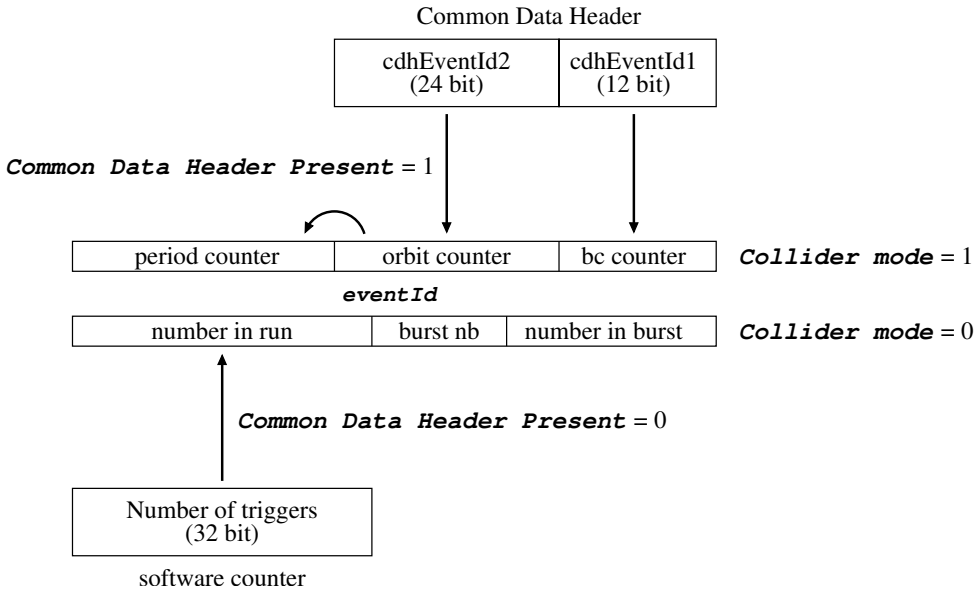

Figure 7.1 Event identification mechanism of the RorcData equipment.

7.2.2 Data transfer mechanism of the RORC device

The mechanism to transfer data from the DDL to the PC memory by one RORC device involves three activities that may run in parallel:

1. ***Fill***: a process using the RORC has to fill the `rorcFreeFifo` with references to free data pages to which the data stream from the DDL has to be transferred. The `rorcFreeFifo` is located in the firmware of the RORC and has 128 entries. Each entry consists of three fields: the physical start address (32 or 64 bits according to the address mode) of the data page, the size (24 bits) in words (= 4 bytes) of the data page, and the index (8 bits) of the `rorcReadyFifo` holding information about the data transfer. In fact, the index field is part of the physical address (bit 3 to 10) of an `rorcReadyFifo` entry. The latter FIFO is further described in the part for activity ***Transfer*** and ***Scan***.
2. ***Transfer***: the RORC transfers data from the DDL to the data page addressed by the top entry of the `rorcFreeFifo`. This can only take place if there is data arriving from the DDL and if the `rorcFreeFifo` is not empty. When the data transfer is completed, the RORC fills the `rorcReadyFifo` with information about the transfer. The `rorcReadyFifo` is located in the memory of the PC (the RORC needs to know the physical start address of it) and has also 128 entries. The RORC writes to the corresponding entry of the `rorcReadyFifo`, which is determined by the index field of the top entry of the `rorcFreeFifo`. Each entry consists of two fields: the length (32 bits) in words of the transferred data, and the transfer status (32 bits). The status field can either be a DTSTW (Data Transmission Status Word) or 0 if more pages are about to follow. A DTSTW marks the end of a DDL block (to allow fragments larger than 2 MB there can be several DDL blocks) or the end of a fragment. For all cases the status field also contains an error bit to indicate a transfer problem. Whenever a free data page with a particular index is given to the RORC during a fill activity, the status field of this indexed `rorcReadyFifo` entry has to be initialized to -1.
3. ***Scan***: a process using the RORC needs to scan the `rorcReadyFifo` entries in

order to find out if there are fragments ready in one or more pages. By looking up the status fields, a sequence can be obtained such as "`0 0 0 DTSTW DTSTW 0 DTSTW 0 0 0 -1 -1`". In this example there are 3 fragments ready (the first one consists of 4 pages, the second of 1 page, and the third of 2 pages) and one fragment is arriving but not finished. To simplify the scan activity, these entries are in increasing order with a wrap-around at 128.

7.2.3 Elements to handle the RORC device

Based on the mechanism to transfer data with the RORC device, Figure 7.2 shows the software elements to handle it. They represent the main data structures of the readout software for the RORC device:

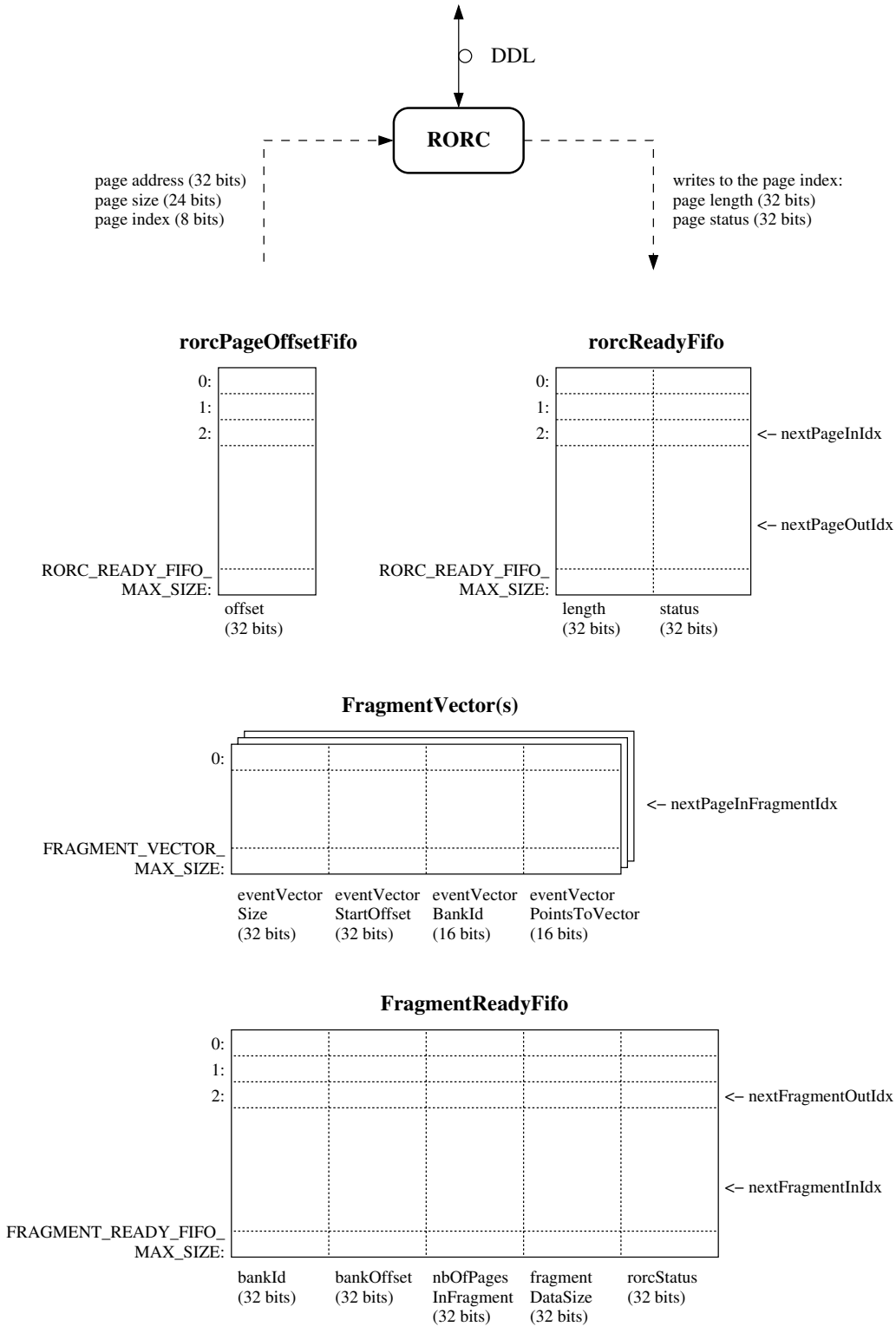

Figure 7.2 The software elements to handle the RORC device.

1. ***`rorcPageOffsetFifo`***: it is used to remember the bank offset of a free page given to the RORC during a fill activity, since this information is missing when the RORC has finished the transfer and is needed to find the location of the data page during a scan activity. The `rorcPageOffsetFifo` has up to 128 entries, where each entry holds the offset (32 bits) within the bank where the free data page is located. The handling is done by the indices `nextPageInIdx` and

nextPageOutIdx, which are further explained in the rorcReadyFifo description.

2. ***rorcReadyFifo***: it is used by the RORC during a transfer activity to store information about the completed data transfer(s), and during a ***Scan*** activity to retrieve the written data pages belonging to a fragment. As already pointed out in Section 7.2.2, the rorcReadyFifo has up to 128 entries, where each entry consists of the length field (32 bits) and the transfer status field (32 bits). The RORC needs to know the physical start address of the rorcReadyFifo. To operate this FIFO, two indices nextPageInIdx and nextPageOutIdx and a flag rorcReadyFifoFull are used (see Section 7.2.6). The index nextPageInIdx always points to the entry where the RORC device will make a notification about its next completed data transfer. This index is advanced (wrap-around at 128) only during a scan activity. The index nextPageOutIdx always points to the entry that determines the next free data page for the fill activity. Only during the fill activity the index nextPageOutIdx can be advanced (wrap-around at 128), but it must not overtake the index nextPageInIdx. At initialization the RORC is filled with 128 free pages and if no data page has arrived so far, all entries in the rorcReadyFifo have -1 in their status field, the index nextPageInIdx is equal to index nextPageOutIdx, and the flag rorcReadyFifoFull is ***TRUE***. During the scan, the status field is read at index nextPageInIdx. If a page has arrived, the status field has turned to 0 or to a DTSTW, and the index nextPageInIdx can be advanced until a status field of -1 is hit. At the same time the RORC can be filled with new free pages, whose indices are taken by advancing the index nextPageOutIdx up to nextPageInIdx. If the filling of new pages fails (e.g. allocation of pages not possible) over a longer period, it may happen that nextPageOutIdx reaches nextPageInIdx after wrapping around and the flag rorcReadyFifoFull is ***FALSE***. In this condition the rorcReadyFifo is empty and the data transfer cannot continue, since there are no free pages in the rorcFreeFifo.

3. ***FragmentVector(s)***: they are used to link together the data pages of one fragment transferred by the RORC. The FragmentVector will become the 2nd level vectors in the event tree (see Figure 7.3). During the ***Scan*** activity the status field is read of the rorcReadyFifo entries. For each complete fragment (status fields form a sequence of zeros, interspersed DTSTWs for DDL blocks, and a terminating DTSTW), a FragmentVector will be produced, where each entry represents one data page. An entry has four fields: the eventVectorBankId field (16 bits) and the eventVectorStartOffset field (32 bits) of a data page, the eventVectorSize field (32 bits) in bytes of the data block within the page, and the eventVectorPointsToVector field (16 bits) to indicate that the pair <bank id, bank offset> points directly to a data page and not to another vector. All this information is obtained from the rorcReadyFifo in conjunction with the rorcPageOffsetFifo. Moreover, the filling is assisted by an index nextPageInFragmentIdx. The parameter FRAGMENT_VECTOR_MAX_SIZE gives the maximum number of data pages per fragment, which must be known in advance for the allocation.

4. ***FragmentReadyFifo***: it is used to queue the fragments transferred by the RORC. Given that one LDC can host more than one RORC, the appropriate fragments from each RORC device need to be built together in a sub-event before any further processing. Since the delivery rate of fragments may differ between the RORCs, the FragmentReadyFifo is designed to buffer these fragments. The parameter FRAGMENT_READY_FIFO_MAX_SIZE gives the

maximum number of entries, where each entry has 5 fields: the `bankId` field (32 bits) and the `bankOffset` field (32 bits) to locate the `FragmentVector`, the `nbOfPagesInFragment` field (32 bits) to know the number of entries in the `FragmentVector`, the `fragmentDataSize` field (32 bit) to know the size in bytes of the fragment, and the `rorcStatus` field (32 bit) to store the DTSTW (the last one terminating the fragment). To operate this `FragmentReadyFifo` the two indices `nextFragmentInIdx` and `nextFragmentOutIdx` and the flag `fragmentFifoFull` are used in a simple way (see Section 7.2.6). An entry is made into the `FragmentReadyFifo` whenever one complete fragment has been found during the scan activity. All the fields in the `FragmentReadyFifo` entry can be filled by exploiting the current `FragmentVector` and the entry of the `rorcReadyFifo` status field. When a sub-event is being built, an entry is taken out from `FragmentReadyFifo`.

5. ***`Data page(s)`***: They contain the raw data delivered by the RORC. For simplicity they are not shown in Figure 7.2. A data page goes through the following cycle: during the ***Fill*** activity it is allocated from buffer `readoutData` and given to the RORC, held by the `rorcFreeFifo` and waiting to be filled by the RORC; during the ***Transfer*** activity it is written by the RORC, held by the `rorcReadyFifo` and waiting to be scanned as ready (status 0 or a DTSTW). During the ***Scan*** activity it is attached to a `FragmentVector` and a complete fragment is carried along with the `FragmentReadyFifo`. During sub-event building it is transferred to the `readoutReadyFifo`, processed and de-allocated by the ***`recorder`*** process (see Section 10.3). As a matter of fact, there is no memory-to-memory copy of data pages in this scheme.

7.2.4 Equipments to handle the RORC device

The RORC readout software is implemented by three equipment types in the ***DDL*** equipment suite which are provided in the package **`readList`** (see Chapter 6):

1. ***`RorcData`*** is responsible for initializing the RORC and handling the autonomously delivered data pages from the RORC. One such equipment needs to be instantiated for each DDL channel in an LDC. It has the attribute ***`GENDATA`*** and can be configured by several parameters (see Section 7.2.4.1).
2. ***`RorcTrigger`*** is responsible for indicating the availability of a sub-event, where each DDL channel contributes a fragment. One equipment for each LDC needs to be instantiated. It has the attribute ***`TRIGGER`*** and can be configured by one parameter (see Section 7.2.4.2).
3. ***`RorcSplitter`*** enables a dual channel D-RORC to work in “split mode”. It does not have any attribute and can be configured by parameters (see Section 7.2.4.3).

The first one, ***`RorcData`***, has the attribute ***`GENDATA`*** and is in charge of reading the data from one RORC channel. The second one, ***`RorcTrigger`***, has the attribute ***`TRIGGER`*** and is used for triggering one or more RORC channel(s). Their pseudo code is given in Section 7.2.6.

The equipment routines are participating in the construction of a sub-event in paged mode (see Chapter 3), as shown in Figure 7.3. The sub-event is described by a 1st level vector, which is composed of the base header and three equipments; each of the equipments is represented by an equipment header and an equipment vector. The equipment vector of the first equipment points via a pair <bank id, bank

offset> to a 2nd level vector, which is composed of three payload vectors in sequence. Each vector points again via a pair <bank id, bank offset> to one data page. The equipment vector of the second equipment points to a 2nd level vector with two payload vectors. The equipment vector of the third equipment points directly via a pair <bank id, bank offset> to one data page. As an option, an equipment vector may always point to a 2nd level vector, even if it contains only one payload vector.

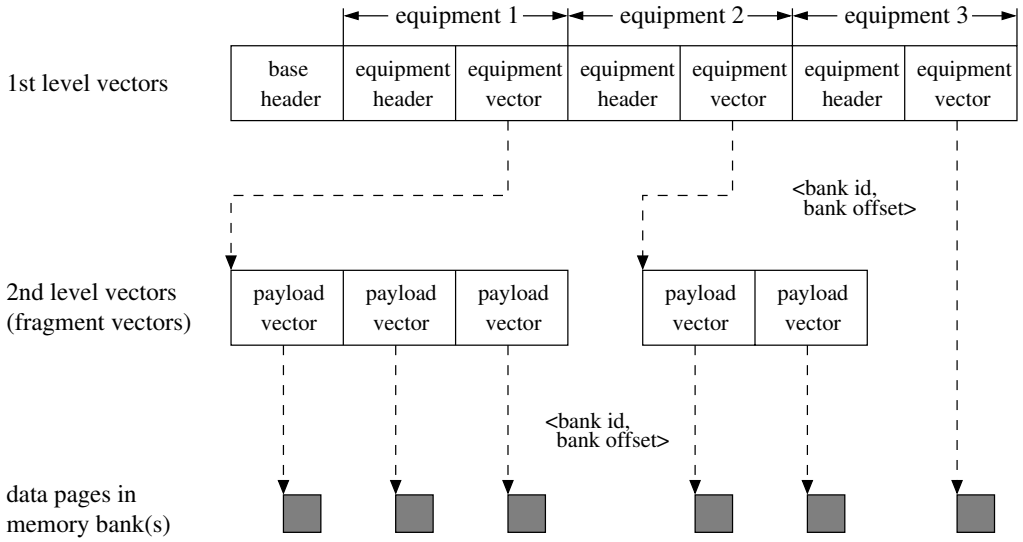

Figure 7.3 Example of one sub-event in paged event mode.

The equipment ***RorcSplitter*** controls the feature on a dual channel D-RORC to duplicate the data stream from an incoming channel to an outgoing channel. It does not carry any attribute, since it does neither generate data nor functions as a trigger. In order to enable this feature, the run options HLT checkbutton (see the ALICE DAQ WIKI) must be set.

7.2.4.1 Equipment RorcData

The equipment routines of ***RorcData*** for reading out data from one RORC channel are the following:

1. `ArmRorcData()`: it checks equipment parameters and logs a message. Allocates and initializes the `rorcPageOffsetFifo`, the `rorcReadyFifo`, and the `FragmentReadyFifo`. It resets and starts the RORC device.
2. `AsynchReadRorcData()`: it tries to fill the `rorcFreeFifo` with free pages. It scans the `rorcReadyFifo` to determine if data pages have arrived from the RORC device. If the status field of a data page has 0 or a DTSTW that terminates a DDL block, then this page will be added to the `FragmentVector`, which will be allocated if it is the first page of a fragment. If the status field of a data page has a DTSTW that terminates a fragment, then this page will be added to the `FragmentVector` as well, and the `FragmentVector` will be put into the `FragmentReadyFifo`. For triggering purpose (see Section 7.2.4.2), the global flag `allFragmentsReadyFlag` will be set to ***FALSE*** if the `FragmentReadyFifo` is empty.
3. `EventArrivedRorcData()`: this routine is empty.

4. `ReadEventRorcData()`: it takes out a fragment from the `FragmentReadyFifo` and uses it to fill the equipment header and equipment vector of the 1st level vector.
5. `DisArmRorcData()`: it stops the RORC device and deallocates all memory blocks.

7.2.4.2 Equipment RorcTrigger

The equipment routines of ***RorcTrigger*** are used for triggering the RORC devices. The global flag `allFragmentsReadyFlag` is used as trigger mechanism. This flag is set to ***TRUE*** at the beginning of each iteration of the inner data-taking loop, and set to ***FALSE*** if `FragmentReadyFifo` is empty. Hence, if there is at least one fragment in each `FragmentReadyFifo`, the value of this flag remains ***TRUE*** ("trigger arrived"). The routines are the following:

1. `ArmRorcTrigger()`: it checks the existence of memory banks, checks if the ***rcShm*** flag `Paged data flag` is set, and logs a message.
2. `AsynchReadRorcTrigger()`: it sets the `allFragmentsReadyFlag` to ***TRUE***.
3. `EventArrivedRorcTrigger()`: it returns the value of `allFragmentsReadyFlag`.
4. `ReadEventRorcTrigger()`: it initializes in the base event header the ***eventId*** field (needed for CDH processing) and the ***eventTriggerPattern*** field.
5. `DisArmRorcTrigger()`: this routine is empty.

7.2.4.3 Equipment RorcSplitter

The equipment routines of ***RorcSplitter*** are the following:

1. `ArmRorcSplitter()`: it enables the data splitting mode for the selected channel. If configured via the parameters, it enables the flow control handling.
2. `AsynchReadRorcSplitter()`: this routine is empty.
3. `EventArrivedRorcSplitter()`: this routine is empty.
4. `ReadEventRorcSplitter()`: this routine is empty.
5. `DisArmRorcSplitter()`: it disables the data splitting mode for the selected channel. It also disables the flow control handling.

7.2.4.4 Configuring the RorcData equipment

The various parameters for the equipment ***RorcData*** can be separated into two groups, depending whether they are related to the payload or not. There is a choice between four data sources:

- ***Equipment software*** (parameter ***dataSource*** = 1): events are generated by the RORC equipments without any RORC hardware, thus only by software. In this mode the DATE setup along with the readout program can be tested.
- ***RORC internal data generator*** (parameter ***dataSource*** = 2): events are generated by the RORC internal data generator. In this mode the RORC can be

tested stand-alone.

- ***Front-end emulator*** (parameter ***dataSource*** = 3): events are generated by the ***front-end interface card*** (FEIC). Consult the Web at ***http://cern.ch/ddl*** for the documentation about the FEIC. In this mode the RORC and the DDL chain (SIU, optical fibers, DIU) can be fully tested.
- ***Detector electronics***: (parameter ***dataSource*** = 0): events are generated by the detector electronics. The complete DDL chain starting from the detector electronics is in operation. This mode needs to be chosen for data taking.

Table 7.1 describes the ***RorcData*** equipment parameters that are not related to the payload. The section about the internals of the RORC equipments (see Section 7.2) helps understanding the meaning of these parameters. Typical values can be found in Listing 7.5 (lines 37 and 39). To achieve optimal performance, the parameter ***rorcReadyFifo*** should not exceed 128 and the size of the data pages (given by the parameter ***rorcPageSize***) should be at least the average size of a fragment. The fragment size is limited by the number of pages per fragment (given by the parameter ***fragmentVectorSize***) times the page size. For example in Listing 7.5 (line 37) the maximum fragment size is 1024 * 100000 bytes.

Table 7.1 RorcData equipment parameters for all data sources

Parameter	Description
eqId	equipment id for the equipment header
rorcRevision	1 = pRORC 2 = single channel D-RORC 3 = dual channel D-RORC 4 = dual channel D-RORC with PCI-X interface 5 = dual channel D-RORC with PCI eXpress interface
rorcSerialNb	serial number of the RORC
rorcChannelNb	0 for the pRORC and single channel D-RORC 0 or 1 for the dual channel D-RORC
dataSource	0 = Detector electronics 1 = Equipment software 2 = RORC internal data generator 3 = FEIC
rorcPageSize	data page size in bytes
rorcReadyFifoSize	number of `rorcReadyFifo` entries
fragmentVectorSize	maximum number of pages per fragment
fragmentReadyFifoSize	number of `fragmentReadyFifo` entries
ctrlPtr	for internal use (any value can be chosen)
readyFifoPtr	for internal use (any value can be chosen)

The parameters of the ***RorcData*** equipment that are related to the payload are described in the following tables. The parameters in Table 7.2 refer to the equipment software as data source. At the moment only incremental data can be generated in this mode and the parameters ***rorcRevision***, ***rorcSerialNb***, and

rorcChannel have no meaning. The parameters in Table 7.3 refer to the RORC internal data generator as data source, and the parameters in Table 7.4 refer to the FEIC as data source. Common to these three modes is that an event counter (starting at 1) is generated as the very first data word, which is counted in the fragment size. Moreover, fragments have fixed data size in case the parameters ***dataGenMinSize*** and ***dataGenMaxSize*** are equal, or random data size in case these parameters are different. Finally, the parameters in Table 7.5 refer to the detector electronics as data source.

Table 7.2 RorcData equipment parameters (equipment software)

Parameter	Description
dataGenMinSize	minimum event size in bytes: • fixed / random: minimum is 4 bytes
dataGenMaxSize	maximum event size in bytes: • fixed / random
dataGenInitWord	first incremental data word
dataGenPatternNo	not in use since only incremental data with event counter is generated (any value can be chosen)
dataGenSeed	seed for random generator
expectedCdHVersion	dummy (not checked)
consistencyCheckLevel	0 = no data checks 1 = first and last data word are checked 2 = all data words are checked
consistencyCheckPattern	5 = incremental data with event counter 8 = incremental data without event counter

Table 7.3 RorcData equipment parameters (RORC internal data generator)

Parameter	Description
dataGenMinSize	minimum event size in bytes: • fixed: minimum is 4 bytes, • random: no effect, always 4 bytes
dataGenMaxSize	maximum event size in bytes: • fixed: maximum is 2097152 bytes • random: the value will be rounded to the next lower power of 2, maximum is 2097152 bytes
dataGenInitWord	first incremental / decremental data word, constant or alternating data word
dataGenPatternNo	1 = constant data 2 = alternating pattern 3 = flying 0 4 = flying 1 5 = incremental data 6 = decremental data 7 = random data
dataGenSeed	seed for random generator

Table 7.3 RorcData equipment parameters (RORC internal data generator)

Parameter	Description
expectedCdHVersion	dummy (not checked)
consistencyCheckLevel	0 = no data checks 1 = first and last data word are checked 2 = all data words are checked
consistencyCheckPattern	5 = incremental data with event counter 8 = incremental data without event counter

Table 7.4 RorcData equipment parameters (FEIC)

Parameter	Description
dataGenMinSize	minimum event size in bytes: • fixed: minimum is 64 byte • random: no effect, always 4 bytes
dataGenMaxSize	maximum event size in bytes: • fixed: the value will be rounded to the next lower power of 2, maximum is 1073741824 bytes • random: the value will be rounded to the next lower power of 2, maximum is 1073741824 bytes
dataGenInitWord	not in use since the first incremental data word is always 0 (any value can be chosen)
dataGenPatternNo	1 = external pattern generator 2 = alternating pattern 3 = flying 0 4 = flying 1 5 = incremental data 6 = decremental data
dataGenSeed	seed for random generator
expectedCdHVersion	dummy (not checked)
consistencyCheckLevel	0 = no data checks 1 = first and last data word are checked 2 = all data words are checked
consistencyCheckPattern	5 = incremental data with event counter 8 = incremental data without event counter

Table 7.5 RorcData equipment parameters (detector electronics)

Parameter	Description
dataGenMinSize	not in use (any value can be chosen)
dataGenMaxSize	not in use (any value can be chosen)
dataGenInitWord	not in use (any value can be chosen)
dataGenPatternNo	not in use (any value can be chosen)

Table 7.5 RorcData equipment parameters (detector electronics)

Parameter	Description
dataGenSeed	not in use (any value can be chosen)
expectedCdHVersion	version number of the CDH (current version is 1)
consistencyCheckLevel	0 = no data checks (recommended) 1 = first and last data word are checked 2 = all data words are checked
consistencyCheckPattern	5 = incremental data with event counter 8 = incremental data without event counter

There are several checks implemented in the RORC equipment software that are always applied, for example the length and status of the each delivered data page has to be correct (see Section 7.2). These checks are not related to the payload and if no other checks are desired, the parameter ***consistencyCheckLevel*** must be *0*. However, consistency checks may be applied on payloads having a specific test pattern, if the parameter ***consistencyCheckLevel*** is set to the value *1* or *2*:

- If it is set to *1*, the first and last data word of each page are checked against the pattern given by parameter ***consistencyCheckPattern***. A check of the first data word (event counter) against the DAQ event counter is optional.
- If it is set to *2*, all data words of each page are checked against the pattern given by the parameter ***consistencyCheckPattern***. A check of the first data word (event counter) against the DAQ event counter is optional.

7.2.4.5 Configuring the RorcTrigger equipment

There is only one parameter for the equipment ***RorcTrigger***, called ***EvInterval***. It allows to specify an additional delay interval in microseconds, which can be useful for testing purposes. The default value is 0, as shown in Listing 7.5 (lines 32-33).

7.2.4.6 Configuring the RorcSplitter equipment

A dual channel D-RORC can be used in “split mode”, where the data arriving over the incoming channel is bit-by-bit copied to the other outgoing channel. The parameters of the ***RorcSplitter*** equipment identify the outgoing channel (number 0 or 1) and define how the data flow is handled. After a RORC reset, the split mode is disabled. Table 7.6 describes the ***RorcSplitter*** parameters.

Table 7.6 RorcSplitter equipment parameters

Parameter	Description
rorcSerialNb	serial number of the RORC
rorcChannelNb	0 or 1 defining the outgoing channel

Table 7.6 RorcSplitter equipment parameters

Parameter	Description
rorcFlowControl	0 = the flow control from the receiving side on the outgoing channel is ignored 1 = the flow control from the receiving side on the outgoing channel is taken into account
ctrlPtr	for internal use (any value can be chosen)

7.2.5 Data flow for multiple RORC devices

One LDC can host more than one RORC device with one or two channels, in which case the fragments from each channel need to be built together to constitute the sub-event. Figure 7.4 shows the logical view for three devices to better understand the asynchronous data flow with multiple RORCs in one LDC.

After initializing all elements, each `AsynchReadRorcData()` equipment routine keeps one RORC channel going (***Fill*** and ***Scan*** activity). If a complete fragment has arrived, this routine puts the constructed `FragmentVector`, which points to the attached data pages, into the specific `FragmentReadyFifo`. There is one of these FIFOs for each RORC device in an LDC. If none of them is empty, in which case equipment routine `EventArrivedRorcTrigger()` returns TRUE, one entry is taken out from each `FragmentReadyFifo` and is assembled ina sub-event via the `ReadEventRorcData()` routine. The result of this process is the 1st level vectors of this sub-event. The bank id and offset of this 1st level vector is put into the `readoutReadyFifo`. If the common run parameter ***Common Data Header Present*** flag is set, there are additional consistency checks to verify whether the assembled fragments belong to the same particles collision by analyzing their CDHs (see Section 3.9).

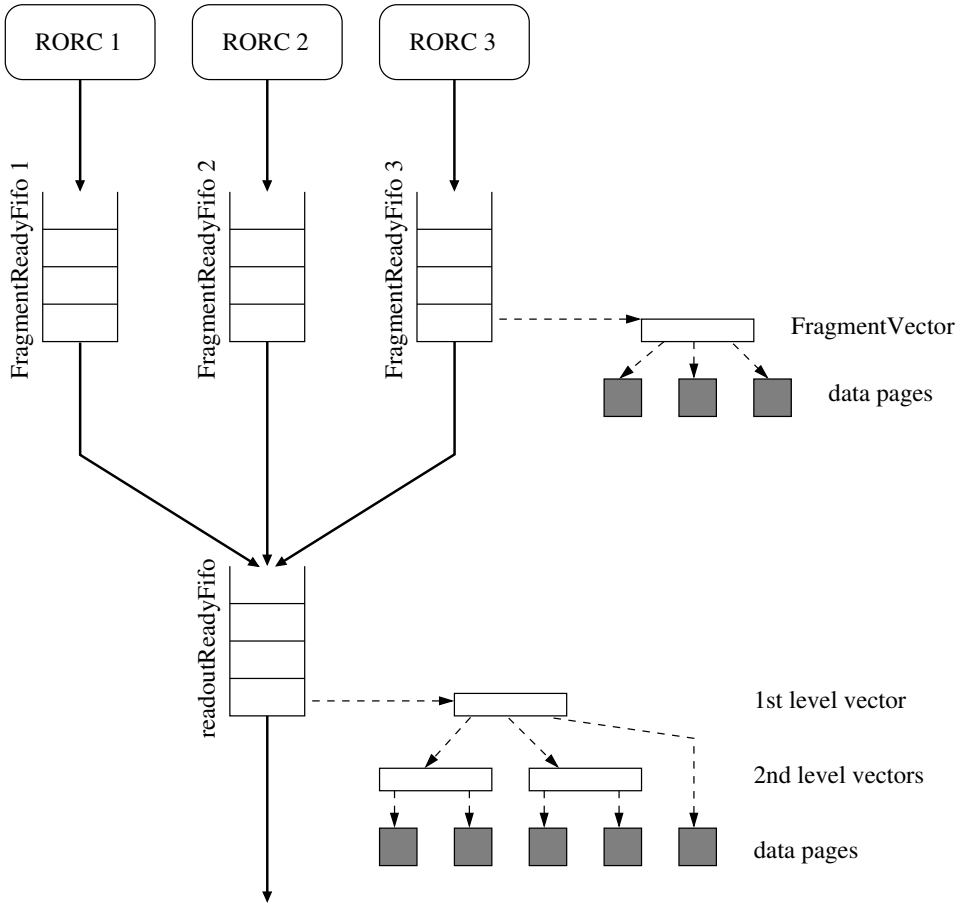

Figure 7.4 The data flow for an LDC with 3 RORC devices

7.2.6 Pseudo code of the RORC equipment routines

The following section presents the pseudo code of the routines `ArmRorcData()`, `AsynchReadRorcData()`, `ReadEventRorcData()`, `DisArmRorcData()`, and for handling FIFOs by a single process. The actual code can be found in the ***${DATE_ROOT}/readList/equipmentList_DDL.c*** file.

Listing 7.1 shows the pseudo code of the routine `ArmRorcData()`. It can be divided in 3 parts. In the first part (line 1), the validity of the equipment parameters (seeSection 7.2.4) is checked. In the second part, the data structures to handle one RORC device are allocated (lines 2-4) and initialized (lines 5-10). As shown in Figure 7.2, these are the `rorcReadyFifo`, the `rorcPageOffsetFifo`, and the `fragmentReadyFifo`. They are allocated from the banks `readoutData` and `readoutFirstLevel`. The indices of these FIFOs are all set to 0 and the flags to ***FALSE***. At this point of time, the contents of these data structures can be ignored. In the third part, the RORC device is initialized (line 11) by calling the ***rorc*** library functions `rorcFind()`, `rorcOpen()`, `rorcReset()`, `rorcArmDDL()`, `rorcStartTrigger()`, `rorcStartDataReceiver()`, whose synopsis is given on the Web site at ***http://cern.ch/ddl***. In the function `rorcStartDataReceiver()`, the physical address of the `rorcReadyFifo` is given to the RORC. The filling with free data pages of the RORC is done by the routine `AsynchReadRorcData()`.

Listing 7.1 Pseudo code of equipment routine `ArmRorcData()`

```
 1:  check the equipment parameters
 2:  allocate a block for rorcReadyFifo from readoutData
 3:  allocate a block for rorcPageOffsetFifo from readoutFirstLevel
 4:  allocate a block for fragmentReadyFifo from readoutFirstLevel
 5:  nextPageInIdx = 0
 6:  nextPageOutIdx = 0
 7:  rorcReadyFifoFull = FALSE
 8:  nextFragmentInIdx = 0
 9:  nextFragmentOutIdx = 0
10:  fragmentFifoFull = FALSE
11:  call functions rorcFind(), rorcOpen(), rorcReset(), rorcArmDDL(),
     rorcStartTrigger(), rorcStartDataReceiver()
```

Listing 7.2 shows the pseudo code of the routine `AsynchReadRorcData()`. Each time this routine is entered, it tries to completely fill the RORC with free data pages in the fill-loop (lines 1-14). This is important for the initialization, when this routine is called the very first time. There are two conditions to exit the fill-loop, either when the `rorcReadyFifo` is full with free data pages (lines 2-4) or the allocation of a free data page was not successful (lines 11-13). In any of these cases the execution can be continued, since allocations will be tried again in the next call. If there are no problems with the allocation of a new data page (line 5), the corresponding status field in the `rorcReadyFifo` is set to -1 (line 7), its bank offset is stored in the `rorcPageOffsetFifo` (line 8), the data page is communicated to the RORC (line 9) via the ***rorc*** library routine `rorcPushFreeFifo()`, and the index `nextPageInIdx` is increased (line 10). After to the fill-loop, the scan-loop (lines 15-55) checks the entries in the `rorcReadyFifo` to find out if the RORC has transferred data pages. The scan always starts at the index `nextPageOutIdx` (line 19) and stops when a status field with -1 is hit (lines 20-22). Since at each advancement of the index `nextPageOutIdx` (line 54) the former status field is set to -1 (lines 53), it is assured that the scan-loop cannot be executed forever. In the special case that the `rorcReadyFifo` is empty (lines 16-18), the scan-loop exits. If the error bit is not set in the status field (lines 23-25), the raw data has been written into this page by the RORC device. This page has to be added to the current `fragmentVector`, which will be allocated (line 27) if this is the beginning of a new fragment (line 26). The handling of a `fragmentVector` is assisted by the index `nextPageInFragmentIdx` to know where the next page entry will be put, and by a variable `fragmentVectorDataSize` to count the total number of bytes of this fragment. Both are initialized to 0 (lines 28-29) when a new `fragmentVector` is allocated. A written data page is put into the current `fragmentVector` by filling its fields (lines 31-35), by advancing the index `nextPageInFragmentIdx` (line 36), and by increasing `fragmentVectorDataSize` (line 37). If the status field indicates a DTSTW that terminates a DDL block (line 38), then only its length is checked. If the status field indicates a DTSTW that terminates a fragment (line 41), then some additional work is done. First the current `fragmentVector` and the last data page of this fragment are resized (lines 42-43), since they might be larger as needed. Then an entry is made for the current `fragmentVector` into the `fragmentReadyFifo` by filling the fields (lines 44-49) and by advancing the index `nextFragmentInIdx` (line 51). After exiting the scan-loop, the `fragmentReadyFifo` is checked (line 56) for emptiness in which case the “trigger arrived” condition is ***FALSE*** (line 57). The equipment ***RorcTrigger*** holds the counterparts in the routine `AsynchReadRorcTrigger()` to initialize this condition to ***TRUE***, and in the routine `EventArrivedRorcTrigger()` to signal this condition.

Listing 7.3 shows the pseudo code of the routine `ReadEventRorcData()`. It takes out the entry (pointer to a fragment) from the `FragmentReadyFifo` to which the index `nextFragmentOutIdx` is pointing (lines 1), and it uses this fragment to fill the equipment header fields (lines 2-7) and the equipment vector fields (lines 8-12). These copying operations construct one equipment entry in the 1st level vector of the sub-event. If the common run parameter ***Common Data Header Present*** flag is set, then the CDH (see Section 7.2.5) of the fragment is processed (lines 13-24), in particular the ***eventId*** field of the base event header is filled. If the CDH processing is switched off, then the software counter ***Number of triggers*** is used for the event identification.

Listing 7.4 shows the pseudo code of the routine `DisArmRorcData()`. First the RORC is stopped by calling the ***rorc*** library routines `rorcStopTrigger()`, `rorcStopDataReceiver()` and `rorcClose()`, whose synopsis is given on the Web site ***http://cern.ch/ddl***. Then all the data structures are de-allocated in the following order: pages in the `rorcReadyFifo`, pages in the `FragmentVector`(s), the `FragmentVector`(s), the `FragmentReadyFifo`, the `rorcPageOffsetFifo`, and the `rorcReadyFifo`.

Finally Listing 7.5 shows the pseudo code to handle a generic FIFO if only one process is using it. Assuming the entries at index `[0,...,maxIdx-1]`, it requires two indices `nextInIdx` and `nextOutIdx` and a flag `fifoFull` to implement the initalize / put / get primitives. This pseudo code applies to `rorcReadyFifo` and `fragmentReadyFifo`.

Listing 7.2 Pseudo code of equipment routine `AsynchReadRorcData()`

```
 1:  begin of fill-loop
 2:    if( rorcReadyFifo is full )
 3:      break fill-loop
 4:    endif
 5:    allocate one data page from readoutData
 6:    if( allocation successful )
 7:      set status field to -1 in rorcReadyFifo[nextPageInIdx]
 8:      put bank offset in rorcPageOffsetFifo[nextPageInIdx]
 9:      call function rorcPushFreeFifo()
10:      advance index nextPageInIdx
11:    else
12:      break fill-loop
13:    endif
14:  end of fill-loop
15:  begin of scan-loop
16:    if( rorcReadyFifo is empty )
17:      break scan-loop
18:    endif
19:    status = status field in rorcReadyFifo[nextPageOutIdx]
20:    if( status == -1 )
21:      break scan-loop
22:    endif
23:    if( status contains error bit )
24:      report error and exit
25:    endif
26:    if( a new fragment )
27:      allocate a fragmentVector from readoutSecondLevel
28:      nextPageInFragmentIdx = 0
29:      fragmentVectorDataSize = 0
30:    endif
31:    fill the fragmentVector[nextPageInFragmentIdx]
32:    - eventVectorBankId field = readoutDataBank
33:    - eventVectorPointsToVector field = FALSE
34:    - eventVectorSize field from length field of
         rorcReadyFifo[nextPageOutIdx]
35:    - eventVectorStartOffset field from
         rorcPageOffsetFifo[nextPageOutIdx]
36:    advance index nextPageInFragmentIdx
37:    increase fragmentVectorDataSize by the eventVectorSize field
38:    if( status is a DTSTW terminating a DDL block)
39:      check the length of the DDL block
40:    endif
41:    if( status is a DTSTW terminating a fragment)
42:      resize the fragmentVector
43:      resize the data page
44:      put an entry into fragmentReadyFifo[nextFragmentInIdx]
45:      - bankId field = readoutSecondLevelBank
46:      - bankOffset field from the fragmentVector
47:      - nbOfPagesInFragment field = nextPageInFragmentIdx
48:      - fragmentDateSize field = fragmentVectorDataSize
49:      - rorcStatus field from the status field of
           rorcReadyFifo[nextPageOutIdx]
50:      update the DDL monitoring fields
51:      advance index nextFragmentInIdx
52:    endif
53:    set status field to -1 in rorcReadyFifo[nextPageOutIdx]
54:    advance index nextPageOutIdx
55:  end of scan-loop
56:  if( fragmentReadyFifo is empty )
57:    allFragmentsReadyFlag = FALSE
58:  endif
```

Listing 7.3 Pseudo code of equipment routine `ReadEventRorcData()`

```
 1:  get fragment from FragmentReadyFifo[nextFragmentOutIdx]
 2:  fill the equipment header
 3:  - equipmentSize from the fragmentDataSize field
 4:  - equipmentType from the equipment parameter
 5:  - equipmentId from the equipment parameter
 6:  - equipmentTypeAttribute from the rorcStatus field
 7:  - equipmentBasicElementSize in bytes
 8:  fill the equipment vector
 9:  - eventVectorBankId from the bankId field
10:  - eventVectorPointsToVector = TRUE
11:  - eventVectorSize from the nbOfPagesInFragment field
12:  - eventVectorStartOffset from the bankOffset field
13:  if( CDH processing )
14:  - version number
15:  - MBZ field
16:  - block length
17:  - status&error bits
18:  - L1 trigger message
19:  - event id
20:  - mini event id
21:  - block attributes
22:  - trigger classes
23:  - participating subdetectors
24:  - ROI
25:  else
26:  - set the event id from software counter
27:  endif
```

Listing 7.4 Pseudo code of equipment routine `DisArmRorcData()`

```
 1:  stop the RORC by calling functions rorcStopTrigger(),
     rorcStopDataReceiver(), rorcClose()
 2:  deallocate all data structures
 3:  - data pages in the rorcReadyFifo
 4:  - data pages in the FragmentVector(s)
 5:  - FragmentVector(s)
 6:  - FragmentReadyFifo
 7:  - rorcPageOffsetFifo
 8:  - rorcReadyFifo
```

Listing 7.5 Pseudo code for handling a FIFO for a single process

```
 1:  // initializing the FIFO
 2:  nextInIdx = 0
 3:  nextOutIdx = 0
 4:  fifoFull = FALSE
 5:
 6:  // putting an element into the FIFO
 7:  if( fifoFull )
 8:    error "FIFO is full"
 9:  else
10:    put FIFO element at index nextInIdx
11:    nextInIdx = nextInIdx + 1 MOD maxIdx
12:    if( nextInIdx == nextOutIdx )
13:      fifoFull = TRUE
14:    endif
15:  endif
16:
17:  // getting an element from the FIFO
18:  if( nextInIdx == nextOutIdx && NOT fifoFull )
19:    error "FIFO is empty"
20:  else
21:    get FIFO element at index nextOutIdx
22:    nextOutIdx = nextOutIdx + 1 MOD maxIdx
23:    fifoFull = FALSE
24:  endif
```

7.3 Introduction to the UDP equipment

The Ethernet socket has been added in DATE as an alternative data source. The UDP equipment reads data coming from the Ethernet port of a PC using the UDP protocol. The readout UDP consists of one Ethernet port used by the front-end electronics to send data, a second port used by readout to receive data and one Ethernet cable that connects the two ports. It can be a copper or an optical fiber cable. Depending on the hardware used it is possible to obtain a data throughput from 1 Gb/s up to 10 Gb/s. Depending on the acquisition needs and on the number of available Ethernet sockets, a PC can be equipped with several UDP equipments (up to 3 Ethernet ports in one PC have been tested so far).

DATE provides all the necessary readout software to operate the Ethernet port on a PC running Linux via the driver of the network card. The following sections concentrate on the equipment software for the UPD readout.

7.4 Internals of the UDP equipment

The goal of the UDP equipment is to read data from one or more LDC Ethernet ports. The front end electronics send data using the UDP protocol, packing events in frames of a maximum size of 9KB. The UDP readout software has to be structured in equipment routines as explained in Chapter 6.

7.4.1 Data transfer mechanism of the UDP equipment

The mechanism to transfer data from an Ethernet socket to the memory of the PC has been inherited from the RORC algorithm and requires the following activities:

1. **`Read`**: the process reads data, if any, from the UDP receiver buffer. A counter is increased every time a packet has been read by this process. When the counter reaches a value equal to the maximum number of packets that the UDP receiver buffer can accept, the process checks if the buffer is empty. If the buffer is empty it sends a word to the front end electronics asking for more data (see Section 7.4.2).
2. **`Transfer`**: the process transfers data from the Ethernet socket to the data page if the `rorcSimulatorFreePages` is bigger than 0. This can only take place if there is data arriving from the socket and if the `rorcReadyFifo` is not full. When the data transfer is completed, the process fills the `rorcReadyFifo` with information about the transfer. The `rorcReadyFifo` is located in the memory of the PC and has 128 entries. Each of these entries consists of two fields: the length (32 bits) in words of the transferred data, and the transfer status (32 bits). The status field can be either a DTSTW (Data Transmission Status Word) or 0 if more pages are about to follow. A DTSTW marks the end of a sub event. Whenever a free data page with a particular index is used by the process during a fill activity, the status field of this indexed `rorcReadyFifo` entry has to be initialized to -1.

7.4.2 The back-pressure algorithm

UDP uses a simple transmission model without explicit hand-shaking dialogues to guarantee reliability, ordering and data integrity. Error checking and correction must be performed in the application. The UDP readout equipment implements a software back-pressure to avoid an overflow of the socket receiving buffer. Figure 7.5 shows the behavior of the back-pressure algorithm. The board will send data to the DAQ system at the maximum speed until a fixed number of packet have been sent. This number of packets can be calculated using the following formula:

fixed number of packets = SOCKET RECV BUF SIZE / MAX UDP PACKET SIZE

Once the number of packets sent reaches the number of packets expected, the detector electronics enters in an idle loop waiting for a specific word coming from the readout software. Once received, the board continues sending data stored in its buffer, if any is present.

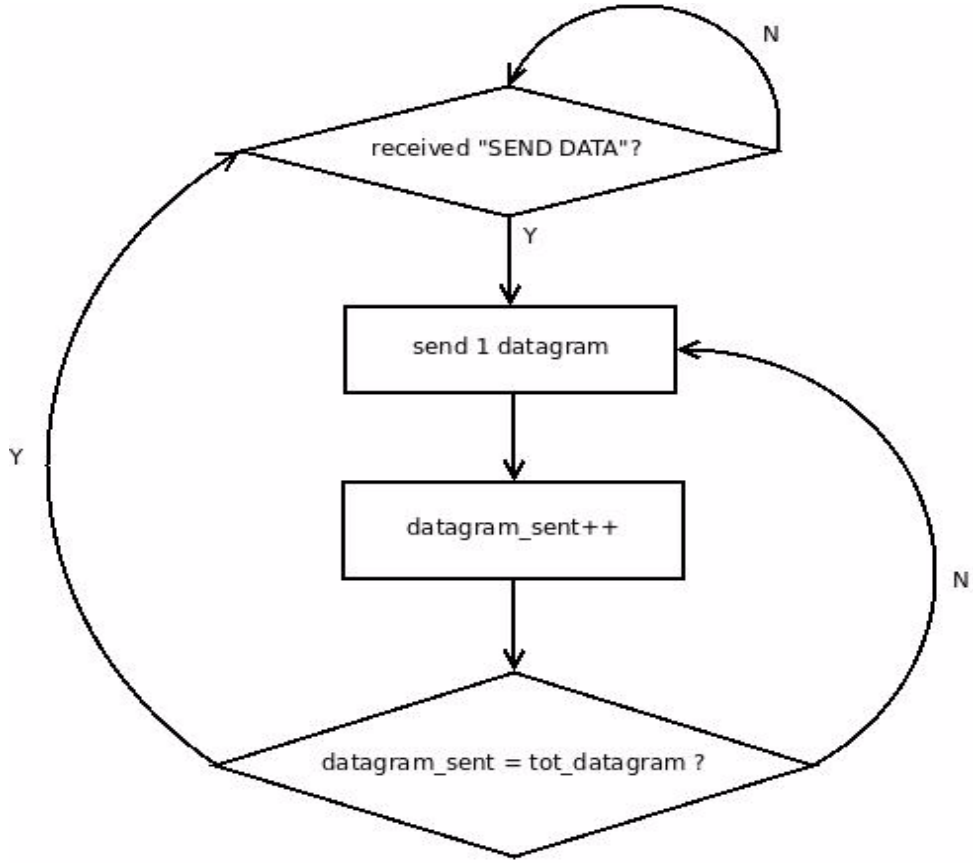

Figure 7.5 The back-pressure algorithm.

7.4.3 Equipments to handle the Ethernet port

The UDP readout software is implemented by two equipment types in the ***UDP*** equipment suite which is provided in the package **`readList`** (see Chapter 6):

- ***`RorcDataUDP`*** is responsible for initializing the socket and to handle the data packets from the Ethernet port. One such equipment needs to be instantiated for each Ethernet socket in an LDC. It has the attribute GENDATA and can be configured by several parameters (see Section 7.2.4.1).
- ***`RorcTrigger`*** is responsible for indicating the availability of a sub-event, where each port contributes a fragment. Exactly one such equipment for each LDC needs to be instantiated. It has the attribute TRIGGER and can be configured by one parameter (see Section 7.2.4.2).

The equipment routines are participating in the construction of a sub-event in paged mode (see Chapter 3), as shown in Figure 7.3. The sub-event is described by a 1st level vector, which is composed of the base header and three equipments. Each of the equipments is represented by an equipment header and an equipment vector. The equipment vector of the first equipment points via a pair <bank id, bank offset> to a 2nd level vector, which is composed of three payload vectors in sequence. Each vector points again via a pair <bank id, bank offset> to one data page. The equipment vector of the second equipment points to a 2nd level vector

with two payload vectors. The equipment vector of the third equipment directly points to one data page via a pair <bank id, bank offset>. As an option, an equipment vector may always point to a 2nd level vector, even if it contains only one payload vector.

7.4.3.1 Equipment RorcDataUDP

The equipment routines of ***RorcDataUDP*** for reading data from one port are the following:

1. `ArmRorcDataUDP()`: it checks equipment parameters and logs a message. It allocates and initializes the `rorcPageOffsetFifo`, the `rorcReadyFifo`, and the `FragmentReadyFifo`. It opens the socket connected to the Ethernet port.
2. `AsynchReadRorcDataUDP()`: it reads data stored in the socket receiving buffer and copies them into the memory of the PC.
3. `EventArrivedRorcDataUDP()`: this routine is empty.
4. `ReadEventRorcDataUDP()`: it takes out a fragment from the `FragmentReadyFifo` and uses it to fill the equipment header and equipment vector of the 1st level vector.
5. `DisArmRorcDataUDP()`: it closes the socket and de-allocates all memory blocks.

7.4.3.2 Equipment RorcTriggerUDP

The equipment routines of ***RorcTriggerUDP*** are used for triggering the Ethernet port. The global flag `allFragmentsReadyFlag` is used as trigger mechanism. This flag is set to TRUE at the beginning of each iteration of the inner data-taking loop, and set to FALSE if `FragmentReadyFifo` is empty. Hence, if there is at least one fragment in each `FragmentReadyFifo`, the value of this flag remains TRUE (“trigger arrived”). The routines are the following:

1. `ArmRorcTriggerUDP()`: it checks the existence of memory banks, checks if the ***rcShm*** flag `Paged data flag` is set, and logs a message.
2. `AsynchReadRorcTriggerUDP()`: it sets the `allFragmentsReadyFlag` to ***TRUE***.
3. `EventArrivedRorcTriggerUDP()`: it returns the value of `allFragmentsReadyFlag`.
4. `ReadEventRorcTriggerUDP()`: it initializes in the base event header the ***eventId*** field (needed for CDH processing) and the ***eventTriggerPattern*** field.
5. `DisArmRorcTriggerUDP()`: this routine is empty.

7.4.4 Data flow for multiple UDP equipments

One LDC can host more than one Ethernet port, in which case the fragments from each channel need to be built together to constitute the sub-event. To understand the asynchronous data flow, see Figure 7.4 showing the logical view for three devices.

8 The trigger system

In the ALICE DAQ system, the detector readout is based on the DDL or the Ethernet/UDP link. The trigger mainly interacts with the detectors, while DATE accepts a continuous flow of data. The DATE software is self-triggered by the availability of complete sub-events in the LDC memory.

This chapter discusses the trigger requirements of DATE and gives some indications on how to set up the trigger system.

8.1 The trigger system

The ALICE trigger is designed for two different types of beams: Pb-Pb beams with 125 ns bunch crossings and pp beams with 25 ns bunch crossings. The trigger system identifies the events that are supposedly worth to be read out and activates their readout.

Triggering the data–acquisition system is a complex operation that involves a variety of actions, such as sending signals to each detector with the proper timing, activate the ***readout*** processes and distributing some information about the event (e.g. event identification). In ALICE, different types of triggers are generated, involving different sets of LDCs. DATE is made to cope with this set of requirements.

8.1.1 The Central Trigger Processor (CTP)

The general architecture of the ALICE Trigger is shown in Figure 8.1. The Central Trigger Processor (CTP) receives the input from the trigger detectors and the LHC clock from the TTC Machine Interface (TTCmi). For every bunch crossing, and according to the busy status of all the detectors, the CTP produces trigger decisions which are transmitted to every detector via its own Locat Trigger Unit (LTU). The LTU converts these decisions into messages which are distributed to the detector electronics via the TTC broadcast system thanks to the TTCvi and TTCex modules. More information about the ALICE Trigger and the TTC system can be found respectively in Ref. [11] and [12].

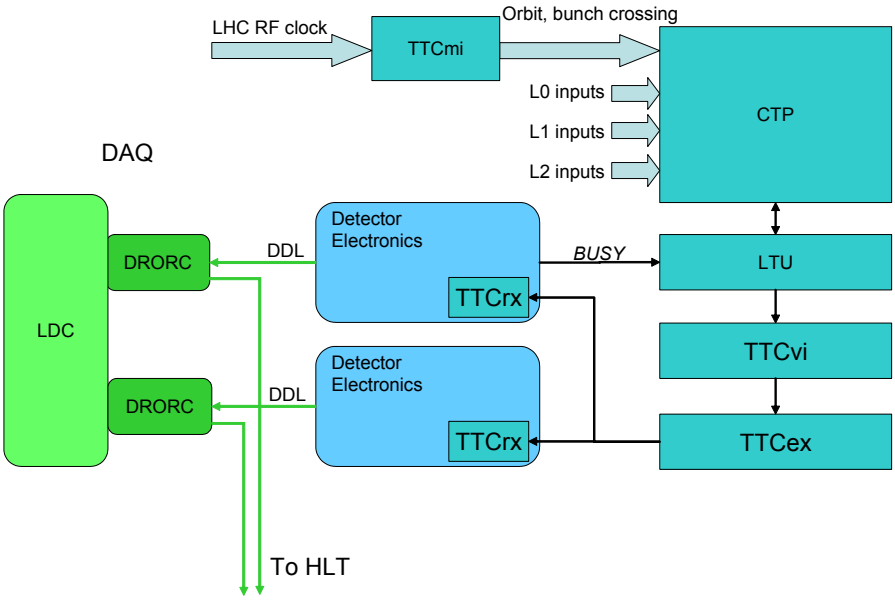

Figure 8.1 ALICE Trigger.

The information transmitted by the TTC messages include trigger information (trigger class for physics triggers and list of detector for software triggers), and a ***unique*** event identification (orbit number and bunch crossing).

In DATE, the major requirement for the trigger system is to inform the LDCs of the availability of the next event, in such a way that they can collect the sub-events in a synchronous way. The synchronization of the LDCs is implemented in DATE by a mechanism that recognizes the events, based on the ***readout*** program (see Chapter 6).

The processing of the ***readout*** program consists of a series of operations made in a tight loop. One of them polls the ***EventArrived*** routine, which provides the event synchronization. The ***EventArrived*** strongly depends on the synchronization method adopted.

The synchronization of the LDCs is achieved by a data–driven mechanism (see Section 8.2). The DDL injects data in the LDC memory in an autonomous way. The DDL data structure keeps the knowledge of the original blocks generated by the detector and marks the boundary of them. The arrival of a new block is notified to the ***readout*** software and is assumed to be a new event.

Since the LDCs work independently (there is no communication between them), it is important that they receive an ordered sequence of triggers. Keeping the order of the sub-events collected by each LDC is essential for the event builders which subsequently assemble the sub-events into a full event. Independent verification mechanisms are put in place to catch the occurrence of an LDC losing a sub-event. One of these mechanisms is based on the ***unique*** event identifier (orbit number and bunch crossing) that is transmitted by the trigger system. This identifier must be transmitted to an electronic module in order to be included in the sub-events by the data source.

DATE uses the event identifier located in the sub-event header to perform various consistency checks, which are made both by ***readout*** in the LDCs and by the ***eventBuilder*** in the GDCs.

8.2 LDC synchronization via the equipments

Two types of readout links are supported by DATE: the DDL and the Ethernet / UDP links.

The DDLs are read by the LDCs via the Read-Out Receiver Card (RORC). Two types of RORCs are presently supported by the ALICE DAQ:

- The dual channel D-RORC interfaces two DDL channels with embedded DIUs to a 64 bit / 64 MHz PCI bus.
- The dual channel D-RORC interfaces two DDL channels with embedded DIUs to a PCI Express (PCIe) bus.

The Ethernet / UDP links are read by using the Gigabit Ethernet or 10 Gigabit Ethernet interface available on the PC Motherboard or added as a PCI or PCIe add-on board.

In this section these different versions of RORC cards and of Ethernet interfaces will all be commonly referred as ***equipments***, since they are exactly identical from the DATE / trigger interface point of view.

The ***equipmentList*** software contains several equipments which handle the RORC hardware and the Ethernet link. Two of these equipments are relevant for the trigger handling. Their behaviour is briefly described here. More details on the operation of them can be found in Chapter 6.

The equipment injects a continuous stream of event fragments into the LDC memory in an autonomous way. There may be several equipments in an LDC; each of them owns an instantiation of the ***RorcData*** equipment for the RORC or ***RorcDataUDP*** equipment for the Ethernet / UDP equipment, which keeps a list of the fragments arrived. The ***RorcData*** equipment keeps the list of available data fragments (`FragmentReadyFifo`) up to date . It also updates the global flag `allFragmentsReadyFlag` by setting it to FALSE if its own `FragmentReadyFifo` is empty.

The ***RorcTrigger*** equipment is unique in each LDC. It is used for triggering the readout of one or more RORC or Ethernet / UDPchannel(s). The global flag `allFragmentsReadyFlag` is used as trigger mechanism. This flag is set to TRUE at the beginning of each iteration of the inner data- taking-loop, and set to FALSE in case of an any empty `FragmentReadyFifo`. Hence, if there is at least one fragment in each `FragmentReadyFifo`, the value of this flag remains TRUE (“trigger arrived”). The readout software is informed when a sub-event is complete and ready to be acquired. The sub-event is ready for ***readout*** when all the RORCs or Ethernet / UDPchannels have received all the fragments belonging to an event and the fragments have been joined into a sub-event.

9

COLE - *CO*nfigurable *L*DC *E*mulator

COLE (***Co***nfigurable ***L***DC ***E***mulator) is designed to create ALICE-like events according to a simple user-defined configuration file. It replaces the standard DATE ***readList*** and follows the directions given in the DATE configuration files and databases.

9.1 Introduction

COLE (***CO***nfigurable ***L***DC ***E***mulator) is designed to create ALICE-like events according to a simple user-defined configuration file. It is a replacement for the standard DATE ***`readList`*** module that is compiled and linked with the readout program. The aims of COLE are to provide complete and flexible control over the structure of the input stream to DATE and the functionality to reconfigure the readout program without the need for re-compilation. The COLE configuration file is used to define details of the events to be created by the readout program, event by event. COLE features include:

1. configurable / flexible:
   - no re-compilation required to reconfigure DATE stream.
   - fully scalable.
   - a single configuration file controls all of COLE.
   - integrated with ALICE DATE databases used for the DATE configuration (detector, trigger, event building policies).
   - support for streamlined and paged event mode.
2. pre-defined event stream:
   - different event types.
   - global synchronized event ID.
   - pre-set trigger / detector patterns.
3. simulated ALICE trigger classes:
   - support for partial event building and use of trigger / detector patterns to create events.
   - non-global events (to perform partial event building).
   - use of pre-configured trigger patterns to create events.
4. simulate ALICE raw data:
   - configurable on an event-by-event, equipment-by-equipment and host-by-host basis.
5. simulate trigger and detector delays:
   - possible to simulate delays on a detector-by-detector and trigger-by-trigger basis.
6. simulate burst mode structure:
   - test-beam like data traffic.
   - creates bursts of a given number of events defined in common run parameters.
   - control of burst number and number in burst.
7. DATE Equipment:
   - COLE is fully implemented as a standard DATE equipment.

9.2 Delayed mode vs. free-running mode

As COLE is mainly used to emulate DAQ systems, it may suffer from the absence of a real triggering system. LDCs may go out of synch and find themselves hundreds of events away, as a function of the relative loads on individual machines and on the event building network. For this reason the delayed mode was introduced. When running in delayed mode, COLE waits for a given delay between events. Synchronization between LDCs is done at start-of-run. Jitters may still occur due to the absence of a trigger system as a function of the actual time of the start-of-run for each LDC and to different load on the LDCs. The LDCs will keep track of the cumulated delays and - if needed - will try to re-synchronize whenever possible. A threshold can be set to generate warning messages coming from LDCs that cannot keep up with the requested timing. Delays can be specified on an event-by-event basis and on a detector-by-detector basis. All delays are given in microseconds.

9.3 System requirements and configuration

DATE must be installed and available on all hosts that are to use COLE. The data-acquisition system must be correctly configured. In addition the following files must be created:

a. **event payloads:** stored in ***${DATE_SITE}/${HOSTNAME}*** they contain the payload associated to all the events created by COLE during a single run. One LDC can have multiple payloads associated to multiple events. More events can share if needed the same payload file. More LDCs may share the same payload files by means of Unix symbolic links.

b. **COLE configuration**: specified by the symbol ***${DATE_COLE_CONFIG}*** it allows the control of the stream created by COLE. It consists of an ASCII file divided into three sections:

   - ***Options***: global options used for all events.
   - ***Events***: the stream of events created by the DATE system as a whole.
   - ***Detectors***: detector-specific parameters.

   An example of COLE configuration is given in Listing 9.1.

Listing 9.1 Example of COLE configuration:

```
 1:  >Options_section
 2:  UseDelay       1
 3:  UseRandomEvent 1 12321
 4:  Threshold      1000
 5:  >Events_section
 6:  trdDetector CAL  *              10+4  raw2 40
 7:  *           PHYS trdTrigger     5     raw1 40
 8:  *           PHYS centralTrigger 1+2+3 raw3 40
 9:  >Detectors_section
10:  trdDetector 20 12
11:  tpcDetector 30 13
12:  itsDetector 40
```

The available options are:

- ***UseDelay***: delayed emulation mode active (1) or inactive (0).
- ***UseRandomEvent***: directs COLE to create events according to the given list (0, default) or to create pseudo-random events (1): in the second case an optional initial seed for the pseudo-random number generator can be provided (default: 12345).
- ***Threshold***: delay (in microseconds) used to issue warning messages when the LDC is unable to keep the requested delay between events (delayed emulation mode only).

The events section defines the events as they are created by the DATE system as a whole. Events are specified as:

- detector pattern: the detector(s) that participate to the event ("*": no detector pattern). This list could also contain names of individual LDCs. Mixing detector(s) and LDC(s) is not allowed. Multiple names must be separated by "+".
- event type (***SOD*** for start of data events, ***EOD*** for end of data events, ***PHYS*** for physics events, ***CAL*** for calibration events, ***SST*** for system software trigger events or ***DST*** for detector software trigger events).
- trigger pattern associated to the event ("*": no trigger pattern). Multiple trigger classes (separated by "+") can be specified.
- attributes set in the event ("*": no attributes set). Attributes are specified by value. Multiple attributes can be given separated by "+".
- payload of the event (path relative to ***${DATE_SITE}/${DATE_HOSTNAME}***) The filename is combined with the *equipmentId* to create a unique, ***per-equipment*** filename. If the filename has no extension, COLE will append ***_equipmentId*** to it (e.g. for equipment ***123***, ***coleData*** becomes ***coleData_123***). If the filename has an extension, COLE will insert ***_equipmentId*** between the base name and the extension (e.g. for equipment ***123***, ***coleData.raw*** becomes ***coleData_123.raw***).
- delay (in microseconds) for the generation of the event (0: no delay). If two values are given, a pseudo-random number will be extracted within the given range. This number will be the same for all the LDCs belonging to the same detector.

The LDCs participating in a data-acquisition system will loop on the given list. For each line (in sequence or following a pseudo-random sequence) encountered, they will wait for the given delay. When they are supposed to contribute to the event, they will first wait for the detector-specific delay and then create their sub-event with the given parameters and payload. Events can be driven on a detector basis or on a trigger basis. In the first case, a list of detectors must be given and the trigger pattern must contain "*". In the second case the detector pattern must contain "*". In both cases, only the LDCs supposed to create events will do so.

The detector section specifies for each detector the delay (or the range of delays) to be applied to all events coming from LDCs from this detector. One number corresponds to a fixed delay. Two numbers will instruct COLE to draw a pseudo-random number within the given range and use that as a delay.

COLE can simulate a burst structure. This is done when the common run parameter ***burstPresentFlag*** is set. A burst will be closed when more than ***simBurstLength*** events will have been created at the level of the full DAQ system (less events may have been created at the individual LDC level according to the triggering criteria).

9.4 COLE as an Equipment

COLE is fully implemented as an equipment and can be used in both streamlined and paged event mode. The equipment configuration file must be correctly defined to use COLE as an equipment. An example file is shown in Listing 9.2. Cole will fill the ***equipmentHeader*** structure as any other standard DATE equipment (basic element size will be set to 4).

Listing 9.2 Example of COLE equipment configuration:

```
1:  >EQTYPES
2:  >Cole 1 TRIGGER GENDATA
3:  EqId %hd
4:  >LDCS
5:  >host1
6:  + Cole firstCole 1
7:  >host2
8:  + Cole secondCole 2
```

9.5 Basic Design

COLE will use and extend the basic structure of the four functions required for each DATE equipment.

9.5.1 `ArmHw()`

Called at start of run to perform system initialization such as loading the detectors, triggers and roles databases. `ArmHw` will also parse the COLE configuration file and load all the payloads.

9.5.2 `EventArrived()`

Simulates the trigger delay used for the delayed emulation mode. A simple state machine will implement the emulation of the trigger and detector delays.

9.5.3 ReadEvent()

Called in the main event loop after the arrival of a trigger to perform the readout of the hardware. It fills in the event header information as defined in the ***${DATE_COLE_CONFIG}*** configuration file. The following fields are handled:

- ***nbInRun*** - event number within run. This is the unique number identifying the event and must be set. This value is used by the event builder and must increase for each event.
- ***burstNb*** - burst number; initialized to 0 by ***readout***.
- ***nbInBurst*** - event number within the burst (starts at 0 at each start-of-burst event).
- ***event type*** - physics, calibration, start-of-burst, end-of-burst.
- ***trigger pattern***.
- ***detector pattern***.
- ***user attributes***.

This function also loads the payload of the event.

9.5.4 DisArmHw()

Called at each end of run to perform rundown. Also used to unload the databases upon completion and to free the dynamic memory allocated by COLE.

9.6 The colecheck utility

The `colecheck` command line utility is available to validate the structure of a ***${DATE_COLE_CONFIG}*** file and to display a summary of any errors found.

Command line syntax:

`colecheck -n <`**`hostname`**`> -f <`**`COLE config file path`**`> [-q] [-h]`

- <**hostname**>: the name of the host you wish to check for in the cole.config file.
- <**COLE config file path**>: the full path to the COLE configuration file.
- `[-q]`: quiet mode (only check for errors).
- `[-h]`: print usage and exit.

`colecheck` checks the syntax of all events defined in the given COLE configuration file and checks whether the **hostname** machine contributes to each event. If no hostname is given, `colecheck` only checks the syntax of the ***${DATE_COLE_CONFIG}*** file.

Data recording 10

This chapter describes the data recording process and how the data can be recorded in the LDCs and in the GDCs. It explains also the conventions concerning the filenames of the data streams that can be created using DATE.

10.1 Introduction

Data recording can be done either on LDCs or on GDCs. A common library is used by all DATE actors doing data recording (local or remote), the same features are therefore available throughout the whole DATE system.

The basic GDC recording functionality can be enhanced by using a dedicated DATE component – the ***mStreamRecorder*** (MSR).

This chapter describes the configuration and the behavior of the data recording process, both common and DATE-role specific.

10.2 Common data recording procedures

The data recording process uses the ***recordingDevice*** runParameter, specified via the ***runControl***. This string can be suffixed by a special character used to define the type of recording channel. The string can specify an arbitrary number of output channels all of the same type (no mixture of different channels is allowed within the same run). These channels are then handled according to the various configuration parameters concerning file size and maximum amount of data to be recorded during the run. It is also possible to include in the file name special characters, to be translated at run-time into machine-specific and run-specific strings.

The data generated in an LDC can be recorded:

- to a (set of) local disk file(s) (no suffix needed).
- to a (set of) local named pipe(s) (suffix: "**|**").
- by sending them to a (set of) GDC(s) (suffix: "***:***").

The data generated in a GDC can be recorded:

- to a (set of) local disk file(s) (no suffix needed).
- to a (set of) local named pipe(s) (suffix: "**|**").
- using an external recording process ("***:***" as ***recordingDevice***).

The ***recorder*** and the ***eventBuilder*** processes can store their events on the local machine (either to a file or to a named pipe). In this case, the full path of the output stream has to be specified in the ***recordingDevice*** runParameter, e.g.:

```
/tmp/my_raw_data.dat
/tmp/my_pipe|
```

To record on a file, the directory in which the file resides should have write access for "***${DATE_USER_ID}***"; one can either give write access for this user (or for the whole world) to the directory or set the ownership of the directory. Files are created (and possibly overwritten) according to the status of the data-acquisition system. A list of comma-separated files can be given as recording device, e.g. "`/tmp/a,/tmp/b,/tmp/c`".

To record onto a named pipe, this must be created - before starting the run - via the Unix command `mknod`. The appropriate file protection must be set to allow user "***${DATE_USER_ID}***" to write into the pipe and to be able to read the pipe via whatever daemon is required. The filename must be suffixed by "|" (removed from the ***recordingDevice*** string to derive the real name of the pipe). Special care must be taken in this recording mode as the absence or the unexpected termination of the data consumer may stall the data-taking process. Some mechanism external to DATE must be implemented to avoid this scenario.

To record with an external recording process, the ***eventBuilder*** must be running in the "online recording" mode. The recording process, launched at start of run, must use the API provided within the DATE ***eventBuilder*** package (Section 10.4.2). DATE provides the online recording application MSR, described in Section 10.5.

If the DAQ system includes one or more GDCs, the ***recorder*** process on the LDC can send the data to them. In this case, the recording device name must be the host name(s) of the event-builder machine(s) separated by "***:***", e.g.:

```
GDC01:
GDC01:GDC02:GDC03:
```

In the above example, the machine GDC01 receives all events when the first string is used while the machines GDC01, GDC02 and GDC03 receive about 1/3 of the events when the second string is used.

For multiple-GDC environments, the algorithm currently implemented in the LDCs sends all the non-physics events to the first GDC of the list (in the example above: GDC01) and then distributes the physics and the calibration events between all the GDCs of the list, using an algorithm based on the event number and on the decisions taken from the EDM (when the EDM is active).

It is possible to limit the total amount of information to be recorded in a run by setting the run parameters ***maxBytes***, ***maxEvents***, and ***maxBursts***. It is also possible to limit the maximum file size (excluding the case of a network channel) using the run parameter ***maxFileSize*** (in this case, special care should be used when the output device is a named pipe: the consumer must be capable to handle the EOF event correctly).

When recording on local file, it is recommended to use a per host, per run file. To allow automatic generation of unique file names based on those parameters, the recording library allows the use of some special characters, namely:

- "`@`" is replaced by the current host name.
- "`$`" is replaced by the current role name.
- "`!`" is replaced by the current role ID.
- "`#`" is replaced by the current run number.

For example, using the ***recordingDevice***:

```
/data/run_#.raw
```

the data of the run 1020 is recorded into the file `/data/run_1020.raw` (assuming there is no limit on the file size). The recording library replaces the ***first occurrence*** (left to right) of the special characters with the corresponding run-time value.

If there is a limit on the maximum file size, the data will be recorded to a sequence of files. Their filenames will be formed by the addition of the original filename for this run and a sequential number (preserving the file extension, if any). In the example above, the following files are created:

```
/data/run_1020.000.raw
/data/run_1020.001.raw
/data/run_1020.002.raw
```

The file with sequential number “000” can be reserved (when the appropriate runtime parameter is set) for the data recorded during the start of run phase. In this case, the first file includes the records of the types ***START_OF_RUN*** and ***START_OF_RUN_FILES*** and it is closed as soon as all the ***START_OF_RUN*** record(s) tagged with ***ATTR_P_END*** attribute have been recorded.

When writing to a set of identical local devices (files, tapes, pipes, etc.), the recording library sends events to the first channel found available (as seen from the Operating System output library). The decision if a channel is busy or not is taken the moment the request to write an event is issued from the data producer (***recorder*** or ***eventBuilder***). The actual destination of a given event is therefore a function of the Operating System and of the device itself and - in general - cannot be predicted beforehand.

10.3 Recording from the LDC

The processes that are always running in an LDC during the data taking phase are the ***readout*** process (see Chapter 6) for receiving the event fragments from the detector electronics, and the ***recorder*** process for moving the assembled sub-events either to local storage devices or to the GDC machines over the event building network. In the following the ***recorder*** process will be presented in more details, in particular its recording capabilities. The source code of it is located in the ***readout*** package.

The ***recorder*** process performs the following sequence of operations in the order described below:

1. maps to all memory banks that are configured for this LDC in the banks database (see Chapter 4).
2. saves its own process ID in the shared memory control region, which allows the ***readout*** process to suspend and to resume the ***recorder*** process.
3. opens file(s) on local storage device(s) or connects to remote GDC machine(s) depending on the LDC run parameter ***recordingDevice***, which is fully explained in Section 10.2:
   - if the name of the recording device does not terminate with “***:***”, then the ***recorder*** process writes all sub-events to files on the local storage device. This is typically the case when the data-acquisition system is composed by a single LDC without event building.
   - if the name of the recording device does terminate with “***:***”, then the ***recorder*** process takes it as the name of a remote GDC machine and opens

a TCP / IP socket connection for transmitting the sub-events. In this case the data-acquisition system has event building provided by one or more GDC machines.

4. enters the event loop, in which the event descriptors are taken out from the ***recorderInput*** FIFO and each sub-event is either written on a file or is sent over the event building network to a GDC, depending on the recording device. All recording operations are done by calling routines of the high level library from the ***recordingLib*** package (see Section 16.4). An event descriptor points either to a streamlined or paged sub-event (see Chapter 3). After a successful recording of a sub-event, the ***recordingLib*** routines also take care to update the relevant run-time parameters and to deallocate the associated memory blocks of the sub-event.
5. closes the local file or the socket connection(s), after exiting the event loop.

In the event loop the ***recorder*** process continuously checks for the arrival of the end of run command. It exits the event loop if one of the following conditions are met:

- the maximum number of bytes to be written (given by the LDC run parameter ***maxBytes***) has already been reached.
- there have been too many errors in writing the file or in the transfer over the event building network.
- the operator asked to stop the run.
- a fatal error occurred, which is indicated by a non-zero value of the ***runEndCondition*** variable in the shared memory control region.

In the first three cases the ***recorder*** process tries to write all the pending sub-events (represented by their event descriptors) in the ***recorderInput*** FIFO onto the recording device before exiting, whereas in the last case the ***recorder*** process does not empty the ***recorderInput*** FIFO before exiting. In the simplest case the ***readout*** process is the producer of the event descriptors, thus the ***recorderInput*** equals the ***readoutReadyFifo***.

The ***recorder*** process uses the ***infoLogger*** package to report and trace error or abnormal conditions, and to trace state changes. The operator can tailor these features to the required needs by setting the value of the LDC run parameter ***logLevel***. Output messages produced by the ***recorder*** process are sent to ***${DATE_SITE_TMP}/${DATE_ROLENAME}/recorder/recorder.log***. The same directory will also store the core files that may be created by the ***recorder*** process in case it gets terminated by an unrecoverable signal. The ***recorder*** process runs as the user defined in the DAQ configuration database. Directories protections and ownerships must be set accordingly.

10.4 Recording from the eventBuilder

Recording on the GDC begins at the level of the ***eventBuilder*** process. Two options are available at this stage: direct recording, where the ***eventBuilder*** writes the raw events directly to a given local device, or online recording, where the events are transmitted to a further stage, for handling and recording. The two

schemes are exclusive on the same GDC while different GDCs can use different schemes and/or different run-time parameters.

10.4.1 Direct recording

The ***eventBuilder*** can record data directly using the DATE ***recordingLib*** package (the same package used by the ***recorder*** process on the LDCs). If multiple output streams are specified, the ***eventBuilder*** uses the first available recording channel. This channel is ensured to be able to accept a new I/O request (although it is not sure if it is able to complete it). All options made available by the DATE recording library are available on the GDCs.

10.4.2 Online recording

The ***eventBuilder*** can transfer "ready" events to an optional processing stage. Events are moved using an internal format and can be accessed by another process using a copy-less, memory mapped access scheme. Only one process can attach itself to the output of the ***eventBuilder*** during a given run. This process must use the API provided within the DATE ***eventBuilder*** package.

Online recording can be activated giving "***:***" as recording device.

The resource ***eventBuilderReadyFifo*** must be declared for the GDC in the banks database in order to be able to perform online recording.

Events are normally transmitted from the ***eventBuilder*** to the requester using `iovec` structures. These structures are standard entities used by Unix I/O libraries and cannot be shared across processes. When the requirement to transfer these vectors to other processes exists, then ***event descriptors*** should be used. Event descriptors are process-independent entities and can be transformed at any time to equivalent `iovec` structures (which are process dependent). Event descriptors are allocated in the process' local address space and therefore need external mechanisms for their transfer to other processes (e.g. shared message queues or shared memory blocks).

An API is available for C and C++. The file ***${DATE_EB_DIR}/libDateEb.h*** should be included (use the "`-I ${DATE_EB_DIR}`" compiler directive) and the library ***${DATE_EB_BIN}/libDateEb.a*** should be used during the link phase. Please note that several other include files and libraries are needed for a successful compilation: refer to the file ***${DATE_EB_DIR}/simpleConsumer.c*** and the associated rules within ***${DATE_EB_DIR}/GNUmakefile*** for a complete list.

---

struct iovec

C Synopsis

```
#include <sys/uio.h>

struct iovec { ... }
```

Description Structure used to describe an event created by the ***eventBuilder***. For the actual implementation of the structure, refer to the system include files and/or to the

relative manual pages (e.g. "`man readv`"). The I/O vector described by this structure has ***numOfLdcs*** + 1 entries (where ***numOfLdcs*** is the number of LDCs contributing to the event) with entry number 0 being the header of the super event.

ebRegister

C Synopsis

```
#include "libDateEb.h"

int ebRegister( void )
```

Description Registers the process with the ***eventBuilder*** and attaches to its memory banks.

Returns ***TRUE*** in case of success, ***FALSE*** otherwise.

ebGetNextEvent

C Synopsis

```
#include "libDateEb.h"

struct iovec *ebGetNextEvent( void )
```

Description Gets the next available event from the output queue of the ***eventBuilder***.

Returns ***NULL*** if the queue is empty, **-**1 on error, pointer to I/O vector otherwise.

ebGetNextEventDescriptor

C Synopsis

```
#include "libDateEb.h"

int ebGetNextEventDescriptor(
   void **descriptor,
   int *descriptorSize )
```

Description Gets the descriptor to the next available event from the output queue of the ***eventBuilder***. On success, the `descriptor` parameter is loaded with the address of the result and the `descriptorSize` parameter contains the size in bytes of the descriptor. The format of the descriptor is not published and may vary between releases of DATE. The event descriptor can be manipulated only using the `ebDescriptorToIovec` routine and it must be released using the `ebReleaseDescriptor()` routine.

Returns 0 if the queue is empty, -1 on error, integer positive value otherwise.

ebReleaseEvent

C Synopsis

```
#include "libDateEb.h"

int ebReleaseEvent( struct iovec * )
```

Description Releases the event described by the given I/O vector. The parameter must be an I/O vector returned by a previous call to `ebGetNextEvent()`. The routine also disposes the input `iovec` that must not be used after this routine returns.

Returns ***TRUE*** in case of success, ***FALSE*** otherwise.

ebReleaseDescriptor

C Synopsis

```
#include "libDateEb.h"

int ebReleaseDescriptor( void *descriptor )
```

Description Releases the descriptor of the event pointed by the input parameter, which must have been returned by a previous call to `ebGetNextEventDescriptor()`. Please note that this call will not release the event itself (for this purpose use the `ebReleaseEvent()` routine).

Returns 0 in case of success, -1 otherwise.

ebDescriptorToIovec

C Synopsis

```
#include "libDateEb.h"
struct iovec *ebDescriptorToIovec( void *descriptor )
```

Description Converts the input parameter, which must have been returned by a previous call to `ebGetNextEventDescriptor()`, to a standard `iovec` structure. The descriptor and the `iovec` must be disposed using the appropriate routines (`ebReleaseDescriptor()` and `ebReleaseEvent()`).

Returns 0 in case of success, -1 otherwise.

ebEor

C Synopsis

```
#include "libDateEb.h"

int ebEor( void )
```

Description Checks for end of run.

Returns ***TRUE*** if the run is closed and no more data is available from the ***eventBuilder***.

ebGetLastError

C Synopsis

```
#include "libDateEb.h"

const char * const ebGetLastError( void )
```

Description Gets a string describing the last error condition encountered by the library.

Returns Pointer to a read-only string.

10.5 Recording with the Multiple Stream Recorder

10.5.1 Overview

The ***online*** recording described in the previous section liberates the ***eventBuilder*** from the overheads related to physical data recording and lets it free to execute its principal task more effectively. The benefits of this mode can be important when event pre-processing is required before recording, especially on multi-processor / multi-core platforms. Additional gains can be achieved by having several concurrent recording streams: when one stream is busy (e.g., is waiting for a file to open), other streams could carry on. This effect is marginal when one is recording to a fast local file system. However, when one is dealing with a remote mass storage system having a limited throughput per data stream, such as CASTOR [9], use of multiple stream recording becomes essential. The experience of the ALICE data challenges suggests that >2 recording streams per GDC are needed to match CASTOR's aggregate throughput with the one of DATE.

The DATE ***mStreamRecorder*** process (MSR) is the default online recording application for the ***eventBuilder***, designed with the above considerations in mind. It enhances the basic GDC recording features by offering:

- concurrent ***asynchronous*** and ***individually configurable*** recording streams.
- a possibility to record to CASTOR.
- real-time transcoding of raw events into a ROOT [10] tree format compatible with the ALICE offline analysis software, using ***AliRoot*** API [16].

MSR can be run either together with the ***eventBuilder*** configured for online recording, or as a stand-alone application to "replay" pre-recorded raw data files. It consists of the main steering and dispatching process, ***disp***, and a number of concurrent ***stream*** processes running on the same machine (see Figure 10.1).

disp is launched at the "***Start processes***" phase of the run control. Its task is:

- to read and interpret the configuration file.
- to configure and fork the stream processes.
- to read the event descriptors from the output FIFO of the ***eventBuilder*** and dispatch them to the streams, via individual FIFOs.
- to report the status information to DATE.

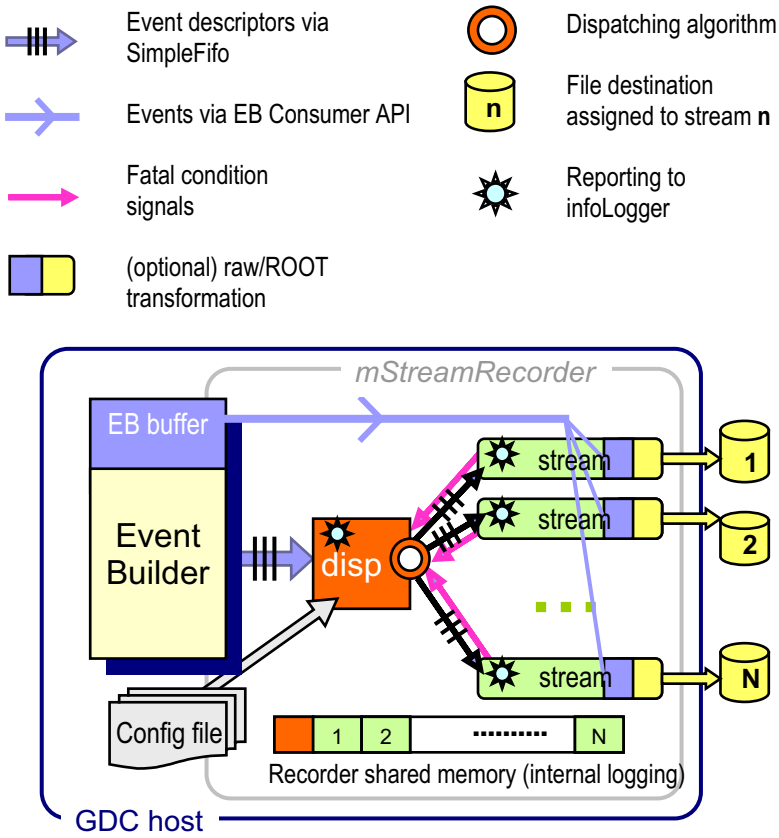

Figure 10.1 A schematic block-diagram of the mStreamRecorder. The legend is shown at the top

The available dispatching methods distribute the events uniformly between the streams. An option of having dedicated streams with custom filtering (e.g., according to trigger pattern) is reserved but not implemented yet.

Each ***stream*** is totally independent of other stream processes. Its tasks are:

- to receive event descriptors dispatched to it by ***disp*** via the individual FIFO.
- to construct the `iovec` pointing to the corresponding event parts (the header and sub-events) in the ***eventBuilder*** buffer.
- to manage output files on a specified destination.
- to write the events, optionally transformed into ROOT structures, to the output.
- to handle I/O errors and report its status to ***disp*** and DATE.

MSR uses the following DATE components: the ***eventBuilder*** client API (Section 10.4.2), the ***simpleFifo*** (Section 16.3) package and the DATE

infoLogger (Chapter 11). The source codes and the related executables are located in ***${DATE_MSTREAM_DIR}*** and ***${DATE_MSTREAM_BIN}*** directories, respectively.

The following sub-sections describe how to configure and run MSR.

10.5.2 MSR configuration file

10.5.2.1 Configuration file: naming and handling

The configuration of `mStreamRecorder` is done on a per-partition basis. This allows different partitions to use dedicated set of parameters such as file size or destination path. Two approaches are possible:

1. create a ***specific*** configuration for each partition;
2. create a ***generic*** template to instantiate for each partition.

A specific configuration file has priority over a generic template.

Specific configuration files use the file name `mStreamRecorder.`**`PART_NAME`**`.config` where **`PART_NAME`** is:

- the name of the partition for standalone runs (e.g. started via **`DCA`**);
- the name of the partition preceeded by `ALL` (e.g. `ALL`**`PHYSICS_1`** for the partition **`PHYSICS_1`**) during global runs (e.g. started via **`PCA`**).

Specific configuration files are used “as is” by `mStreamRecorder`.

A generic template must be saved under the name **`mStreamRecorder.config.template`** and it contains the configuration defined below with the inclusion of the following run-time fields:

- `__WRITE_VOLUME__`: this field can be used to select a partition-dependent write volume and it is replaced at run-time by the name of the partition;
- `__FILE_NAME_ATTR__`: this field can be used to create a file name which is partition-dependent and usable by the **`TDSM`** for run-time selection purposes. It is replaced at run-time by **`_nameOfDCA`** for standalone runs (**`nameOfDCA`** contains the name of the DCA, usually the name of the detector) or by the keyword ‘`_Technical`’ during **`technical`** runs (to allow writing the data coming from technical runs into a separate directory, easier to handle from the PDS side). During **`physics`** runs, the field is simply removed from the file name.

The template approach shall be used for sites running multiple partitions. Test sites should rather use specific creation files which are easier to read and to maintain.

10.5.2.2 Configuration examples

A DATE user may wish to record to a variety of output destinations (local file system, CASTOR, remote file server) and use different data formats (raw ***eventBuilder*** format, ROOT tree) and / or protocols (local, RFIO, ROOTd). This diversity can be described in a concise and flexible way by the MSR configuration

file whose syntax is based on the "`name key=value key=value...`" paradigm.

In simple cases of a uniform configuration for all GDCs, this file may consist of just a few lines, as is illustrated by examples in Listing 10.1 and Listing 10.2. They differ mainly in the definition of the default output stream (***`default_str`***): CASTOR files in Listing 10.2 need more attributes to be defined. Other minor differences illustrate the grammatical features which will be explained later.

Listing 10.1 A simple configuration with 3 streams per GDC, recording to a local disk

```
 1:  >COMMON Nstreams=3
 2:  >RECORDERS
 3:    default_rec
 4:  >OSTREAMS
 5:    default_str path=/scratch fsize=1024
```

Listing 10.2 A simple configuration with 3 streams per GDC, recording to CASTOR

```
 1:  >COMMON
 2:  >RECORDERS
 3:    default_rec Nstreams=3 stream=default_str
 4:  >OSTREAMS
 5:    default_str fsize=1024
 6:    pool=alice_stage stager=stagealice
 7:    path=/castor/cern.ch/alice/daq_dev/daq_recorder
```

Listing 10.3 shows the configuration for ROOT recording to CASTOR via ROOTd protocol, the same for all GDCs.

Listing 10.3 The configuration for ROOT recording to CASTOR

```
 1:  # ROOT recording to CASTOR
 2:  >COMMON timing=1 loglevel=1 Nstreams=3 root=3
 3:  >RECORDERS
 4:    default_rec method=2
 5:    stream=default_str fsize=255 !! NB: fsize has no effect here!
 6:  >OSTREAMS
 7:    default_str sleep=1 path=/castor/cern.ch/user/d/developer
 8:    filename=%h_%R_%s_%f_%T.root ! ROOT-style filename
 9:    mxrecl=0 fsize=1024
10:    pool=default stager=lxs5007 !!!! special CASTOR stager!
```

Finally, a more complicated example in Listing 10.4 shows how to arrange separate configurations for different GDCs and define a variety of streams with different output destinations. It also illustrates most of the syntactic and semantic rules of the MSR configuration language.

Listing 10.4 The configuration with special properties for the GDC pcaldXXgdc

```
 1:  !\
 2:  A special configuration for pcaldXXgdc
 3:  Author: DATE developer April 2005
 4:  \!
 5:  # common definitions
 6:  >COMMON method=1 loglevel=1 sleep=1
 7:
 8:  # recorder definitions
 9:  >RECORDERS
10:    default_rec method=2 stream=default_str   ! single stream
11:    pcaldXXgdc fsize=255 Nstreams=4           ! four streams
12:      filename=%h_%r_%s_%f.data stream=default_str stream=public \
13:      filename=Test_%r_%f.data stream=test
14:
15:  # output stream definitions
16:  >OSTREAMS
17:    default_str sleep=2 path=/local \
18:      mxrecl=0 ! line-continuation sign "\" is optional
19:      pool=public stager=stagepublic fsize=2047 ! fsize is in MB
20:    test path=/castor/cern.ch/user/d/developer/test_dir \
21:      =public ! a copy of all attributes of stream public
22:    public mxrecl=20000 path=/castor/cern.ch/user/d/developer
```

10.5.2.3 File names

MSR produces per host, per run and per stream output files. The meta-characters "`@`" and "`#`" can be used in their names, as described in Section 13.2. In addition, MSR interprets the percent sign "`%`" as a meta-character in all filenames appearing in the configuration file. The letters preceeded by "`%`" are replaced with the corresponding values at the run time, as follows:

- `%r` – (same effect as "`#`") the run number, without leading zeroes.
- `%h` – (same effect as "`@`") the short host name, like the one produced by the Linux command "`hostname --short`".
- `%H` – the full host name, e.g. `pcaldXX.cern.ch`, as produced by the Linux command "`hostname --long`".
- `%G` – the full GDC name, as specified by the hostname attribute in the role database.
- `%g` – same as "`%G`", with the "`gdc`" suffix stripped off, if present.
- `%R` – the 8-digit run number, with leading zero-padding.
- `%s` – the stream number (the numbering starts with `0`).
- `%f` – the sequential file number (the numbering starts with `1`).
- `%F` – the 3-digit sequential file number, with leading zero-padding.
- `%T` – the current time stamp, in the form `YYYYMMDD_hhmmss`.
- `%S` – the stream sequential number.

The default output file name is "`%h_%R_%s_%f.data`". Unlike with the direct GDC recording, the files with sequential number `0` are not created. The start-of-run events are transferred to the first stream(s) that become available to ***disp***.

A user can change the default file name template by using the ***filename*** attribute in the MSR configuration file. The name "`%h_%R_%s_%f_%T.root`" is strongly

recommended for ROOT data files. The directory part of the fully specified file name must be defined by the obligatory ***path*** attribute. CASTOR files are distinguished by the "`/castor/cern.ch`" prefix in the path name. Note, that ROOT recording is ***not*** automatically enabled even if the "`.root`" suffix is specified. For that purpose the attribute ***root*** should be used.

10.5.2.4 The configuration file syntax: tags and attributes

This section presents a formal description of the MSR configuration file syntax and semantics.

A MSR configuration file is a free-format plain text file in which the word items (contiguous strings of non-blank text characters, up to 512 characters long) are interpreted as either ***tags***, or ***attributes***. The items are separated by blanks, or tabs, or new lines, or an arbitrary mixture of them.

The tags starting with a bracket ">" are called ***structural***: they identify different levels in the configuration tree. All other tags are just ***names*** of objects belonging to those levels.

The attributes, distinguished by the equal sign "=" in the item, have to follow the tag which they qualify. The part to the left of the equal sign is the attribute ***key***, the part to right is the attribute ***value***, internally stored as a text string. The combination of a tag and its attributes will be referred to as a ***tag definition***. Tags without attributes are syntactically correct.

The attributes with the empty key, such as "`= something`" have a special meaning: they are replaced with a copy of all attributes belonging to another tag given as the value. For example, in Listing 10.4 the stream "`test`" has an attribute "`= public`", so all attributes of the stream "`public`" will be appended to "`test`". As the result, the stream "`test`" will receive the new attribute ***mxrecl***, as well as the second ***path*** attribute (which will be ignored at the semantic level because of rules of precedence, described later).

The layout of configuration files is not fixed by the syntax. For example, the entire configuration file may consist of a single line, e.g.

```
>COMMON Nstreams=3 >RECORDERS default_rec >OSTREAMS default_s
tr path=/scratch
```

However, for sake or readability, it is recommended to place tags at the beginning of separate lines, as in the examples quoted earlier. Indents, tabs, new lines and comments can be used freely to ease the reading.

The comments are introduced by a hash "#" and an exclamation mark "!" characters. Their use is illustrated by Listing 10.4. The lines with "#" as the first non-blank character are purely commentary. An exclamation mark begins an ***inline comment*** spanning the rest of the line, while the combination "!\" begins a long comment which extends to the end of line containing the terminating symbol "\!".

Backslash "\" can be optionally used as a line-continuation sign, though the syntax does not require it. The rest of the line after a backslash is ignored and can be used for inline comments, like with the "!".

All meaningful items in the configuration files are case-sensitive.

10.5.2.5 The configuration file structure

All tags, together with their attributes, must be grouped into three sections similar to the "roles" in the DATE roles database:

- **`common section`**: the structural tag `>COMMON` itself and its attributes.
- **`recorder section`**: the structural tag `>RECORDERS` and the name tags listed after it. By a ***recorder*** we mean an instance of the MSR running on a given GDC. The name tag defining a specific recorder must be identical to the GDC name, defined in the DATE roles database (see, for example, Listing 4.3).
- **`output stream section`**: the structural tag `>OSTREAMS` and the name tags listed after it. All varieties of stream configurations needed for all GDCs are described here. These configurations are instantiated by ***stream*** attributes appearing in recorder tags.

Within each section, the tags may appear in any order. The sections `>COMMON`, `>RECORDERS` and `>OSTREAMS` may also appear in any order within the configuration file, but the important requirement is that they must all be present. For the common section this simply means that the tag `>COMMON` must be present (with or without attributes). As to the recorder and stream sections, each of them must contain the corresponding structural tag (without any attributes) and at least one default tag definition:

- **`default_rec`** tag, describing the ***default recorder***, must appear in the recorders section. All rules for recorder definitions apply to it. Its main purpose is to provide the configuration for the GDCs which are not explicitly defined in the recorders section. In particular, if all GDCs are equal, ***default_rec*** can be the only recorder defined, as shown in Listing 10.1, Listing 10.2 and Listing 10.3.
- **`default_str`** tag, describing the ***default stream***, must appear in the streams section. The attribute "`stream=default_str`" is automatically added to any recorder tag having no ***stream*** attributes specified explicitly or implicitly (via copying). This feature is illustrated by Listing 10.1. If all streams in the system have the same properties, ***default_str*** can be the only stream defined.

Thus, there are five tags which must appear in any MSR configuration file.

10.5.2.6 Scopes of attributes and rules of precedence

All attributes used in MSR are listed in Table 10.1 and described in more detail in Section 10.5.3. Each of them (except the syntax-level ***copy*** attribute) defines a certain property or parameter of the object it qualifies. The attributes appearing in `>COMMON` apply to all recording streams and all recorders. The attributes appearing in a specific (***recorder*** or ***stream***) tag definition apply only to that definition and the ones derived from it. For example, the stream attribute ***fsize*** in the tag `pcaldXXgdc` in Listing 10.4 applies only to streams created for this recorder, including the instance of the default stream; it is not propagated to instances of the default stream for other recorders. The exceptions are the default recorder and stream, defined by ***default_rec*** and ***default_str*** tags: their attributes provide default values for other recorders and streams.

Table 10.1 Attributes in MSR configuration files

Attribute key	Type	Default value[a]	Property of[b]	Description
path	char*	none (obligatory)	**S**(R,C)	path of output file
stager	char*	"stagepublic"	**S**(R,C)	CASTOR stager host name
pool	char*	"public"	**S**(R,C)	CASTOR pool name
fsize	int	256	S(R,**C**)	max file size (Mbytes)
nevents	int	0 (no limit)	S(R,**C**)	maxnumber of events per file
timing	int	0 (no timing)	S(R,**C**)	timing logging
timer_log	char*	"Stream_time.%R"	S(R,**C**)	timing log file
sleep	int	1 (minimal)	S(R,**C**)	***stream*** polling latency
mxrecl	int	0 (no buffering)	S(R,**C**)	record length for buffered writing (non-ROOT only).
filename	char*	"%h_%R_%s_%f.data"	**S**(R,C)	output file name template
root	int	-1 (no transcoding)	S(R,**C**)	ROOT recording mode
compress	int	0 (no compression)	S(R,**C**)	compression level
filtermode	int	0 (no filtering)	S(R,**C**)	3rd level filtering
maxtagsize	double	2.e8	S(R,**C**)	max size of tag DB (bytes)
runDBFS	char*	"/tmp/meta%s"	S(R,**C**)	run DB path name
tagDBFS	char*	"/tmp/tags%s"	S(R,**C**)	tags DB path name
alienHost	char*	***NULL*** (no ***AliEN*** DB)	S(R,**C**)	***AliEN*** host [17] reserved for future use
alienDir	char*	***NULL*** (no ***AliEN*** DB)	S(R,**C**)	***AliEN*** directory [17] reserved for future use
method	int	1 (equal load)	R(**C**)	dispatching method
Nstreams	int	none (optional)	**R**(C)	forced number of streams
stream	char	***default_str***	**R**	creates an instance of the named stream
(empty)	char	none (optional)	R,S only	copies attributes from another name
use	int	none (optional)	C	forced recorder name
loglevel	int	1 (minimal)	C	log level
dump	int	0 (no dump)	C	to debug the config file
run	int	none (optional)	C	forced run number in stand-alone mode
source	char	none (optional)	C	full name of the data source file in stand-alone mode
Nev	float	none (optional)	C	event limit in stand-alone mode

a. The built-in default value. The attribute without defaults are either ***obligatory*** (must appear in the configuration file) or ***optional*** (no effect, if absent).

b. The tag which an attribute qualifies (***C***= "`>COMMON`", ***R***=recorder, ***S***=stream). Alternative attribute placements are indicated in parentheses. The recommended attribute placement is shown in bold. Any attribute that can be placed in `>COMMON` may also appear among command-line arguments.

Most of the attributes qualifying streams may also appear in recorder and common definitions. Similarly, the recorder attributes (except ***stream***) may also appear in the common section. An alternative placement of an attribute changes its scope and significance. The attribute value assignment rules are given in Table 10.2. When a "lower-level" attribute appears in a "higher-level" definition, its value overrides all lower-level definitions. This feature makes the configuration file grammar more flexible, permitting to affect an entire group of objects without touching the original low-level definitions.

Table 10.2 Rules of precedence for MSR attribute values

priority	The effective value of the attribute *A* for the stream *S* on recorder *R* is retrieved from:	The effective value of the attribute *A* for the recorder *R* is retrieved from:
1 = ***highest priority***	command-line or `>COMMON` section (e.g., ***sleep*** attribute in Listing 10.4)	command-line or `>COMMON` section
2	`R` tag in the `>RECORDERS` section: the last occurrence of `A` attribute preceding the corresponding `stream=S` (e.g., ***filename*** for the streams of recorder `pcaldXXgdc` in Listing 10.4)	`R` tag in the `>RECORDERS` section: the first occurrence of `A` attribute in the tag description
3	The first occurrence of `A` in the tag `S` in `>OSTREAMS` section (e.g., ***mxrecl*** and ***path*** for the stream ***public*** in Listing 10.4)	The first occurrence of `A` in ***default_rec***
4	The first occurrence of `A` in ***default_str*** (e.g., ***pool*** and ***stager*** for all ***streams*** in Listing 10.4)	The built-in default value from Table 10.1
5 = ***lowest priority***	The built-in default value listed in Table 10.1 (e.g., the ***timing*** attribute for all ***streams*** in Listing 10.4)	

The built-in defaults (Table 10.1) have the lowest priority and are applied only if the corresponding attributes are not specified, directly or indirectly, for a given stream. The ***copy*** attributes are exercised at the syntax parsing stage, so the copied properties are regarded as if they were explicitly specified.

The order of attributes within the tag definition matters only for stream-related attributes in recorder definitions. In that case the last value preceding the `stream=X` is applied to the instance of `X`. For example, `fsize=255` in Listing 10.3 has no effect, as there are no ***stream*** attributes after it. In all other cases, when the same attribute appears several times within a tag, the first value is taken.

10.5.2.7 Summary

In summary, the MSR configuration file consists of tags and tag attributes, grouped into three sections. The tags in the >RECORDERS section describe individual GDCs. The tags in the >OSTREAMS section describe abstract output streams, instantiated by ***stream*** attributes of recorder tags. The obligatory tags ***default_str*** and ***default_rec*** describe the default stream and recorder, respectively. The >COMMON section contains the attributes whose values are enforced globally. Almost all attributes have built-in defaults which can be modified at different scope levels. Multiple definitions are resolved using rules of precedence. The format of the configuration file is free and may contain inline comments.

10.5.3 Description of the MSR configuration attributes

This sub-section describes how to specify the values of the MSR configuration attributes, summarized in Table 10.1. Initially stored as text strings, these values are interpreted according to their ***type*** at the semantic parsing stage. Most of them have meaningful built-in default values. All attributes, except ***stream***, can be specified in the command line. In that case, they are ***prepended*** to the >COMMON section and, therefore, have the highest precedence.

- **path**
  The full pathname (without terminating "/") of the directory which will contain the output file. For CASTOR files it has to start with "/castor/cern.ch". This attribute is obligatory and must be specified for any stream created by MSR. The specified pathname must have write permissions for the owner of the MSR executables ***disp*** and ***stream***, stored in the directory ***${DATE_MSTREAM_BIN}*** (they have SUID and SGID bits set).
- **stager**
  The CASTOR [9] stager host name. MSR assigns the value of this attribute to the CASTOR environment variable ***STAGE_HOST***. By default, the CERN public stager is used.
- **pool**
  The CASTOR disk pool name. MSR assigns the value of this attribute to the CASTOR environment variables ***STAGE_POOL*** and ***STAGE_SVCCLASS***. By default, the CERN public pool is used.
- **fsize**
  The file size limit, in MB. The output file is closed when the actually written size exceeds this limit (less a safety margin, for ROOT files).
- **timing** and **timer_log**
  The detailed stream timing can be enabled for each output stream, by specifying "timing=1". The log will be written via ***infoLogger*** to the log stream specified by the ***timer_log*** attribute (by default, "Stream_time.%R", common for all streams). The statistics (minimal, maximal and mean values, the accumulated sums) are recorded at each file close for the following time intervals: time spent while waiting for events from disp, write operations, time between consecutive writes, file open and close latencies.
- **sleep**
  The stream processes poll the dispatcher FIFO while waiting for the next event. Whenever the FIFO is found to be empty, the stream executes usleep(s), where "s" is the value of the ***sleep*** attribute. "sleep=0" turns sleeping off (not

recommended!) and any negative value enables the minimal possible non-CPU consuming wait interval (10 ms or 20 ms, depending on the Linux platform).

- **mxrecl**
  A non-zero value enables buffering for non-ROOT recording. The optimal value should be found experimentally, as it strongly depends on running conditions and the average event size. By default, buffering is disabled.

- **filename**
  The output file name, see Section 10.5.2.3 for details. The filenames should contain symbols identifying streams and sequential file numbers, to avoid clashes. Such clashes are not detected by the MSR and might not even cause run-time errors, but the data will be tacitly overwritten.

- **root**
  A non-negative value enables ROOT recording and specifies the access protocol for the raw DB created by the corresponding stream. The currently supported values are: "`0`" (writing to a local filesystem, using class ***AliRawDB***) and "`3`" (writing to CASTOR via rootd daemon, using class ***AliRawCastorDB***). The values "`1`" and "`2`" are reserved for RFIO / CASTOR (using class ***AliRawRFIODB***) and plain ROOTd (using class ***AliRawRootdDB***). For further details about ROOT-related classes, refer to their description in the ***AliROOT*** documentation [16].

  The ROOT recording must also be enabled at the compilation level, by defining the ***ROOTsys*** macro in the MSR ***GNUmakefile*** (Section 10.5.4). A non-ROOT version of MSR will abort if a non-negative value of ***root*** is specified.

  Using different ROOT recording modes for different streams, though possible with MSR, is discouraged.

- **compress**, **filtermode**, **maxtagsize**, **runDBFS**, **tagDBFS**, **alienHost** and **alienDir**
  For the streams with the ROOT recording enabled, these attributes qualify the ROOT recording mode and are transferred to the class ***AliMDC*** constructor (the corresponding API function is reproduced in Listing 10.5). The special values are:

  - leading "`-`" in ***runDBFS*** and / or ***tagDBFS***: the corresponding DB creation is suppressed and its pathname is reset to ***NULL***.
  - "`maxtagsize=0`": the tag DB creation is suppressed, ***tagDBFS*** is reset to ***NULL***.

Listing 10.5 An API function used by the MSR to create an AliMDC object

```
 1: include "AliMDC.h"
 2: // creating AliMDC object for ROOT recording with MSR
 3: void *alimdcCreate(
 4:   int         compress,
 5:   int         filtermode,
 6:   const char* runDBFS,     //
 7:   Bool_t      rdbmsRunDB,  // =0 (MySQL run DB is disabled)
 8:   const char* alienHost,
 9:   const char* alienDir,
10:   double      maxtagsize,
11:   const char* tagDBFS) {
12:     return new AliMDC( compress,
                           kFalse,
                           AliMDC::EfilterMode(filtermode),
                           runDBFS,
                           rundbmsRunDB,
                           alienHost,
                           alienDir,
                           maxtagsize,
                           tagDBFS );
13: }
```

- **method**
  This attribute defines the dispatching method used by the corresponding recorder to distribute the events between the streams. The value "1" corresponds to the "equal-load" method and is assumed by default. The value "2" corresponds to the "first-available" method which bypasses the busy streams (having their buffer FIFOs almost full). Both methods are protected by an internal time-out which may temporarily disable the overloaded stream(s). The difference in performance for the two methods is marginal and strongly depends on the running conditions.

- **Nstreams**
  This attribute enforces the specified number of streams for the corresponding recorder. By default, MSR creates as many streams, as there are ***stream*** attributes in the recorder tag. If no ***stream*** attributes are present in that tag, one default stream (defined by the obligatory ***default_str*** tag) is assumed. If the value of ***Nstreams*** is less than the actual number of ***stream*** attributes listed in the tag, then the trailing streams are discarded. Otherwise, the entire list is iterated until the number of streams requested by ***Nstreams*** is reached.

- **stream**
  This attribute creates an instance of the named stream for a given recorder. The order in which ***stream*** attributes are listed within the same recorder tag may be relevant for two reasons. First, if the effective value of the ***Nstreams*** for that recorder is less than the number of its ***stream*** attributes, only the leading ones are retained. Second, if it is desirable to modify the properties of streams instantiated for a given recorder, all modifying attributes must ***proceed*** the corresponding ***stream*** attribute (see, for example, the use of the filename templates for the recorder `pcaldXXgdc` in Listing 10.4).

- **use**
  This attribute can be placed in the `>COMMON` section to enforce the specified recorder tag on ***all*** recorders. It overrides the default behavior of MSR which determines the GDC hostname and picks the recorder tag with that name (or the ***default_rec*** tag, if such tag is missing).

- **loglevel**

Defines the volume of status and diagnostic messages. When running in the DATE mode, all messages from all ***disp*** and ***stream*** processes are sent to the ***infoLogger***, with the recorder and stream names prepended. Note, that the detailed and debug levels of ***infoLogger*** are enabled only in a special debugging version of MSR produced with the "`_d_`" macro defined in the MSR ***GNUmakefile*** (see Section 10.5.4).

Each recorder sends a single-line starting message to the ***runLog*** stream. All subsequent messages go to the common dedicated log stream ("`LOGNAME = mStreamRecorder`"). Fatal errors are reported with the ***FATAL*** macro.

The timing logging (see the description of ***timing*** and ***timer_log*** attributes) is not affected by ***loglevel***.

- **dump**
  This attribute is useful when composing or testing configuration files, especially in the stand-alone mode. Its only effect is to produce a detailed dump of the configuration tree structure immediately after the syntax parsing and write it to ***stdout***.
- **run**
  This attribute can be placed in the `>COMMON` section to override the run number coming with the data. It is effective only in the stand-alone mode. If "`run=0`" is specified, MSR will stop after parsing the configuration file, without creating any streams.
- **source**
  The full name of the ***local*** source raw data file for the stand-alone ("replay") mode. This file can be created either directly by a GDC, or by MSR running with DATE in the non-ROOT mode. If the ***source*** attribute is omitted in the stand-alone mode, MSR will generate some meaningless dummy events which are good only to test MSR in the non-ROOT mode. For consistent testing, a short sample raw data file ***${DATE_MSTREAM_DIR}/sample_source.dat*** can be used.
- ***Nev***
  Specifies the number of events to process in the stand-alone mode. By default, MSR stops after processing all events in the source file (see ***source*** attribute). If ***Nev*** is given, the source file is "replayed" (in a loop, if needed) until the required number of events is reached.

10.5.4 How to build and run MSR

In order to build MSR components, run `gmake` in the ***${DATE_MSTREAM_DIR}*** directory containing the MSR source codes. The makefile ***GNUmakefile*** puts MSR libraries and executables into the directory ***${DATE_MSTREAM_BIN}***. Check that the executable files `disp`, `stream` and `cleanup` have the "`s`" bits set and the owner of these files has a write-access to the directories that MSR will be writing to. The makefile has internal variables-switches to control the makefile execution and / or to create functionally different versions of MSR:

- `EB=1` produces the "DATE" version reading from the ***eventBuilder*** and reporting to ***infoLogger***; `EB=0` produces the "stand-alone" version reading from a pre-recorded file, reporting to standard output and independent of any

DATE components, except ***simpleFifo***.

- ROOT=1 produces the ROOT-aware version using ***AliROOT*** API. The pathnames ***${ROOTSYS}*** (the ROOT installation directory) and ***${ALIROOT}*** (the ***AliROOT*** directory), required for the ROOT version, are defined within the makefile. ROOT=0 produces the version stripped of all ROOT-related features and independent of any ROOT resources.
- DEBUG=1 produces the debugging version of MSR. Currently, its only difference from the standard version (produced with DEBUG=0) is that the code to produce the detailed and debug level of logging messages (see the ***loglevel*** attribute) is included only in the debugging version of MSR. In future, the standard version can be further optimized by delegating some error checks to the debugging version.
- HC=1 for the DATE version replaces the default reporting to ***infoLogger*** with printing to the standard output, like in the stand-alone version (this applies also to timing logs, see the ***timer_log*** attribute).
- SHARED=1 instructs the makefile to create shared MSR libraries and link MSR correspondingly. With SHARED=0, static linking is performed. Note that all ROOT libraries are always dynamically-linked, independently from that switch.
- TRACE switch simply controls the makefile verbosity. TRACE=-1 retains the default gmake output. TRACE=0 is equivalent to "-s" option of gmake (any output except for errors is suppressed). TRACE=1 prints a one-line header for all actions the makefile performs (including the intrinsic compilation rules) and, finally, TRACE=2 adds the action headers to the default gmake output.

These variables are assigned in the beginning of ***GNUmakefile*** as follows: EB=1, ROOT=1, DEBUG=0, HC=0, SHARED=1 and TRACE=1. To modify the default value(s), either edit the ***GNUmakefile*** and re-make MSR, or specify the alternative assignment(s) as gmake arguments, for example:

```
gmake -W GNUmakefile DEBUG=1 SHARED=0
```

Note, that ***GNUmakefile*** itself is in the list of common dependencies for MSR, so editing ***GNUmakefile*** (or giving its name in the "-W" option) will force gmake to re-compile the entire MSR.

Before running MSR, one has to prepare the configuration file. By default, MSR uses the name ***${DATE_SITE_CONFIG}/mStreamRecorder.config*** (or ***./mStreamRecorder.config***, if the ***DATE_SITE_CONFIG*** environment variable is undefined). Using command-line options, an alternative filename can be specified as shown below. At Point 2, a special syntax is used to handle global templates and partition-specific configuration files (see Section 10.5.2).

The standard way to run MSR together with DATE is by using the DATE ***runControl*** Human Interface (Section 14.5). Before starting processes:

- set ***recordingDevice*** to "***:***" in the "***GDC configuration***" menu.
- check "***Recording enable***" and "***Recording to PDS***" options in the main menu.

To run the stand-alone version of MSR, enter a fully specified name of the ***disp*** executable, with optional arguments. Apart from any number of the configuration attributes, two options, "-v" and "-f", can be specified in the command line. The

"`-v`" option will cause ***disp*** to get loaded and print the MSR version number and properties. The "`-f`" option followed by a full filename specifies the configuration file to be used. A few examples are given in Listing 10.6. Note, that the configuration attributes specified in the command line have the highest priority and override the corresponding values in the configuration file (see Table 10.2).

Listing 10.6 Examples of starting MSR in the stand-alone mode

```
 1:  ./Linux/disp -f my.config Nev=1.e6 run=287 source=/scratch/%h.dat
 2:  ${DATE_MSTREAM_BIN}/disp
     source=${DATE_MSTREAM_DIR}/sample_source.dat
 3:  ./Linux disp -v
 4:     disp build IM 050505; debug version; EB;
 5:     ROOT (linked with
     ROOTSYS=/adcRoot/ROOT/Linux/CurrentRelease/root)
 6:  ./Linux/stream -v
 7:     stream build IM 050505; EB;
 8:     ROOT (linked with
     ROOTSYS=/adcRoot/ROOT/Linux/CurrentRelease/root)
 9:  ./Linux/disp -f test.config use=thatGDC dump=1 loglevel=2 run=0
10:  ./Linux/test_config test.config use=thatGDC
```

Line 1 shows an example of a "replay" run with a custom configuration file. It shows that the data source filenames may contain meta-characters. Line 2 shows a replay of the standard MSR sample file. Line 3 tests the version of the ***disp*** program. Line 6 shows that the ***stream*** program version can also be tested. This is also a way to test-load the ***stream*** process. Line 9 shows how to test a configuration file. The file will be fully parsed by ***disp*** and the streams for the recorder running on `thatGDC` defined in this file will be prepared, with all their parameters printed out. However, the actual streams will not be created and the execution will stop at that stage, because of the "`run=0`" attribute. The MSR tool application `test_config` (see Line 10) performs the same actions. Its advantage is that it does not depend on any external resources, like DATE or ROOT libraries.

The infoLogger system

11

A data-acquisition setup can consist of many nodes, each of them possibly running several DATE processes. For development and operation, it is needed to know how the distributed components behave, and what happens on the different machines. The DATE ***infoLogger*** package provides facilities to generate, transport, collect, store and consult log messages. This chapter describes how the ***infoLogger*** works and how to use it.

11.1 Introduction

The ***infoLogger*** system provides facilities to send, collect and browse log messages created by DATE components and user processes on different machines.

Figure 11.1 shows the overall architecture of the ***infoLogger*** system. A process calling a function of the ***infoLogger*** library sends the message to the local ***infoLoggerReader*** daemon. This process collects all the messages of the node where it runs, and sends them to a central ***infoLoggerServer*** daemon, which stores the received messages in a MySQL database. If at some point the transmission chain is broken, the messages are written to disk to avoid losing them. The ***infoBrowser*** user interface allows to read messages, either stored in the database or received online by the central server.

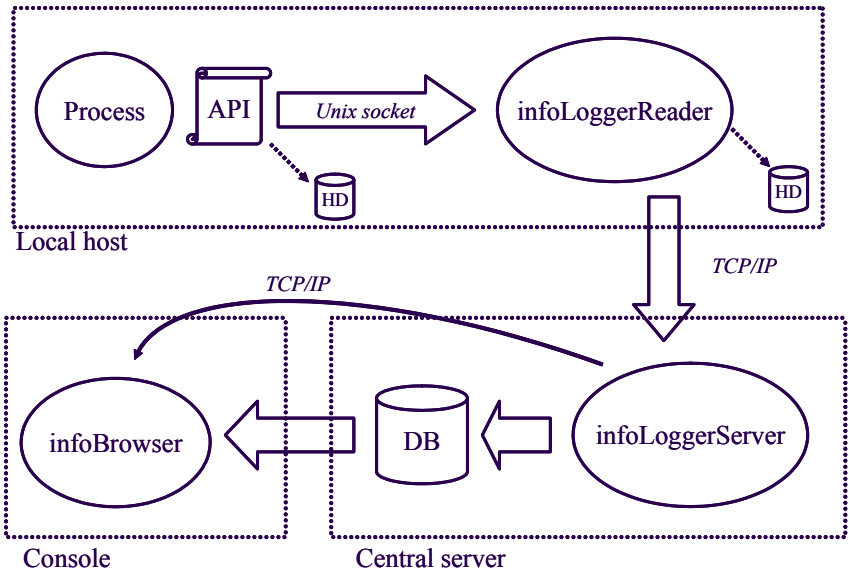

Figure 11.1 The DATE infoLogger architecture

11.2 infoLogger configuration

The connection parameters between the different processes are stored in the MySQL configuration database and should be defined with ***editDb*** (see Section 4.5) in the environment variables section. Table 11.1 describes the variables required.

Table 11.1 infoLogger configuration parameters - environment variables

Variable name	Meaning
DATE_INFOLOGGER_LOGHOST	Name of the host running the ***infoLoggerServer*** process
DATE_INFOLOGGER_MYSQL_DB	Name of the database to be used to store log messages.

Table 11.1 infoLogger configuration parameters - environment variables

Variable name	Meaning
DATE_INFOLOGGER_MYSQL_HOST	Host running the database server. It is recommended to have it on the same machine as the ***infoLoggerServer***.
DATE_INFOLOGGER_MYSQL_USER	MySQL username to connect to the database.
DATE_INFOLOGGER_MYSQL_PWD	MySQL password to connect to the database (with above user).

These environment variables are stored in the database configuration class named ***infoLogger***. They are created by default when installing the DATE database, and loaded at runtime by the components (readers and server). They are not loaded by the DATE setup procedure in the environment; the values are queried only when starting the ***infoLogger*** processes.

Default socket port numbers for the communication between the processes are defined in ***$(DATE_ROOT)/commonDefs/shellParams.common***, with the global environment variables ***DATE_SOCKET_INFOLOG_RX*** (reader to server) and ***DATE_SOCKET_INFOLOG_TX*** (browser to server). They don't need to be redefined.

The Unix named socket used to communicate between a log client and the local reader is based on the value of ***$(DATE_SITE)*** and does not need to be defined.

DATE messages are stored in the specified database. MySQL infoLogger table structure should be initially created, once configuration parameters are defined, with the command `/date/infoLogger/newDateLogs.sh -c`.

In addition, some ***infoLogger*** related log files are created in ***${DATE_SITE_LOGS}*** when needed, as described in Section 11.3. This variable is set by default to ***${DATE_SITE}/logFiles***.

Please note that the verbosity of some runtime processes are defined by the ***runControl*** run parameter named ***loglevel***. The higher this integer, the higher the verbosity of the processes. When used, this variable is handled by the processes before any call to the ***infoLogger***. Details about this parameter are described in Chapter 14.

11.3 The infoLogger processes

The ***infoLogger*** components create some log files in the ***${DATE_SITE_LOGS}*** directory in addition to the DATE log repository. These files contain status information, daemon error messages, and DATE log messages that could not be transmitted. For performance reasons, these log files are opened only once by all the ***infoLogger*** components. Therefore, you should first stop the ***infoLogger*** daemons before removing these files. Because of the operating system implementation, no error occurs on the daemons side if you remove the files while

the daemons run. They will continue to write to the same files, which are not accessible any more by the user and which will be destroyed when closed.

11.3.1 infoLoggerReader

The ***infoLoggerReader*** daemon is started automatically by the first process using the DATE ***infoLogger*** library on a specific host. It listens to a Unix named socket which name is based on ***${DATE_SITE}*** (no configuration required), and receives all log messages created on the node by the ***infoLogger*** library. Then messages are sent to the central server, as defined by variable ***DATE_INFOLOGGER_LOGHOST***, on TCP port ***DATE_SOCKET_INFOLOG_RX*** using a special protocol.

A control script, `${DATE_INFOLOGGER_BIN}/infoLoggerReader.sh`, is provided to start, stop, or restart it. ***DATE_SITE*** must be defined when invoking the script.

The ***infoLoggerReader*** processes associated to a specific ***DATE_SITE*** can be started and stopped remotely with the ***dateSiteDaemons*** script. It should be used to restart ***infoLoggerReader*** after a change in the configuration (for example a change of the host name where the ***infoLoggerServer*** is running, etc.).

11.3.2 infoLoggerServer

The ***infoLoggerServer*** daemon runs on a central node, defined by the variable ***DATE_INFOLOGGER_LOGHOST***, and receives log messages sent by remote ***infoLoggerReader*** processes. It stores messages in a MySQL database. It also accepts connections on TCP port ***DATE_SOCKET_INFOLOG_TX*** where clients can connect to get log messages as soon as they are received by the server, without querying the repository.

The message order of insertion and delivery is not guaranteed, only the message timestamp is reliable to order messages coming from a given machine. The accuracy of clock synchronization is critical when correlating events from different nodes, and it is not under the control of the DATE system.

A control script, `${DATE_INFOLOGGER_BIN}/infoLoggerServer.sh`, is provided to start, stop, or restart it. ***DATE_SITE*** must be defined when invoking the script.

The ***infoLoggerServer*** process associated to a specific ***DATE_SITE*** can be started and stopped remotely with the ***dateSiteDaemons*** script. It should be used to launch ***infoLoggerServer*** before using DATE, or to restart ***infoLoggerServer*** after a change in the configuration (i.e. database parameters, node name, etc.).

11.3.3 infoBrowser

The ***infoBrowser*** process is a user interface to extract and display log messages from the ***infoLogger*** system. A full description of it is given in the operations section of the ALICE DAQ WIKI.

11.4 Log messages repository

Each DATE log message is made of several attributes:

- ***Severity***: the information level of each message. This can be one of:
  - ***Information***: used for messages concerning normal running conditions.
  - ***Error***: when an abnormal situation has been encountered but execution can somehow continue.
  - ***Fatal***: when an unrecoverable situation has been detected and normal execution cannot be guaranteed. It usually causes the end of the current run.
- ***Timestamp***: time of the message creation, with microsecond resolution, as provided by the local operating system where the message is created.
- ***Host name***: host where the message was created.
- ***Process ID***: identifier of the process creating the message.
- ***User name***: user running the process creating the message.
- ***System***: system originating the message. For all DATE processes, this is set to ***DAQ***. This field is useful if the infoLogger facilities are shared with other systems, like the ***ECS***. It is defined by the value of the environment variable ***DATE_INFOLOGGER_SYSTEM***.
- ***Facility***: the activity family, usually the DATE package name creating the message. This can be for example ***readout***, ***recorder***, ***runControl***, or ***operator*** in case of a message coming from the command line.
- ***Stream***: the log stream name the message belongs to. It is usually the name of a detector (standalone operation) or of a partition (global runs).
- ***Run number***: when available, the run number associated to the message. This field can be undefined (empty value or -1). Most processes related to the ***runControl*** set this attribute when logging messages.
- ***Message***: the log information. It is a text string. End of line characters are not allowed in a message. Multiple-line messages are split into different messages (with the same other attributes).

Log messages, with all the associated information as described above, are centrally collected for the whole DATE system, and stored in a repository. Two implementations of the repository are available: the ***infoLoggerServer*** can either write to a flat file or to a MySQL database.

Some tags have a maximum string length allowed in the MySQL version of the repository. These limits are defined in the file `${DATE_INFOLOGGER_DIR}/newDateLogs.sh`.

11.4.1 MySQL database

Messages are stored in a table named ***messages***, which columns correspond to the log fields previously described. The MySQL database connection parameters and ***infoLogger*** tables should be initially created as described in Section 11.2.

11.4.2 Archiving

Messages received by the ***infoLoggerServer*** are stored in the ***messages*** table in the MySQL database. It is important to archive older messages to optimize the resources used to insert, browse or process log information in the main table. The quantity of information stored in the main table is also limited by the maximum file size allowed by the file system.

A utility is provided to manage the amount of logs over time:

- `${DATE_INFOLOGGER_DIR}/newDateLogs.sh -d`

  Deletes all messages in the main table (but not the archives).

- `${DATE_INFOLOGGER_DIR}/newDateLogs.sh`

  This script, launched without option, creates an archive from the main table. A new table (or file) is created, whose name includes the current date and time. All messages from the main table are moved to this archive.

It is recommended to delete regularly or archive the messages from the main log table, for example with a regular ***cron*** job calling `newDateLogs.sh` script. Be sure that ***DATE_SITE*** is defined when this script is called.

Log information stored in archived tables is still available to the ***infoBrowser.***

11.4.3 Retrieving messages from repository

The ***infoBrowser*** interface is the best tool to browse and display information stored in the log repository. It includes filtering and searching capabilities, and allows to export data to a text file. The full description of the ***infoBrowser*** is given in the operations section of the ALICE DAQ WIKI.

A simple data extraction tool, `${DATE_INFOLOGGER_DIR}/getLog.sh`, is also provided to extract information from the main table to ***stdout***, independently from the repository type. This script can also output logging information in the format of the DATE system version 4, and select data from a specific stream. Call it with **-h** option to get details on usage. This output can then be piped to ***awk*** or ***grep***, when looking for specific messages.

Users familiar with SQL can also query directly the ***messages*** table.

11.5 Injection of messages

Any process running on any DATE host can use the ***infoLogger*** system to transfer debug, information and error logs. Messages can be injected into the logging system using the command line tools, or with the APIs provided (C and Tcl). The DATE setup procedure must have been executed before launching any process using the ***infoLogger*** library.

All inserted messages must be native strings. If the message tag contains carriage returns, the message is split into several log messages with the same remaining tags.

When necessary, it is possible to copy messages injected in the ***infoLogger*** system to ***stdout*** by setting the environment variable ***DATE_INFOLOGGER_STDOUT*** to `TRUE`. It can be useful for interactive tools.

Note that the ***stream*** is set automatically at run time by the ***runControl***, therefore it should not be set manually to other values.

11.5.1 Logging from the command line

A set of executables allows to inject messages from the command line or from scripts. These tools are located in the ***${DATE_INFOLOGGER_BIN}*** directory. For details on their usage, invoke them with the **-?** command line option. The DATE setup procedure must be executed before using these programs.

- `log`

  Log a given message. Severity and facility may be provided.

- `logTo`

  Log a given message to a given stream. Severity and facility may be provided.

- `logFromStdin`

  Read messages from standard input. Messages are strings delimited by end of line character. It is possible to pipe the output of a program into this utility to have it injected into the log system. Severity, facility and stream may be provided.

Unless otherwise specified, the facility tag is set to ***operator***, and the destination stream tag is set to ***defaultLog***.

11.5.2 Logging with the C API

This section describes the macros and the methods available for programs written in C. A set of primitives are defined for the programmer in the header file ***${DATE_INFOLOGGER_DIR}/infologger.h***.

A C program by default sends messages tagged with ***Facility*** set to the package name (for DATE packages) or with the image filename (for non-DATE packages). This behavior can be changed by setting the C preprocessor variable ***LOG_FACILITY*** to the desired value. This must be done before including the ***infoLogger.h*** file, e.g.:

Listing 11.1 Setting the Facility name in C programs

```
1:  #define LOG_FACILITY "myFacility"
2:  include "infoLogger.h"
```

Unless otherwise specified, the destination stream tag is set to ***runLog***.

Compilation of C programs require the inclusion of the file ***infoLogger.h***. The linking with the library `libInfo.a` is also needed. Once the DATE setup procedure has been executed, a command to build the program is

```
gcc myProgram.c -I${DATE_INFOLOGGER_DIR} \
${DATE_INFOLOGGER_BIN}/libInfo.a
```

By default, a connection to the ***infoLoggerReader*** process is opened (and the daemon launched if necessary) when the first log message is created. It remains open during the life of the process. To explicitly open/close this connection, the functions infoOpen() and infoClose() can be used.

Additional functions calls exist which allow printf-like arguments, avoiding thus the need of a local buffer to prepare the message when variables values have to be included.

infoOpen

C Synopsis

```
#include "infoLogger.h"
void infoOpen(void)
```

Description Opens the connection to ***infoLoggerReader***. Its usage is optional: this function is called automatically when logging the first message. If the socket is not found, the ***infoLoggerReader*** is started.

infoClose

C Synopsis

```
#include "infoLogger.h"
void infoClose(void)
```

Description Closes the connection to the ***infoLoggerReader***. Its usage is optional.

infoLog_f

C Synopsis

```
#include "infoLogger.h"
void infoLog_f(const char* const facility, const char
severity, const char* const message, ...)
```

Description The specified *message* is injected into the ***infoLogger*** system, in the default log *stream*, with the given *severity* and *facility*.

message is a string as accepted by `printf`. Additional parameters can be provided for string formatting. This avoids buffering a message that includes variable values.

infoLogTo_f

C Synopsis

```
#include "infoLogger.h"
void infoLogTo_f(const char* const stream, const char* const
facility, const char severity, const char* const message, ...)
```

Description The specified *message* is injected into the ***infoLogger*** system, in the given log *stream*, with the given *severity* and *facility*.

message is a string as accepted by `printf`. Additional parameters can be provided for string formatting. This avoids buffering a message that includes variable values.

LOG

C Synopsis

```
#include "infoLogger.h"
void LOG( char severity, char *message )
```

severity can be one of:
LOG_INFO
LOG_ERROR
LOG_FATAL

Description The specified *message* is injected into the ***infoLogger*** system, in the default ***runLog*** stream, with the given *severity*.

LOG_TO

C Synopsis

```
#include "infoLogger.h"
void LOG_TO( char *stream, char severity, char *message )
```

severity can be one of:
LOG_INFO
LOG_ERROR
LOG_FATAL

Description The specified *message* is injected into the ***infoLogger*** system, in the given log *stream*, with the given *severity*.

LOG_ALL

C Synopsis

```
#include "infoLogger.h"
void LOG_ALL( char *stream, char severity, char *message )
```

severity can be one of:
LOG_INFO

LOG_ERROR
LOG_FATAL

Description The specified *message* is injected into the ***infoLogger*** system, both in the default ***runLog*** stream and in the given log *stream*, with the given *severity*.

INFO

C Synopsis
```
#include "infoLogger.h"
void INFO( char *message )
```

Description The specified *message* is injected into the ***infoLogger*** system, in the default ***runLog*** stream, with **Info** severity.

ERROR

C Synopsis
```
#include "infoLogger.h"
void ERROR( char *message )
```

Description The specified *message* is injected into the ***infoLogger*** system, in the default ***runLog*** stream, with **Error** severity.

FATAL

C Synopsis
```
#include "infoLogger.h"
void FATAL( char *message )
```

Description The specified *message* is injected into the ***infoLogger*** system, in the default ***runLog*** stream, with **Fatal** severity.

INFO_TO

C Synopsis
```
#include "infoLogger.h"
void INFO_TO( char *stream, char *message )
```

Description The specified *message* is injected into the ***infoLogger*** system, in the given log *stream*, with **Info** severity.

ERROR_TO

C Synopsis
```
#include "infoLogger.h"
void ERROR_TO( char *stream, char *message )
```

Description The specified *message* is injected into the ***infoLogger*** system, in the given log *stream*, with **Error** severity.

FATAL_TO

C Synopsis
```
#include "infoLogger.h"
void FATAL_TO( char *stream, char *message )
```

Description The specified *message* is injected into the ***infoLogger*** system, in the given log *stream*, with **Fatal** severity.

INFO_ALL

C Synopsis
```
#include "infoLogger.h"
void INFO_ALL( char *stream, char *message )
```

Description The specified *message* is injected into the ***infoLogger*** system, both in the default ***runLog*** stream and in the given log *stream*, with **Info** severity.

ERROR_ALL

C Synopsis
```
#include "infoLogger.h"
void ERROR_ALL( char *stream, char *message )
```

Description The specified *message* is injected into the ***infoLogger*** system, both in the default ***runLog*** stream and in the given log *stream*, with **Error** severity.

FATAL_ALL

C Synopsis
```
#include "infoLogger.h"
void FATAL_ALL( char *stream, char *message )
```

Description The specified *message* is injected into the ***infoLogger*** system, both in the default ***runLog*** stream and in the given log *stream*, with **Fatal** severity.

LOG_NORMAL_TH
LOG_DETAILED_TH
LOG_DEBUG_TH

Description The symbols define the thresholds for message generation. Messages may be injected into the ***infoLogger*** system depending on the `logLevel` run parameter. Refer to Section 14.11 for more details on this convention. The corresponding values are defined in ***infoLogger.h***.

11.5.3 Logging with the Tcl API

A subset of the C API can be called directly from Tcl scripts. The list of accessible functions is given in ***${DATE_INFOLOGGER_DIR}/infologger.i***.

It includes, in particular, the ***infoLog*** and ***infoLogTo*** functions described in Section 11.5.2.

To use the library, load the module at the beginning of the Tcl script:

```
load $env(DATE_INFOLOGGER_BIN)/libInfo_tcl.so infoLogger
```

Then, call the ***infoLogger*** functions with the same arguments as defined in the C interface:

```
infoLog "my facility" "I" "This is an information message"
```

12 The eventBuilder

The DATE ***eventBuilder*** is the software package running on a Global Data Collector (GDC), receiving data from several Local Data Concentrators (LDC), assembling the data into single events and recording them to the output stream.

This chapter includes a description of the event-builder architecture and describes how sub-events are identified as belonging to the same event and how they are built as a single event.

This chapter describes also how the ***eventBuilder*** uses some of the other DATE packages such as the ***runControl*** and the ***infoLogger***.

12.1 Overview

The DATE ***eventBuilder*** is a software package responsible for merging together several streams of sub-events data originated from different readout subsystems into a single stream of events. This stream can be directed to the appropriate recording device or - using a memory-mapped scheme - to the next processing stage (filtering, compression, special recording).

A DATE data-acquisition system is composed of one or several parallel readout streams. Each of these stream(s) is carrying the data produced by the front-end electronics of one detector or part of it. This front-end electronics of each stream is controlled and readout by one processor called the Local Data Concentrator (LDC).

The event building is performed by processors called the Global Data Collector (GDC). The sub-events are transferred from the LDCs to the GDCs using the socket library of TCP/IP. The transfer is executed by the DATE **recorder** process running on the LDC when it is configured to use a GDC as output device.

The sub-events are received by the ***eventBuilder*** which - according to the directives given by the event-building database - creates the appropriate event and forwards it to the following processing stage (recording or online transfer). The output of the ***eventBuilder*** can be either recorded directly to one or more local devices or sent to a further processing stage using fifos and memory buffers. Details on the different recording schemes and their description can be found in Section 10.4

The ***eventBuilder*** is running under the control of the DATE ***runControl*** system from which it receives the commands to start and stop the run and the parameters needed for a run. The ***infoLogger*** functions are used for normal and exceptional messages and to report run statistics and descriptions.

12.2 The event-builder architecture

12.2.1 The data transfer from the LDC to the GDC

The LDC ***recorder*** program (see Section 10.3) writes data onto an output stream whose name is given by the ***runControl***. Amongst several possibilities, the output stream can be a GDC machine where the ***eventBuilder*** is running. By defining the output stream to be a GDC, the LDC becomes part of a multiple hosts DAQ system.

When the run is starting, the LDC ***recorder*** program opens the output stream on one TCP/IP socket of the GDC. The ***eventBuilder*** accepts the connection, negotiates the socket parameters and, when the run is declared started by the ***runControl*** system, begins to poll the channel for incoming data. Whenever new data is available, this is accepted and stored in the event-builder data buffer. When the event is completely readout, the ***eventBuilder*** takes the appropriate action. Once the event is completed, it is moved either to the recording stage or to the next processing stage, according to the configuration database.

If the ***eventBuilder*** runs out of memory, it stops accepting data from the LDC(s). Thanks to the backpressure applied by the TCP/IP socket library, this stops the recording process on the LDC(s) - at least for what concerns this particular ***eventBuilder*** (if the ***recorder*** process has multiple channels active, recording on the LDC can continue on other free channels).

12.2.2 The communication protocol between the LDC and the GDC

The communication protocol between the LDC ***recorder*** and the GDC ***eventBuilder*** is based on the DATE data format (see Section 3.5). For each event the following operations are performed:

- the ***eventBuilder*** reads the event header. On the basis of the event header, the ***eventBuilder*** knows the event type, the event number and the event length.
- the validity of the header is checked:
  - ***magic word*** field and number of bytes effectively read compared to standard header length. If the event header is incorrect, an error is issued to the ***infoLogger*** and a special event header is created with the type ***EVENT_FORMAT_ERROR***.
- statistics are accumulated on the different event types.
- data is read into the ***eventBuilder*** data buffer. If the data is truncated, an error bit (***EVENT_DATA_TRUNCATED***) is added into the event type.

The cycle is repeated until the run is declared closed and either the LDC closes its channel or an abort (quick exit) sequence is started.

12.2.3 The communication protocol between the eventBuilder and the edm

The communication protocol between the ***eventBuilder*** and the ***edm*** is unidirectional, from the ***eventBuilder*** to the ***edm***. The messages exchanged are two: ***EDM_MESSAGE_NEARLY_FULL*** and ***EDM_MESSAGE_NEARLY_EMPTY***. Both messages must be terminated by the character ***EDM_MESSAGE_SEPARATOR***. All these constants are defined in ***${DATE_EDM_DIR}/edm.h***. The communication channel is set in asynchronous mode, therefore it is impossible to block the ***eventBuilder*** (if the channel between the ***eventBuilder*** and the ***edm*** becomes busy, the ***eventBuilder*** queues the messages that will be sent whenever the channel becomes free again).

12.2.4 The event-building process

The process of building the event is based on the header of the incoming event and from the directives recorded in the event-building control database.

The decision is always based on the ***eventType*** field of the event header. It can be refined by the ***eventDetectorPattern*** field (the set of detector(s) selected to readout the given event) or by the ***eventTriggerPattern*** field (the set of trigger(s) active for the given event). The first rule that matches the event is selected. The rule decides if the event should be built (the event includes all LDCs

contributing to a given event) or not built (one LDC event per GDC event). If no rule can be applied to a given event, the ***eventBuilder*** requests an end of run with error condition.

If the detector pattern stored in the event header enables a subset of the detectors included in the run, the ***eventBuilder*** will expect data only from those detectors. Therefore a "full build" rule may become a "partial build" action whenever a partial readout was performed.

When running with the HLT, the HLT response may be used to change the building rule as the HLT response can enable or disable individual LDCs. The ***eventBuilder*** takes the HLT response into consideration and changes the event-building policy accordingly, expecting data only from those LDCs whose readout has been enabled.

12.2.5 SOR/EOR records, files and scripts

The ***eventBuilder*** handles SOR and EOR records, files and scripts using the same method used by ***readout***.

Please note that hosts running both as LDC and as GDC transfer the same files and execute the same scripts (common and specific) twice, once as LDC and once as GDC. SOR and EOR scripts can differentiate their actions by testing the environment variable ***${DATE_HOST_ROLE}*** (which is set to "`ldc`" if the script is being called by the ***readout*** process and to "`gdc`" if the script is being used by the ***eventBuilder*** process).

12.3 Data buffers

The ***eventBuilder*** must be able to perform its main function: to put together sub-events belonging to the same event. To do this, it must ensure to have enough memory available for each LDC to build at least one event.

The ***eventBuilder*** is given statically one memory bank. This bank is dynamically partitioned into two sections: the per LDC section and the public section. The per LDC section is further partitioned into one sub-section for each of the LDCs connected at start of run. The public section is available to all LDCs.

The partition between public and private pools is done using compilation-time configuration parameters and run-time dymanic parameters. The actual values and the usage of all pools is available as a statistics record sent via the ***logBook*** stream, as described in Section 12.7.3.

The ***eventBuilder*** refuses to run if the available memory is too small to allocate a minimum number of events. The check is made at start of run time using the maximum event size as declared by the run control for the LDCs connected to the ***eventBuilder***. If the available memory is not sufficient, the ***eventBuilder*** sends an appropriate error message and requests an end of run with error. It is then up to the data-acquisition system administrator to provide bigger memory buffers, as requested by the ***eventBuilder*** via the ***eventBuilder*** stream.

The event-builder data buffer can be implemented using any of the available supports. This includes the process ***HEAP***; note that, in this case, allocation is performed once at start of run time (and not on demand) and that the only possible recording option is direct recording (events cannot be shared with post-processing stages as memory transfer is impossible). ***BIGPHYS/PHYSMEM*** are also allowed and may actually perform better than IPC seen the different approach of the operating system between the two methods (less overheads, no swapping, less conflicts, different resources).

When post-processing is requested, the ***eventBuilder*** must be given space for its ready fifo. This can be done either by declaring a bank dedicated to all ***eventBuilder*** resources or by using two banks, one for ***eventBuilderReadyFifo*** and the second for ***eventBuilderDataPages***. The second method ensures a better tuning of the two resources and eventually allows the use of two different support methods(e.g. IPC for the ***eventBuilderReadyFifo*** and ***PHYSMEM*** for the ***eventBuilderDataPages***).

12.4 Consistency checks on the data

The ***eventBuilder*** checks the data that is sent to it. This operation can reveal fatal errors originated in the readout electronics or readout software. In case of error, this is signaled through the ***infoLogger*** and the ***eventBuilder*** requests the run control to stop the run. Furthermore, a rule of the event-building database must match every given event. This rule can specify a subset of detectors (either directly or indirectly via the ***eventTriggerPattern*** and ***eventDetectorPatterns*** fields) in which case all LDCs belonging to the subset must contribute to the event. If no subset is specified, all LDCs are supposed to contribute to the event with a (possibly empty) sub-event.

12.5 ALICE events emulation mode

The ***eventBuilder*** can run is a special ALICE events emulation mode. Target of this special function is to emulate as close as possible to the behavior of a production GDC when running in ALICE production. When this mode is selected, the data from each LDC is taken individually and unpacked as if it would come from several LDCs. The payload of the event must contain a real ALICE event, with one event header (coming from a GDC) and one or more events headers, equipment headers and payload. All this can be either built or extracted as is from an event recorded during a previous run. The result of this operation is an event that closely looks like its equivalent produced by the original LDCs (the only differences are the event ID, the event timestamp and the fact that the event data comes from consecutive memory blocks rather than from several scattered memory locations). The event-builder refuses payloads that do no match this structure and aborts the run with error as soon as this happens.

This special running mode can be selected by setting the environment variable ***DAQ_EMULATE_ALICE_EVENTS*** to the value `111`. This can be done either by

asserting a DAQ-wide variable or by going machine by machine. Special care must be taken not to use this mode in production setups (even though the event-builder will most likely abort the run due to the inconsistent format of the event payload).

12.6 The control of the eventBuilder

The ***eventBuilder*** is running under the control of the ***runControl*** system. In the GDC, the ***rcServer*** process is responsible for maintaining the control shared segment and for allocating the required memory buffers at start of run time.

The ***eventBuilder*** cannot control detached processes (such as a post-processing stage). As for any other DATE process, this function must be delegated to the DATE ***runControl*** system.

12.7 Information and error reporting

12.7.1 Usage of the infoLogger

The ***eventBuilder*** uses the DATE ***infoLogger*** package (see Chapter 11) to report statistics, information and error messages.

12.7.2 Run statistics update

The ***eventBuilder*** updates regularly the information stored in the ALICE LogBook using the statistics update routines which are part of the DATE LogBook package (see Section 24.3).

12.7.3 End-of-run messages

At the end of the run, the ***eventBuilder*** updates the run log with run statistics, warning and error messages. Run statistics include memory usage, timers, counters, event-building rules usage, per LDC counters and run-time performances.

12.8 Configuration

The ***eventBuilder*** should be configured statically (event-building rules, memory banks) and dynamically (run control).

The event-building static configuration is described in Section 4.3.5.

For the memory banks, the ***eventBuilder*** must have a data buffer of sufficient space to be able to perform its function. The declaration should be done as described in Section 4.3.6. Please note that the `maxEventSize` parameter for the ***eventBuilder*** does not apply to the events coming from the LDCs: here the individual `maxEventSize` (as declared for each of the LDCs) applies both for configuration purposes and for run-time checks. Only the events created by the ***eventBuilder*** itself (e.g. SOR records, SOR files etc...) use the `maxEventsSize` parameter.

13 The event distribution manager

This chapter describes the DATE Event Distribution Manager software package (EDM). The event distribution is the process of distributing all the sub-events produced by the same trigger to a single destination machine (GDC). It allows a smooth GDC load balancing and consists of three different processes, two of which are running on each LDC and one is running on a machine, called ***edmHost***.

The next pages include a description of the event distribution manager architecture and describe the ***edmAgent*** and the ***edmClient*** processes in the LDC, as well as the ***edm*** process in the ***edmHost***. These processes are needed to activate the event distribution mechanism towards the event builder software running in the GDC machines.

13.1 Overview

The DATE event distribution manager (EDM) is a software package responsible for the distribution of the parallel readout streams coming from the different LDCs and belonging to the same trigger, called sub-events, to a single destination machine, called GDC. On the GDCs, the event builder package is responsible for building the sub-events to form a single event, defined as the collection of data pertaining to the same particle collision. In a DAQ system there may be several GDCs performing event building functions, all connected to the same switching network, supporting TCP/IP protocol, to which all the LDCs and the ***edmHost*** machine are also connected. The EDM system allows the distribution of sub-events across the GDCs, all the physics and calibration events being randomly distributed to all GDCs regardless of the trigger class they belong to. Special event types, such as **START_OF_RUN**, **START_OF_RUN_FILES**, **END_OF_RUN**, **END_OF_RUN_FILES**, **START_OF_BURST**, **END_OF_BURST** are always sent to the first GDC which declared itself to the ***edm*** process. No sub-events are broadcast to all the GDCs. The actual transfer of sub-events is executed by the ***recorder*** process (see Section 10.3) in the LDC, when it is configured to use one or multiple GDCs as output device. The choice of the destination GDC is taken following the instructions given by the ***edm*** process. This mechanism permits a smooth GDCs load balancing and makes it possible to adapt at run-time the data flow to the capabilities of the GDCs. It excludes from the system the GDCs which are too overloaded and puts them back as soon as they are free. In a similar way, if a GDC goes down for whatever reason, the data acquisition does not stop, but simply removes this GDC from the list of possible destinations for the sub-events, thus avoiding hang-ups in the data-acquisition system. As soon as the GDC is up and running again, it is re-inserted in the list of possible destinations for the sub-events.

The user can choose to perform a run with or without the EDM software by means of the checkbutton labelled **EDM** in the run control main window. In a system where only one GDC is available, it does not make any sense to activate the EDM software. In case the **EDM** checkbutton is not selected, the run control does not start any ***edm*** related process. In this case, the event distribution algorithm is performed by the ***readout*** process (see Section 6.1) running in the LDCs through a simple scheme of sub-event distribution in a round-robin fashion, independent of any distributed knowledge about the GDC status. The destination GDC for each event is set in the field ***eventGdcId*** of the event header, based on the total number of GDCs and on the eventId of the sub-event. The dispatch algorithm uses the ***eventId*** field which is mapped to an actual GDC by means of a hash table, whose function is to avoid periodicities introduced by non-uniform distribution of the ***eventId*** field. The same mechanism, combined with the availability of the GDCs, is also used by the **EDM** via a library shared between the **readout** and the **EDM** packages .

The EDM software is running under the control of the ***runControl*** system (see Chapter 14) from which it receives the commands to start and stop the run and the parameters needed for a run. The ***infoLogger*** functions (see Chapter 11) are used to report messages.

13.2 The EDM architecture

The EDM architecture in shown in Figure 13.1

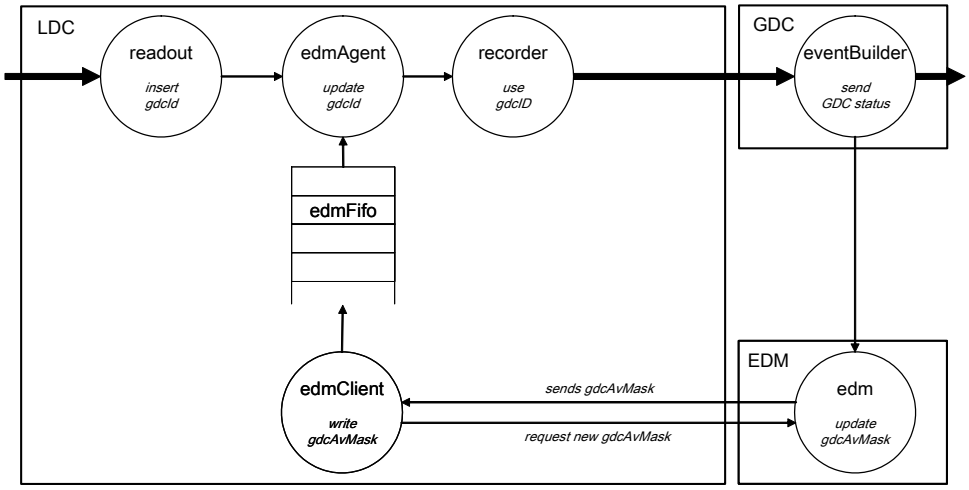

Figure 13.1 The EDM architecture.

The EDM software includes the three following processes, launched by the run control:

- the ***edm*** in the ***edmHost***.
- the ***edmAgent*** on each LDC.
- the ***edmClient*** on each LDC.

The ***edm*** process keeps track of the list of available GDCs. It receives the status of each GDC from the ***eventBuilder*** process, which sends the following messages to the ***edm***:

- nearly full: this message declares the GDC on which the ***eventBuilder*** is running as unavailable, therefore to be removed by the ***edm*** from the list of available GDCs.
- nearly empty: this message declares the GDC on which the ***eventBuilder*** is running as available, therefore to be added by the ***edm*** to the list of available GDCs.

The threshold in percentage below which the event builder sends the message “nearly empty” and the one above which the ***eventBuilder*** sends the message “nearly full” can be chosen for each run via the ***ebNearlyEmpty*** and ***ebNearlyFull*** run parameters.

The ***edm*** builds a **GDC availability mask**, which contains the list of available GDCs, the first and the last eventId for which the mask is valid, respectively called ***firstEventId*** and ***lastEventId***. To calculate these two eventIds, the ***edm*** uses the ***edmInterval*** run parameter, which indicates the size of the validity range for a mask, in units of eventId.

The structure of the GDC availability mask is the following:

```
struct edmMask_t {
   eventIdType firstEventId;
   eventIdType lastEventId;
   V32 mask [GDC_AVMASK_NGROUPS];
};
```

where mask is an array of 32 bit integers, as many as needed to accommodate the highest bit corresponding to the maximum GDC identifier that has been declared in the configuration data base (see Chapter 2).

The user has to configure the minimum fraction of GDCs that should be free for the system to continue the run without waiting. This is done by means of the ***edmQuorumPercent*** run parameter, which indicates the percentage of GDCs that should be available before sending the GDC availability mask to all the LDCs. With the exception of the first mask, sent at start of run, the ***edm*** sends the GDC availability mask to all the LDCs only if the number of available GDCs is bigger than the minimum quorum requested.

Since the ***edm*** process does not have any knowledge on the eventId of the sub-event being processed in the LDC, it must be instructed by the LDCs when time has come to send a new GDC availability mask. In order to reduce the dead time, the LDC tells the ***edm*** to send a new mask in advance with respect to the last event ID for which the previous mask is valid. The LDC signals to the ***edm*** that a new GDC availability mask is needed when it reaches the bottom range of the validity for a mask, which is set by the user as ***edmDelta*** run parameter. In practice the LDC issues a request for a new mask when it is processing the event whose eventId is equal or higher to `lastEventId - edmDelta`.

All the LDCs tag the sub-events with an increasing event ID, which is the same for all the sub-events belonging to the same trigger and is recorded in the field ***eventId*** of the event header. The monotony for the event ID is checked for all sub-events of type PHYSICS_EVENT: each event ID must be higher than the event ID of the previous sub-event. If this is not the case, an error message is issued to the ***infoLogger*** and the ***readout*** process asks to stop the run.

In order to avoid TCP / IP socket connections in the LDC processes responsible for the main data flow of the sub-events, the software in the LDC has been organized in two separate processes: the ***edmClient*** and the ***edmAgent***.

The ***edmClient*** is responsible for the TCP / IP communication with the ***edm***. It receives the GDC availability mask from the ***edm*** and it sends to the ***edm*** the request to get a new GDC availability mask.

The ***edmAgent*** process running in the LDC takes a sub-event from the ***readout*** FIFO (where it has been inserted by the ***readout*** process), reads the GDC availability mask from the ***edm*** FIFO located in the shared control region in the LDC and inserts in the header of the physics sub-event the destination GDC. Then it passes the sub-event on to the ***recorder*** process for the actual dispatching of it to the ***eventBuilder*** process on the destination GDC. Only event descriptors are read and passed on: there is no memory-to-memory copy involved. The decision on the destination GDC is taken by each LDC independently from each other, using data-driven algorithm, based on the ***eventId*** field of the sub-event header. The algorithm forbids sending sub-events to all the GDCs declared as unavailable by the ***eventBuilder*** during the validity range of the GDC availability mask.

The communication between the ***edmClient*** and the ***edmAgent*** process running on the same LDC happens through flags in the shared control region and the ***edm*** FIFO.

13.2.1 The edm process

The ***edm*** process is responsible for creating and maintaining the GDC availability mask, as well as for sending it to all the LDCs participating in the run. When the run is starting, the ***edm*** process waits for connection declarations. LDCs can only connect once, because the run can't continue if one LDC disconnects or breaks down, while the GDCs can connect and disconnect at any time during the run. When receiving the first connection request from an LDC or GDC, the ***edm*** negotiates some socket parameters for the connections, adds the GDC to the list of available GDCs and sets the first validity range of event IDs in the GDC availability mask. Once all the LDCs and all the GDCs are connected, the ***edm*** begins to poll as many channels as the number of GDCs and LDCs which participate in the run with a timeout, specified by the user as ***edmPollTimeOut*** run parameter (expressed in milliseconds). This allows the ***edm*** to periodically check for the arrival of end of run or abort run commands.

The ***edm*** updates the GDC availability mask when:

- a GDC connects or sends the **nearlyEmpy** message: the ***edm*** adds it to the list of available GDCs.
- a GDC disconnects or sends the **nearlyFull** message: the ***edm*** removes it from the list of available GDCs.

Before sending the GDC availability mask, the ***firstEventId*** and the ***lastEventId*** fields of the structure of type ***edmMask_t*** are updated as follows:

- `firstEventId = lastUpperBoundSent + oneEventId`

  where ***lastUpperBoundSent*** is the lastEventId field of the last mask sent, and ***oneEventId*** is 1 event number (nbInRun) or 1 bunch crossing depending on the mode of running (i.e. on the ***colliderMode*** run parameter).

- `lastEventId = max (firstEventId, maxWakeUpId) + edmInterval`

  where ***firstEventId*** is the firstEventId field, calculated above, and ***maxWakeUpId*** is the highest event ID for which a request for a new mask has been received. Each request for a new mask is accompanied by the event ID of the sub-event at which the LDC issuing the request discovers that it needs a new mask.

With the exception of the first mask, which is sent when all the machines participating in the run have connected to the ***edm***, the ***edm*** sends the GDC availability mask when:

- a wakeUp message is received from LDC(s) accompanied by the event ID for which the request has been issued.
- a GDC connects and there was already a request for mask pending and the quorum, not reached before, is now reached.
- when a GDC sends **nearlyEmpty** and there was already a request for mask

pending and the quorum, not reached before, is now reached.

After sending the GDC availability mask, the following variables are saved in the shared control region, so that they can be displayed by the run control status display:

- `lastThresholdSent = lastEventId - edmDelta`
- `lastUpperBoundSent = lastEventId`

In order to avoid sending the mask for multiple wakeUp messages received by different LDCs, and therefore increasing the network traffic, the actual sending of the mask happens only if the event ID for which the request is made is higher or equal than the last threshold sent, i.e. if the requested mask has not already been sent.

The ***edm*** asks to stop the run if not all the LDCs are connected; this check is made when the run is starting.

13.2.2 The edmClient process

In the initialization phase, the ***edmClient*** process connects to the ***edm***, after getting the port for the connection from the environment variable **DATE_SOCKET_EDM**, negotiates the socket parameters for the connection, and declares itself to the ***edm***.

It then polls the input channel to receive the GDC availability mask from the ***edm***. After having performed some checks on its validity, it writes the GDC availability mask in the LDC shared control region. It periodically checks on one side for run control commands and on the other side for the value of the flag ***wakeUpRequestFlag*** in the LDC shared control region to know whether it has to send a request for a new mask to the ***edm***.

The possible values of the ***wakeUpRequestFlag*** are:

- **EDM_REQUEST_FLAG_REQ**: set by the ***edmAgent*** when a new mask is needed
- **EDM_REQUEST_FLAG_SENT**: set by the ***edmClient*** after sending the request for a new mask.

Each request for a new mask is accompanied by the event ID for which the request is issued. This allows the ***edm*** to discard multiple requests of GDC availability mask for the same validity range coming from several LDCs, which may happen since not all the LDCs participate in all the events.

13.2.3 The edmAgent process

In the initialization phase, the ***edmAgent*** reads the first GDC availability mask from the ***edmFifo*** in the control region. When the run is started, it takes every sub-event descriptor from the ***readout*** process and performs some actions depending on the event ID. There are three main cases:

1. The event ID is higher than the ***lastEventId*** field of the GDC availability mask:

- the ***edmAgent*** tries to get the new GDC availability mask from the ***edmFifo***.
- if a new mask is available in the ***emdFifo*** it uses it to set the destination GDC and calculates the threshold of the current mask as

  `currentThreshold = lastEventId field - edmDelta`.
- if a new mask is not available in the ***edmFifo***, the ***edmAgent*** checks if the request for a new mask is already pending, in which case it waits for the new mask to be written in the ***edmFifo*** by the ***edmClient***. The waiting time expressed in microseconds as ***recMaskSleepTime***.
- if there is no pending request for a new mask, the ***edmAgent*** checks whether the LDC on which it is running is the one which has to issue the request. In order to avoid that all the LDCs participating in the same trigger issue a request for a new GDC availability mask, the following algorithm has been implemented: only the LDC whose identifier is the lowest identifier involved in the event instructs the ***edmClient*** to issue the request.
- in case the request for a new mask is issued by setting the ***wakeUpRequestFlag*** to **EDM_REQUEST_FLAG_REQ** in the shared control region, in such a way that the ***edmClient*** process running in the same LDC can do the actual request to the ***edm*** via the socket library.
- when the mask is available, it fills the ***eventGdcId*** field of the header with the destination GDC, returned by the distribution algorithm.

2. The event ID is between the current threshold and the ***lastEventId*** field of the GDC availability mask:
   - if there is already a pending request, simply fills the ***eventGdcId*** field of the header with the destination GDC, returned by the distribution algorithm.
   - if there is no request for a new mask pending and no mask is available in the ***edmFifo***, the ***edmAgent*** checks whether the LDC on which it is running is the one which has to issue the request; given that is the case it issues the request for a new mask just as in the previous case.
3. The event ID is smaller than the current threshold, just fill the ***eventGdcId*** field of the header with the destination GDC, returned by the distribution algorithm.

The cycle is repeated until the run is declared as stopped and either the ***edm*** closes its channel or an abort (quick exit) sequence is started.

The ***edmAgent*** is performing various checks on the GDC identifiers of the GDC availability mask (for example if the GDC identifiers are compatible with the ones declared in the configuration database) and, in case of error, asks to stop the run.

13.3 The synchronization with the run control

The EDM software is running under the control of the ***runControl*** system. The run is declared as started only after the completion of the following sequence of operations:

- all the GDCs and the LDCs declare themselves to the ***edm***.
- the ***edm*** sends the first GDC availability mask to all the ***edmClient*** processes running on the LDCs.
- the ***edmClient*** writes it into the ***edmFifo*** in all the LDCs.
- the ***edmAgent*** reads it from the ***edmFifo*** and sets it as the current GDC availability mask in all the LDCs.

13.4 Information and error reporting

The EDM uses the DATE ***infoLogger*** package (see Chapter 11) to report statistics, information and error messages.

The runControl

14

This chapter describes the architecture of the runControl system, its various components, and their interactions.

14.1 Introduction

Within a DAQ system, several data acquisitions can be performed at the same time: this is the case, for example, of several detectors independently collecting calibration data. Every data acquisition requires a configuration and the definition of parameters and options. Moreover it is performed by several processes that must be started and stopped at the right moment on many machines.

The runControl system handles the configuration and synchronization issues. It is based on Finite State Machines and it uses packages external to DATE: DIM [3], a Distributed Information Manager and SMI++ [4] are, in particular, heavily used.

14.2 Architecture

Every data acquisition, performed for one single detector or a group of detectors defined by the Experiment Control System, is controlled by a ***runControl*** process that steers the data acquisition according to operator commands. Several ***runControl*** processes with different names can run at the same time and control different data acquisitions.

Every ***runControl*** process has a runControl interface based on Finite State Machines. This interface receives all the commands sent to the ***runControl*** process and rejects those incompatible with the current status of the process. The interface also guarantees that, at any time, the source of commands is unique. It may be a given ***runControl Human Interface*** or a component of the Experiment Control System (ECS), described in the second part of this manual.

For the same ***runControl*** process, many ***runControl Human Interface***s can coexist, but at most one at a time can have the mastership of the ***runControl*** process: this last one can be used to send active control commands, whereas the others can only be used to get information. When the authorized source of commands is the ECS, none of the ***runControl Human Interface***s can send active commands: this possibility is restricted to the ECS.

When the list of machines to be used for a given data acquisition is defined, the ***runControl*** process spawns a ***Logic Engine*** process. The ***Logic Engine*** contains all the logic about starting and stopping the different processes on the different machines. The ***Logic Engine*** translates operator commands into sequences of commands that are then sent, in parallel, to the remote machines.

On every remote machine a process, called ***rcServer***, can start and stop processes according to commands received from the ***Logic Engine***s. The ***rcServer*** also performs some local error handling and returns various counters and information to the other DAQ machines. An ***rcServer*** can be used, at different times, by different ***runControl*** processes and can therefore receive commands from different ***Logic Engine***s in the context of different data acquisitions.

An interface, common to all the ***rcServer***s, guarantees that every ***rcServer*** is used at any time by at most one ***runControl*** process and receives commands

from one ***Logic Engine*** in the context of one and only one data acquisition. This interface is called ***RCS interface***.

The architecture of the runControl system is shown in Figure 14.1.

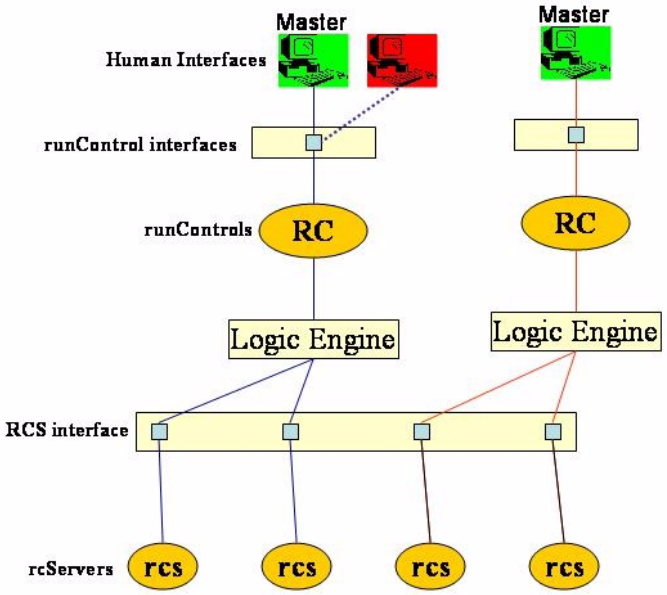

Figure 14.1 The runControl system architecture.

14.3 The runControl process

At startup time, a ***runControl*** process with name defined by the symbol ***RCNAME*** performs the following operations:

- reads the detectors and roles DATE database, applying the following restriction:
  - if ***${RCNAME}*** does not start with the string "ALL", then ***${RCNAME}*** is treated as a detector name and the ***runControl*** process loads from the DATE database the information about that detector only. For example, if the name assigned to the ***runControl*** process is TPC, then the ***runControl*** process loads information about the TPC and ignores the other detectors.
  - if ***${RCNAME}*** starts with the string "ALL", then ***${RCNAME}*** is treated as the potential name of an Experiment Control System partition prefixed with ALL. In this case, if an ECS partition with that name exists, then the ***runControl*** process loads informations about the detectors belonging to the partition and ignores the other detectors. If an ECS partition with that name does not exist, then the ***runControl*** process loads information about all the detectors. For example, the name assigned to the ***runControl*** process is “ALLITS”: if an ECS partition named ITS exists, then the ***runControl*** process loads information about the detectors of the ITS partition; if an ECS partition named ITS does not exist, then the ***runControl*** process loads information about all the detectors.

- in both cases (i.e ***${RCNAME}*** starting or not starting with the string “ALL”), if the DATE database contains information about detectors named “HLT” and “TRIGGER”, then the information about these detectors is loaded in memory knowing that HLT and TRIGGER play special roles within the data acquisition.

- sets the current DAQ configuration to an empty one, sets all the run parameters to their DATE hardcoded defaults, and resets all the run options.
- if a default DAQ configuration has been saved in the DATE database, then the default DAQ configuration is loaded and replaces the empty one. The name of the default DAQ configuration in the MySQL database is ***DEFAULT***.
- if a set of customized, default run parameters has been saved in the DATE database, then the set of customized, default run parameters is used to overwrite the DATE hardcoded defaults. The name of this set in the MySQL database is ***DEFAULT*** .
- if a set of customized, default run options has been saved in the DATE database, then the run options are set accordingly. The name of this set of run options in the MySQL database is ***DEFAULT*** .

Having completed the above sequence of operations, the ***runControl*** process sets its status to ***DISCONNECTED*** and waits for operator commands.

The main commands are the following:

- ***CONNECT***(name): loads from the DATE database the DAQ configuration with the given name and generates a new current configuration. It then creates the ***Logic Engine*** with the logic for the machines selected in the new current configuration. If the name of the DAQ configuration is ***NONE***, then the first step is skipped and the ***runControl*** process continues with the existing current configuration. If the name of the DAQ configuration is ***NEW***, then the ***runControl*** process gets the new DAQ configuration from the ***runControl Human Interface*** having the mastership of it (obviously this possibility does not exist when the source of active commands is the ECS).

  If the ***CONNECT*** command completes successfully, the ***runControl*** process locks the ***rcServer***s referenced by the newly created ***Logic Engine*** and sets its status to ***CONNECTED***.

- ***LOCK_PARAMETERS***(name): loads from the DATE database a set of run parameters with the given name. If the name is ***NONE***, then the ***runControl*** process continues with the current run parameters. If the name is ***NEW***, then the ***runControl*** process gets the new run parameters from the ***runControl Human Interface*** having the mastership of it (obviously this possibility does not exist when the source of active commands is the ECS).

  If the ***LOCK_PARAMETERS*** command completes successfully, the ***runControl*** process sets its status to ***READY***.

- ***START_PROCESSES***(name): loads from the DATE database a set of run options with the given name. It then tells the ***Logic Engine*** to start the required data-acquisition processes on all the selected machines. If the given name is ***NONE***, then the first step is skipped and the ***runControl*** process continues with the current run options. If the name is ***NEW***, then the ***runControl*** process gets the run options from the ***runControl Human Interface*** having the mastership of it (obviously this possibility does not exist when the source of active commands is the ECS).

If the source of active commands is a ***runControl Human Interface***, the ***runControl*** reads the current run number from the DAQ database, increments it and saves it back. If the source of active commands is the ECS, then the run number is defined by the ECS and transmitted to the ***runControl*** with the ***START_PROCESSES*** command.

If the ***START_PROCESSES*** command completes successfully (i.e. the ***runControl*** gets from the ***Logic Engine*** a feedback confirming that all the required processes are running), the ***runControl*** process sets its status to ***STARTED***.

- ***START_DATA_TAKING***: sends to all the selected machines (via the***Logic Engine***) the authorization to start the data taking. The ***runControl*** process then sets its status to ***RUNNING***.
- ***STOP_DATA_TAKING***: tells the ***Logic Engine*** to stop the data-acquisition processes on all the selected machines. When all the processes are stopped, the ***runControl*** process sets its status to ***READY***.
- ***STOP_PROCESSES***: has the same effect as ***STOP_DATA_TAKING.*** The only difference is that it can be issued when the actual data taking has not yet been started.
- ***ABORT_PROCESSES***: has the same effect as ***STOP_PROCESSES***. The ***ABORT_PROCESSES*** command is however stronger than the ***STOP_PROCESSES*** command and may actually kill the processes that fail responding. The ***ABORT_PROCESSES*** command can be sent from a ***runControl Human Interface*** when an error condition has activated the Abort button.
- ***UNLOCK_PARAMETERS***: unlocks the run parameters and sets the status of the ***runControl*** process to ***CONNECTED***.
- ***DISCONNECT***: stops the ***Logic Engine*** and unlocks the ***rcServer***s referenced by the stopped ***Logic Engine***. The ***runControl*** process then sets its status to ***DISCONNECTED***.

In addition to the operator commands, the ***runControl*** process gets some feedback from the ***Logic Engine***. This feedback allows the handling of requests, such as ***EndOfRun*** requests issued by processes running on the remote machines and transmitted by the ***rcServer***s to the ***Logic Engine***.

The ***runControl*** process also fills the eLogbook with information about the different runs (start time, end time, list of detectors).

Finally the ***runControl*** process acts as a DIM server and provides the following DIM services to the subscribing clients:

- ***${DAQ_ROOT_DOMAIN_NAME}${RCNAME}_DAQ::${RCNAME}_CONTROL_MESS*** : clients subscribing to this service receive the information and error messages issued by the ***runControl*** process with name ***${RCNAME}***. The ***DAQ_ROOT_DOMAIN_NAME*** environment variable is defined in the DATE database.
- ***${DAQ_ROOT_DOMAIN_NAME}${RCNAME}_DAQ::${RCNAME}_CONTROL_RUNNUMBER*** : clients subscribing to this service receive the run number being used by the ***runControl*** process with name ***${RCNAME}***. The ***DAQ_ROOT_DOMAIN_NAME*** environment variable is defined in the DATE database.

- ***${DAQ_ROOT_DOMAIN_NAME}${RCNAME}_DAQ::${RCNAME}_CONTROL_EOR***: clients subscribing to this service are notified when the run being controlled by the ***runControl*** process with name ***${RCNAME}***is finished. The ***DAQ_ROOT_DOMAIN_NAME*** environment variable is defined in the DATE database.

14.4 The runControl interface

Every ***runControl*** process has a runControl interface with two main functions:

- it rejects commands incompatible with the current status of the ***runControl*** process.
- it guarantees that, at any time, the source of commands is unique: a given ***runControl Human Interface*** or the ECS.

The runControl interface is implemented as an SMI domain containing objects required to perform the two main functions described above. It also contains a few other objects associated to minor functions, such as enabling the ***Abort*** button in the ***runControl Human Interface***s or keeping a track of the final state of the last performed data taking.

The name of this SMI domain is ***${DAQ_ROOT_DOMAIN_NAME}${RCNAME}_DAQ*** where ***RCNAME*** is a variable containing the name assigned to the ***runControl*** process at start time and ***DAQ_ROOT_DOMAIN_NAME*** is the environment variable is defined in the DATE database.

14.5 The runControl Human Interface

The ***runControl Human Interface*** may be used to send all the commands described in Section 14.3 to the ***runControl*** process. In addition to that, it allows database operations that can be performed without commands to the ***runControl*** process. Examples of operations of this second type are the definition of a default DAQ configuration, the creation of default sets of run parameters and run options, the creation of named sets of run parameters and run options to be used during special runs.

A detailed description of the ***runControl Human Interface*** is given in the ALICE DAQ WIKI.

14.6 The Logic Engine

The ***Logic Engine*** is based on the list of machines selected to play a role in the data acquisition and therefore it cannot exist as long as the DAQ configuration is

not defined. The ***Logic Engine*** is created by the ***runControl*** process when it receives the ***CONNECT*** command.

The ***Logic Engine*** receives commands from the ***runControl*** process, creates from these commands and from the ***StartOfRun*** and ***EndOfRun*** logic the sequences of commands to be executed on every remote machine, sends these commands to the ***rcServer***s running on the remote machines. It also returns to the ***runControl*** process the feedback about requests issued by processes running on the remote machines, such as ***EndOfRun*** requests.

The ***Logic Engine*** could be handled by a single SMI domain. However, in practice it is implemented as a set of SMI domains, where every domain controls a group with a limited number of remote machines. An additional SMI domain coordinates these domains and therefore the groups of remote machines. This implementation allows better usage of CPU resources and, if necessary, the cooperation of more than one PC.

The name of the top level domain is ***${DAQ_ROOT_DOMAIN_NAME}${RCNAME}_CONTROL*** where ***RCNAME*** is a variable containing the name assigned to the ***runControl*** process at start time and ***DAQ_ROOT_DOMAIN_NAME*** is the environment variable is defined in the DATE database. The other domains dealing with groups of remote machines are named ***${DAQ_ROOT_DOMAIN_NAME}${RCNAME}_CONTROL_1***, ***${DAQ_ROOT_DOMAIN_NAME}${RCNAME}_CONTROL_2***, etc.

14.7 The rcServers

An ***rcServer*** process must run on all the machines of the DAQ system where the ***Logic Engine***s need to start processes. At startup time, every ***rcServer*** creates on the local machine a shared memory control region. This region contains various flags and counters and is used as interprocess communication object by the ***rcServer*** and its children. The ***rcServer*** then waits for a ***runControl*** process needing its services.

When a ***runControl*** process executes a ***CONNECT*** command, it creates a ***Logic Engine***, locks the ***rcServer***s running on the remote machines that are part of its DAQ configuration, and starts using them. These ***rcServer***s provide to the ***runControl*** process, to its ***Logic Engine***, and to its ***runControl Human Interface***s the following services:

- start and stop processes according to commands from the ***Logic Engine***. Two types of processes are started and stopped by the ***rcServer***: processes controlled through the shared memory control region and processes that, once started interact with the ***Logic Engine*** directly. The DATE processes are processes of the first type: the shared memory control region is used by the ***rcServer*** to send commands and parameters to the processes and by the processes to issue requests, such as ***EndOfRun*** requests, and to update various flags and counters. Once started, the processes of the second type interact with the ***Logic Engine*** and are ignored by the ***rcServer***. Examples of processes of this type are the ***DDL Data Generator***s, and some synchronous process required by calibration procedures.

- perform local error handling. The ***rcServer*** continously checks that the started DATE processes are alive. When it finds a missing process, it issues an ***EndOfRun*** request.
- handle ***EndOfRun*** requests issued by the running processes.
- provide the information required by the Status Display as a DIM service.
- provide error and information messages as a DIM service.

When the ***runControl*** process executes the ***DISCONNECT*** command, it unlocks the previously locked ***rcServer***s. These ***rcServer***s start again waiting for a ***runControl*** process needing their services.

At any time, the crash of an ***rcServer*** being used by a ***runControl*** process forces the ***runControl*** process to execute a ***DISCONNECT*** command.

14.8 The RCS interface

This interface guarantees that the ***rcServer***s receive only valid commands and that every ***rcServer*** is used by at most one ***runControl*** process in the context of one and only one data acquisition.

The RCS interface is implemented as an SMI domain whose name is ***${DAQ_ROOT_DOMAIN_NAME}_RCSERVERS***.

14.9 Run parameters

This section describes the Common and role specific run parameters used by the ***runControl*** and by the sprocesses started on the various machines. The Common RunParameters are described in (Table 14.1).

Table 14.1 Common RunParameters

Name	Parameter name	Description	Default	Range
Collider mode	***colliderMode-Flag***	Defines the event ID field of the base event header • 0 = Fixed target • 1 = Collider mode ***readout*** sets the event type attribute field in the event header accordingly	1	0 , 1
Common Data Header Present	***cdhPresentFlag***	Common data header in raw data • 0 = Not present • 1 = Present	1	0 , 1

Table 14.1 Common RunParameters

Name	Parameter name	Description	Default	Range
Burst structure in Cole	***burstPresentFlag***	When Cole is used • 0 = No burst structure • 1 = Burst mode	0	0 , 1
No. of events in burst	***simBurstLength***	When Cole is used, defines number of events per burst	100	>= 0

The LDC, GDC, and EDM RunParameters are described in Table 14.2, Table 14.3, and Table 14.4

Table 14.2 LDC RunParameters

Name	runParameter name	Description	Default	Range
Max. number of sub-events	***maxEvents***	Maximum number of sub-events in a run. Zero (0) means no limit. If set, when the LDC hits the limit, an end of run request is issued.	0	>= 0
Max. bytes to record	***maxBytes***	Maximum number of bytes to be collected in a run. Zero (0) means no limit. If set, when the LDC hits the limit, an end of run request is issued.	0	>= 0
Max. number of bursts	***maxBursts***	Maximum number of bursts to be collected in a run. Zero (0) means no limit. If set, when the LDC hits the limit, an end of run request is issued.	0	>= 0
Max. number of errors	***maxErrors***	Maximum number of allowed non fatal errors.	10	>= 0
Max. event size	***maxEventSize***	It indicates the maximum size in bytes of a sub-event.	2 000 000	>= 0
Max. file size	***maxFileSize***	Each run may be recorded on multiple files. This is the maximum size of each file in bytes. • Zero (0) means no limit. • A positive value should be used only when the ***RecordingDevice*** is a disk file. This parameter is ignored if the ***Recording disabled*** run option is selected or when the recording is done only in the GDCs.	0	0-2.e9

Table 14.2 LDC RunParameters

Name	runParameter name	Description	Default	Range
Logging level	***logLevel***	It controls the generation of messages by all the DATE processes running on an LDC machine. The possible values are described in Section 14.11.	10	0 - 100
Local Recording device	***localRecordingDevice***	The setting must be done according to Section 13.2. This parameter is ignored if the ***Recording disabled*** run option is selected or when the recording is done only in the GDCs.	/dev/null	
SOR in Separate File	***sorSeparateFile***	If set to 1 the SOR event is stored in a separe file	0	0 - 1
Paged data flag	***pageDataFlag***	Defines the event data structure. Possible values: • Streamlined events (0) (see Section 5.3.1) • Paged events (1) (see Section 5.3.2)	1	0 , 1
Monitor enable flag	***monitorEnableFlag***	Switch to enable and disable the possibility of monitoring. It may introduce a penalty on the data rate performances. Zero (0) means disabled, one (1) enabled.	1	0 , 1
LDC socket size	***ldcSocketSize***	Defines the size of the socket used by the recording library (described in Section 20.4). Possible values: • 0: system default • >0: socket size set to given value • <0: socket size = MIN (***-ldcSocketSize***, ***maxEventSize***)	0	Integer
Max. time for SOR/EOR phases	***phaseTimeoutLimit***	Maximum duration (in seconds) of any phase of the start and stop procedures executed on the LDC. The run is aborted if any LDC process does not complete the phase in due time. • If zero (0) a value of 30 s is used by default.	30	0 - 600

Table 14.2 LDC RunParameters

Name	runParameter name	Description	Default	Range
Recorder sleep time	***recorderSleep-Time***	The ***recorder*** goes to sleep while events are arriving to give priority to ***readout***. The time interval (expressed in microseconds) is picked up from this parameter. If zero (0) or negative a value of 10 microseconds is used.	0	Integer
Completion sleep time	***checkComple-tionSleepTime***	Sleeptime (in milliseconds) when checking for I/O completion in ***recorder***.	1	1 - 1000
Rec sleep for mask	***recMaskSleep-Time***	Polling loop interval (in microseconds), when waiting for edm masks. If zero (0) or negative a value of 500 000 microseconds is used.	0	Integer
Max. # of sleeps for mask	***recMaskSleep-CntLimit***	Maximum number of consecutive polling loops when waiting for an edm mask, before aborting.	500	> 1
startOfData/ endOfData event enabled	***sodEodEnabled***	START_OF_DATA/END_OF_DATA event flag. If set to 0 the SOD event is disabled. If set to 1 the SOD event is enabled. if set to 2 the SOD event is enabled and all the events received before it are discarded.	1	0, 2
startOfData timeout	***startOfData-Timeout***	Timeout when waiting for START_OF_DATA events	10	>= 1
endOfData timeout	***endOfDataTime-out***	Timeout when waiting for END_OF_DATA events	10	>= 1
EDM agent enable	***edmEnabled***	EDM agent flag	1	Locked
HLT agent enable	***hltEnabled***	HLT agent flag	1	Locked
Real Hostname	***realHostname***	IP hostname		Locked

Table 14.3 GDC RunParameters

Name	runPramater name	Description	Default	Range
Max. bytes to record	***maxBytes***	Maximum number of bytes to be collected in a run. Zero (0) means no limit. If set, when the GDC hits the limit, an end of run request is issued.	0	>= 0
Max. number of errors	***maxErrors***	Maximum number of allowed non fatal errors.	10	>= 0

Table 14.3 GDC RunParameters

Name	runPramater name	Description	Default	Range
Max. SOR/EOR file size	***maxEventSize***	It indicates the maximum size in bytes of SOR/EOR file event. Please note that this parameter applies to SOR/EOR file events only.	4 000 000	>= 0
Max. file size	***maxFileSize***	Each run may be recorded on multiple files. This is the maximum size of each file in bytes. • Zero (0) means no limit. • A positive value should be used only when the ***recordingDevice*** is a disk file. This parameter is ignored if the ***Recording disabled or the Recording on PDS*** run option is selected.	0	0-2.e9
Logging level	***logLevel***	It controls the generation of messages by all the DATE processes running on a GDC machine. The possible values are described in Section 14.11.	10	0 - 100
Local Recording device	***localRecordingDevice***	The setting must be done according to Section 13.2. This parameter is ignored if the ***Recording disabled or the Recording on PDS*** run option is selected.	/dev/null	
SOR in Separate File	***sorSeparateFile***	If set to 1 the SOR event is stored in a separe file	0	0 - 1
Monitor enable flag	***monitorEnabledFlag***	Switch to enable and disable the possibility of monitoring. It may introduce a penalty on the data rate. Zero (0) means disabled, one (1) enabled.	1	0 , 1
Max. time for SOR/EOR phases	***phaseTimeoutLimit***	Maximum duration (in seconds) of any phase of the start and stop procedures executed on the GDC. The run is aborted if any GDC process does not complete the phase in due time. • If zero (0) a value of 30 s is used by default.	30	0 - 600

Table 14.3 GDC RunParameters

Name	runPramater name	Description	Default	Range
Nearly empty	***ebNearlyEmpty***	Threshold (in %) below which the ***eventBuilder*** changes its state from ***ebNearlyFull*** to ***ebNearlyEmpty***	10	0 - 100
Nearly Full	***ebNearlyFull***	Threshold (in %) above which the ***eventBuilder*** changes its state from ***ebNearlyEmpty*** to ***ebNearlyFull***	90	0 - 100
Max. number of events	***maxEvents***	This parameter is kept for future development and is not used in the present version.	0	Locked
EDM agent enable	***edmEnabled***	EDM agent flag	1	Locked
HLT agent enable	***hltEnabled***	HLT agent flag	1	Locked
Real Hostname	***realHostname***	IP hostname		Locked

Table 14.4 EDM RunParameters

Name	runPrameter name	Description	Default	Range
Max. number of errors	***maxErrors***	Maximum number of allowed non fatal errors.	1	>= 0
Logging level	***logLevel***	It controls the generation of messages by all the DATE processes running on an EDM machine. The possible values are described in Section 14.11	10	0 - 100
Max. time for SOR/EOR phases	***phaseTimeoutLimit***	Maximum duration (in seconds) of any phase of the start and stop procedures executed on the EDM. The run is aborted if any EDM process does not complete the phase in due time. • If zero (0) a value of 30 s is used by default.	30	0 - 600
Poll time out	***edmPollTimeOut***	Sleep time (in milliseconds) when polling the input channels.	100	>= 0
Quorum percent	***edmQuorumPercent***	Percentage of GDCs required to be available before new masks are sent.	50	0 - 100
edm delta	***edmDelta***	Size of the bottom range of validity for a mask, where a request for a new mask must be issued (in units of eventId).	200	>= 0

Table 14.4 EDM RunParameters

Name	runPrameter name	Description	Default	Range
GDCs validity mask interval	***edmInterval***	Size of range validity for a mask (in units of eventId).	5000	>= 0
Real Hostname	***realHostname***	IP hostname		Locked

14.10 Run-time variables

The run-time variables that can be seen via the runControl are described in Table 14.5, Table 14.6, and Table 14.7.

Table 14.5 LDC run-time variables

Name	Description
Number of equipments	Number of equipments in the 1st level vector. Set by the routine ***ArmHw.***
Number of triggers	Number of triggers. Incremented by ***readout*** at each physics event (not for the other types of events) after calling ***ReadEvent***, and then stored in eventHeader.triggerNb.
Current Trigger rate	Triggers per second since the last Status Display update. Computed by ***rcServer.***
Average Trigger rate	Triggers per second since the SOR. Computed by ***rcServer.***
Number of sub-events	Number of processed sub-events. Incremented by ***readout*** for all types of events before calling ***ReadEvent***.
Sub-event rate	Sub-events per second since the last Status Display update. Computed by ***rcServer.***
Sub-events recorded	Number of sub-events recorded by the LDC on disk, or sent to the GDCs. Set by ***recorder.***
Sub-event recorded rate	Sub-events recorded per second since the last Status Display update. Computed by ***rcServer.***
Bytes injected	Number of KB received by the LDC. Set by ***readout.***
Byte injected rate	Number of bytes injected per second since the last Status Display update. Computed by ***rcServer.***
Bytes recorded	Number of KB recorded by the LDC on disk or sent to the GDCs. Set by ***recorder***.

Table 14.5 LDC run-time variables

Name	Description
Byte recorded rate	Bytes recorded per second since the last Status Display update. Computed by ***rcServer.***
Bytes in buffer	Difference between ***Bytes injected*** and ***Bytes recorded.*** Computed by ***rcServer.***
Nb evts w/o HLT decision	Number of events waiting for HLT decision***.***
inBurst flag	This variable is set to one (1) during the burst and is reset to zero (0) otherwise. Set by ***readout***, only when the ***burstPresentFlag Common RunParameter*** is set to 1.
Recorder pid	PID of the ***recorder*** process.
Number of bursts	Set by ***readout*** at each event, after calling ***ReadEvent***, to the value found in eventHeader.burstNb.
Number of sub-events in burst	Set by ***readout*** at each event, after calling ***ReadEvent***, to the value found in eventHeader.nbInBurst.
recMaskSleepLoopCnt	Copy of internal counter of consecutive sleeps waiting for edm mask. Set by ***edmAgent.***
recMaskSleepCnt	Number of events for which an edm mask was not available. Set by ***edmAgent.***
Nb. of Readout FIFO full	Counts how many times readout has been waiting for space of the readoutReady FIFO. Set by ***readout.***
edmClient SOR/EOR phases	A number indicating the phase of the start or stop procedure executed by the ***edmClient*** process. Zero (0) means completion.
wakeUpReqFlag	Flag of communication between ***edmAgent*** and ***edmClient*** to indicate the status of a request for an edm mask. Used by ***edmAgent*** and ***edmClient.***
EventID for wakeup request	Event identifier for which a request of wake up has been triggered. Used by ***edm, edmClient, edmAgent.***
edmMaskValidityRange	Validity range for the edm mask. Set by ***edm.***
edmMask	edm mask
lastEventId	EventId monotonically increasing for all the special event types. Set in ***readout.***
readout SOR/EOR phases	A number indicating the phase of the start or stop procedure executed by the ***readout*** process. Zero (0) means completion.

Table 14.5 LDC run-time variables

Name	Description
edmAgent SOR/EOR phases	A number indicating the phase of the start or stop procedure executed by the ***edmAgent*** process. Zero (0) means completion.
hltAgent SOR/EOR phases	A number indicating the phase of the start or stop procedure executed by the ***hltAgent*** process. Zero (0) means completion.
recorder SOR/EOR phases	A number indicating the phase of the start or stop procedure executed by the ***recorder*** process. Zero (0) means completion.
Machine identifier	Machine identifier as defined in the DATE roles database.
Spare variable 1 - 10	Reserved for DATE developers.
Spare String 1 - 5	Reserved for DATE developers.
Site Spec 1 - 5	Reserved for DATE developers.
Site Spec String 1-2	Reserved for DATE developers.

Table 14.6 GDC run-time variables

Name	Description
Number of sub-events	Number of processed sub-events. Incremented by ***eventBuilder***
Sub-event rate	Sub-events per second since the last Status Display update. Computed by ***rcServer.***
Events recorded	Number of events recorded on disk by the GDC. Set by ***eventBuilder.***
Event recorded rate	Events recorded per second since the last Status Display update. Computed by ***rcServer.***
Bytes recorded	Number of KB recorded on disk by the GDC. Set by ***eventBuilder***.
Byte recorded rate	Bytes recorded per second since the last Status Display update. Computed by ***rcServer.***
File count	Number of files recorded in the current run. Set by the ***eventBuilder***.
Nb. times nearly full	Number of times the ***eventBuilder*** has declared itself nearly full. Set by the ***eventBuilder*** (internal counter).

Table 14.6 GDC run-time variables

Name	Description
Nb. times nearly empty	Number of times the ***eventBuilder*** has declared itself nearly empty. Set by the ***eventBuilder*** (internal counter).
Nb. of incomplete events	Number of incomplete events. Set by the ***eventBuilder*** (internal counter).
eventBuilder pid	PID of the ***eventBuilder*** process.
Status of EVB vs. EDM	String describing the status of the ***eventBuilder*** sent to the ***edm*** (***edm*** active only). Set by ***eventBuilder.***
eventBuilder SOR/EOR phases	A number indicating the phase of the start or stop procedure executed by the ***eventBuilder*** process. Zero (0) means completion.
Machine identifier	Machine identifier as defined in the DATE roles database.
Spare variable 1 - 10	Reserved for DATE developers.
Spare String 1 - 5	Reserved for DATE developers.
Site Spec 1 - 5	Reserved for DATE developers.
Site Spec String 1 - 2	Reserved for DATE developers.

Table 14.7 EDM run-time variables

Name	Description
FIFO size	Size of edm mask FIFO.
EDM FIFO full count	Number of times the edm mask FIFO is full. Set by ***edmClient***.
EventID for wakeup request	Event identifier for which a request of wake up has been triggered. Used by ***edm, edmClient, edmAgent.***
maxWakeUpId	The highest event identifier for which a request of wake up has been triggered. Set by ***edm.***
Last validity range threshold	Last validity range threshold sent. Set by ***edm.***
Last validity range upper bound	Last validity range upper bound sent. Set by ***edm.***
edmMaskValidityRange	edm mask validity range
edmMask	edm mask
Unavailable GDCs	List of GDCs not present in the edm mask. Computed by ***rcServer.***

Table 14.7 EDM run-time variables

Name	Description
edm SOR/EOR phases	A number indicating the phase of the start or stop procedure executed by the ***edm*** process. Zero (0) means completion.
Machine identifier	Machine identifier as defined in the DATE roles database.
Spare variable 1 - 10	Reserved for DATE developers.
Spare String 1 - 5	Reserved for DATE developers.
Site Spec 1 - 5	Reserved for DATE developers.
Site Spec String 1 - 2	Reserved for DATE developers.

14.11 Control of the log messages

All DATE packages create log messages for information, error reporting and statistics purposes. It is possible to control, on a site-by-site and role by role basis, the amount of information generated during DATE operation.

The run parameter ***loglevel*** is used to check the amount and the details of the messages sent via the DATE ***infoLogger***. The following conventions are commonly followed by all the DATE processes:

- ***loglevel*** equal to 0: no statistics and no logging (even in case of error).
- ***logLevel*** above 0: run statistics are appended to the **logBook** stream and error messages are sent to the **runLog** and package-specific streams.
- ***loglevel*** between 10 and 19 (***LOG_NORMAL_TH*** and ***LOG_DETAILED_TH***-1): normal level of logging. Run statistics, errors and information messages are created. This is the level which should be used in normal conditions and during data taking.

logLevel of 20 (***LOG_DETAILED_TH***) and above: level of logging used for the development of the DATE software. A lot of messages are produced and in particular, some messages are produced for each event. This level has a dramatic effect on the performance and should therefore not be used during normal data taking.

14.12 Log Files

Error and information messages are logged by all the components of the ***runControl*** system using the ***infoLogger*** (Section 11.5). However, the ***stdout*** and ***stderr*** streams of the processes are stored as temporary files in the ***${DATE_SITE_TMP}*** directory. These files are usually read only by the DATE developers and are overwritten when the associated processes are restarted. In

some cases, the previous version of the log file is renamed to allow more debugging.

All these temporary log files have a `.stdout` postfix and a name that follows the conventions listed below:

- log files created by SMI domains have a name in upper case characters, equal to the name of the SMI domain.
- log files created by ***runControl*** processes have a name that is the concatenation of the ***runControl*** process name and the fixed string '`_control`'. The name is in lower case characters.
- log files created by ***runControl Human Interface***s have a name that is the concatenation of the ***runControl*** process name, the fixed string '`_controlhi`', and the process identifier (PID) of the ***runControl Human Interface***. The name is in lower case characters.
- log files created by ***rcServer***s have a name that is equal to the symbolic name assigned in the DATE database to the component of the DAQ system where the ***rcServer*** runs. If the symbolic name of an LDC is `pcxxldc`, then the ***rcServer*** running on it creates a '***pcxxldc.stdout***' log file. The name is in lower case characters.

The physmem package

15

This chapter describes the DATE package ***physmem***. The package contains a Linux kernel module to support shared access to a large contiguous block of non-paged physical memory which has been reserved at boot time. The installation procedure, a description of the utility programs, and some information about the kernel module implementation are presented.

15.1 Introduction

The supporting mechanism of a memory bank required by each DATE host can be either of type **IPC**, **PHYSMEM**, **BIGPHYS**, or **HEAP** (see Chapter 4). If a memory bank needs to be large and non-paged, the best choice on the Linux operating system is **PHYSMEM**. Furthermore, a memory bank of type **PHYSMEM** allows to obtain the physical base address of this memory, which is mandatory for the RORC readout software (see Chapter 7).

Since a Linux operating system does not provide a mechanism to allocate and deallocate large amounts of non-paged memory, this chapter describes the ***physmem*** approach that was developed for DATE. It exploits the feature that a separate block of memory can be reserved at boot time, which is not seen by Linux. An additional kernel module driver is needed to access this ***physmem*** memory. In comparison to the ***bigphysarea*** approach, which is based on the same principles, the ***physmem*** approach does not rely on specific Linux kernel patches.

The DATE package ***physmem*** provides all the necessary software to make the ***physmem*** memory available. This package is self-contained and resides in the ***${DATE_ROOT}/physmem*** directory. The following documentation describes the installation procedure of the ***physmem*** kernel module (see Section 15.2), the usage of the utility programs (see Section 15.3), and gives some information about the ***physmem*** kernel module internals (see Section 15.4).

15.2 Installation of the physmem driver

This section provides a guide to install and configure the software in order to access the ***physmem*** memory on a machine that runs a Linux operating system with a 2.4 and 2.6 kernels. The installation of the ***physmem*** driver is part of the basic DATE installation.

15.2.1 Configuring the boot loader

As a first step, the physical memory of a machine must be partitioned into a region for Linux and a region for ***physmem***. This can be achieved during the boot process of a Linux operating system by passing the ***mem*** parameter to the kernel. This parameter defines the size of the Linux memory region, whereas the remaining memory can be used for ***physmem***. For example, if a machine has 4.5 GB of memory installed and the *mem* parameter is set to 1024M, then the Linux memory region encloses 1 GB and the ***physmem*** memory region gets the remaining 3.5 GB. However, the Linux operating system does not always set precisely this memory boundary as requested by the *mem* parameter.

The boot loader of a Linux operating system can be used to pass the *mem* parameter to the kernel. In case of boot loader GRUB, Listing 15.1 shows an example of its configuration file ***/etc/grub.conf*** to trim the memory region for Linux to 1 GB (line 3). In case of boot loader LILO, Listing 15.2 shows an example of the

configuration file ***/etc/lilo.conf*** to trim the memory region for Linux to 1 GB (line 4).

Listing 15.1 Example of GRUB to trim the Linux memory region to 1 GB

```
1:  title Red Hat Linux (2.4.21-4)
2:    root (hd0,0)
3:    kernel /boot/vmlinuz-2.4.21-4 ro root=/dev/hda1 mem=1024M
4:    initrd /boot/initrd-2.4.21-4.img
```

Listing 15.2 Example of LILO to trim the Linux memory region to 1 GB

```
1:  image=/boot/vmlinuz-2.4.21-4
2:    label=linux2421-4
3:    initrd=/boot/initrd-2.4.21-4.img
4:    append="mem=1024M"
5:    read-only
6:    root=/dev/hda1
```

After rebooting the machine, the memory region for Linux will be reduced to the value given in the respective configuration file. This can be verified by issuing the following command:

```
> cat /proc/meminfo
```

15.2.2 Setting up the physmem driver

The ***physmem*** memory is represented by the two device files ***/dev/physmem0*** and ***/dev/physmem1***, both with major device number *122*. Device file ***/dev/physmem0*** with minor device number *0* is exclusively used by the RORC utility programs (see Chapter 20) and the assigned ***physmem*** memory has a default size of 96 MB. Device file ***/dev/physmem1*** with minor device number *1* is used for a memory bank of type ***PHYSMEM***. The assigned memory for each ***physmem*** device is a separate block of the physical memory.

The ***physmem*** software package is installed together with the DATE kit. However, if someone wants to use the stand-alone DDL and RORC software (described in Chapter 20) he has to install the ***physmem*** driver and library. For a stand-alone installation, follow the given procedure below:

- The header, source, object and executable files of the ***physmem*** driver, library and test programs are in the common AFS area:

  `/afs/cern.ch/alice/daq/ddl/physmem/`

  This directory contains the different versions of the software as separate sub-directories. It also contains the different versions in compressed formats.
- The compressed file names show the version number and the time of archiving. Always use the latest date of a given version. The latest distributed version can be found on the following Web page:

  http://cern.ch/ddl/rorc_support.html

- Copy the compressed file on a local directory and uncompress it. Use the

following command for extracting the files:

```
gtar -xvzf physmem_vx.y_year.month.day.tgz vx.y/
```

where `x.y` is the version number of the package.

- To do the driver, library and utility compilation, type the following commands:

```
cd vx.y
make -f Makefile clean
make -f Makefile
```

- To create the device files, and prepare the driver to be loaded at boot time type as ***root*** the following commands
  `make -f Makefile dev`
  This command creates the device files, loads the driver module to the appropriate place and edits the ***/etc/rc.modules*** file for automatic loading of the module at boot time. If one wants to give a parameter to the driver, she/he can modify this file.
- To load the ***physmem*** driver kernel module without booting type as ***root***:

```
make -f Makefile load
```

- In case an older version of the ***physmem*** driver is already loaded, then type as user root:

```
make -f Makefile reload
```

- To check if the driver is loaded type:

```
./check_driver.
```

  This script shows if the driver is loaded (calling `/sbin/lsmod`) and the driver messages during load time (calling `dmesg`). Listing 15.3 gives an example dialog in which the memory regions for ***physmem*** and for Linux are 3220504576 bytes (786256 pages) and 1073741827 bytes (262144 pages) respectively. In this example the assigned memory to ***/dev/physmem0*** is 100663296 bytes (the default 96 MB) starting at physical address 0x40000000, and the assigned memory to ***/dev/physmem1*** is 3119841280 bytes (2975 MB), starting at physical address 0x46000000. There is a “gap” in the ***physmem1*** memory between the physical addresses 0xcff50000 `to` 0x100000000, which is assigned to other devices by the BIOS, dividing ***physmem1*** memory into 2 zones.

Listing 15.3 Example to list the physmem physical base addresses and sizes

```
 1: > ./check_driver
 2: lsmod:
 3: Module                  Size  Used by
 4: physmem                44680  0
 5:
 6: dmesg:
 7: physmem: loading driver version 4.13 with physmemsize=0 and
    physmemsize0=0 (all in MB)
 8: physmem: linux version code = 2.6.18 (SLC 5.4), instruction set =
    64 bit
 9: physmem: physical base address: 0x40000000
10: physmem: physmem total size:   3220504576 bytes (786256 pages)
11: physmem: Linux total size:     1073741824 bytes (262144 pages)
12: physmem: device physmem0 starts at 0x40000000 with 100663296
    bytes (96 MB)
13: physmem: device physmem0 uses 1 mem zone(s)
14: physmem: physmemMapZones [ device ]      [ zone ]        [
    physZone ]
15: physmem: physmemMapZones [ 0 ]           [ 0 ]           [ 0 ]
16: physmem: device physmem1 starts at 0x46000000 with 3119841280
    bytes (2975 MB)
17: physmem: device physmem1 uses 2 mem zone(s)
18: physmem: physmemMapZones [ device ]      [ zone ]        [
    physZone ]
19: physmem: physmemMapZones [ 1 ]           [ 0 ]           [ 0 ]
20: physmem: physmemMapZones [ 1 ]           [ 1 ]           [ 1 ]
21: physmem: physZone0: start 0x40000000, end 0xcff50000
22: physmem: physZone1: start 0x100000000, end 0x130000000
```

In the case of a 64 bit architecture, if the memory size is bigger then 4 GB, the BIOS can assign some memory area to devices like video memory just below of the 4 GB limit. This part of the memory can not be used by the ***physmem*** devices. In this way one of the ***physmem*** devices (generally ***/dev/physmem1***) is divided into two zones. Both zones have continuous physical addresses but there is a "gap" in the physical addresses between the two zones (even if the mapped user addresses are continuous between the two zones). In the previous Listing 15.3 we can see an example of the zones. The programs using ***physmem*** devices should be careful not to use the memory between the zones. The routines in the physmem library give assistance to this problem: a routine (`physmemBlockIsNotContinuous()`, see its description in Section 15.5) checks whether a given memory area contains unusable addresses.

During the loading process of the ***physmem*** kernel module the entire memory beyond the region of Linux is claimed as ***physmem*** memory. However, a specific size of the total memory region for ***physmem*** can be enforced by passing the parameter `physmemsize` to the ***physmem*** kernel module. This parameter is optional and specifies the total size of the memory region for ***physmem*** in MB. If this parameter is *0* or not present, the whole memory beyond the region of Linux will be claimed. As an example, the following commands load the ***physmem*** kernel module by putting a limit of 256 MB to the ***physmem*** memory:

```
> /sbin/rmmod physmem
> /sbin/insmod Linux/physmem.ko physmemsize=256
```

The size of the assigned memory for device***/dev/physmem0*** can be controlled by passing the parameter *physmemsize0* to the ***physmem*** kernel module. This parameter is optional and specifies the size in MB. If this parameter is 0 or not present, the default of 96 MB will be used. As an example, the following commands

load the ***physmem*** kernel module by putting a limit of 16 MB to the device ***/dev/physmem0***:

```
> /sbin/rmmod physmem
> /sbin/insmod Linux/physmem.ko physmemsize0=16
```

The size of the assigned memory for device***/dev/physmem1*** is given by the total size of the as ***physmem*** memory reduced by the size of the assigned memory for device***/dev/physmem0***, but not more than 2 GB, in case of 32 bit system. Both ***physmem*** module parameters can be passed to the ***physmem*** kernel module in one line. If one wants the driver to be loaded with some parameters at the next boot time, she/he can modify the ***/etc/rc.modules*** file accordingly.

Figure 15.1 in Section 15.4 shows the physical memory layout.

15.2.3 Testing the physmem driver

The DATE package ***physmem*** includes an utility program `physmemTest` to write and read back a pattern to the first and to the last 10000 bytes of the assigned memory for each ***physmem*** device. It can be used to test the functions of the ***physmem*** driver. Listing 15.4 shows how to start this utility program and a successful response. For the description of the program see Section 15.5. No other application should use the ***physmem*** memory during this test in order to avoid data corruption.

Listing 15.4 Example of testing the physmem driver with utility `physmemTest`

```
 1: > /date/physmem/Linux/physmemTest -m 0
 2: Opening /dev/physmem0
 3: Physical address = 0x40000000
 4: Physical usable size of device (no hole) : 0x6000000
 5:  + Number of memory zones : 1
 6:  + Mem zone 1 : 0x40000000 -> 0x46000000 (size = 0x6000000)
 7: Mmap done to 0x2accfcbda000 -> 0x2acd02bda000, trying to access
    physmem
 8:  + test write: from 0x2accfcbda000, 1000 bytes written
 9:  + test read:  from 0x2accfcbda000, all 1000 bytes are ok
10: Writing just before the memory end
11:  + test write: from 0x2acd02bd9c18, 1000 bytes written
12:  + test read:  from 0x2acd02bd9c18, all 1000 bytes are ok
13: unmapp'ing
14: physical full size of device (including hole) = 0x6000000
15: re-mapp'ing
16: Mmap done to 0x2accfcbda000 -> 0x2acd02bda000, trying to access
    physmem
17: Reading from the start of memory
18:  + test read:  from 0x2accfcbda000, all 1000 bytes are ok
19: Reading just before the memory end
20:  + test read:  from 0x2b2f31e25c18, all 1000 bytes are ok
21: >
22: > /date/physmem/Linux/physmemTest -m 1
23: Opening /dev/physmem1
24: Physical address = 0x46000000
25: Physical usable size of device (no hole) : 0xb9f50000
26:  + Number of memory zones : 2
27:  + Mem zone 1 : 0x46000000 -> 0xcff50000 (size = 0x89f50000)
28:  + Mem zone 2 : 0x100000000 -> 0x130000000 (size = 0x30000000)
29: Mmap done to 0x2b0e21b11000 -> 0x2b0edba61000, trying to access
    physmem
30:  + test write: from 0x2b0e21b11000, 1000 bytes written
31:  + test read:  from 0x2b0e21b11000, all 1000 bytes are ok
32: Writing just after the 'hole' in memory
33:  + test write: from 0x2b0eaba61000, 1000 bytes written
34:  + test read:  from 0x2b0eaba61000, all 1000 bytes are ok
35: unmapp'ing
36: physical full size of device (including hole) = 0xea000000
37: re-mapp'ing
38: Mmap done to 0x2b0e21b11000 -> 0x2b0f0bb11000, trying to access
    physmem
39: Reading from the start of memory
40:  + test read:  from 0x2b0e21b11000, all 1000 bytes are ok
41: phys_zones[1].start = 0x100000000
42: Reading just after the 'hole' in memory
43:  + test read:  from 0x2b0edbb11000, all 1000 bytes are ok
44: Writing just before the phys_zone[1].end
45:  + test write: from 0x2b0f0bb10c18, 1000 bytes written
46:  + test read:  from 0x2b0f0bb10c18, all 1000 bytes are ok
```

15.3 Utility programs for physmem

The DATE package ***physmem*** includes several utility programs for the ***physmem*** memory. They are useful for debugging purpose. The program `physmemTest` performs a simple read / write test on the first and last 10000 bytes of the assigned memory for each ***physmem*** device. With the help of the program `physmemFill` a pattern (also made of zeros for cleaning purposes) can be written into a section of the ***physmem*** memory. A section of the ***physmem*** memory can be displayed with the program `physmemDump`. In the following their synopsis is presented.

physmemTest

Synopsis `physmemTest [{-m|-M} DEVICE_NUMBER]`

Description The `physmemTest` program opens the device defined by the `DEVICE_NUMBER program parameter`, finds out the physical base address and the size of assigned memory, and maps to the whole assigned ***physmem*** memory using both mapping methods. For the details of mapping methods see Section 15.4, ”Internals of the physmem driver”.

If the assigned ***physmem*** memory has only one memory zone the program writes a pattern (incremental data words with starting value 0) to the first 10000 bytes, reads back and checks the first 10000 bytes, writes a pattern to the last 10000 bytes, reads back and checks the last 10000 bytes. Then it remaps the assigned memory using the second mapping method, reads back and checks the first and last 10000 bytes, and closes the corresponding device.

When the assigned ***physmem*** memory has two or more memory zones the program writes a pattern (incremental data words with starting value 0) to the first 10000 bytes, reads back and checks the first 10000 bytes, writes a pattern to the first 10000 bytes of the second zone, reads them back and checks them. Then it remaps the assigned memory using the second mapping method, reads back and checks the first 10000 bytes of the first and second zones. At the end it writes the pattern to the last 10000 bytes of the second zone, reads them back and checks them, and closes the corresponding device.

Parameters:

- `DEVICE_NUMBER`: this parameter defines the minor device number. The default value is 1 indicating device ***/dev/physmem1***.

Example see Listing 15.4.

Tests the ***/dev/physmem0*** and ***/dev/physmem1*** memory.

physmemFill, physmemFillWithAddress

Synopsis

```
physmemFill {{-o|-O} OFFSET | {-p|-P} PHYSICAL_ADDRESS}
            [{-l|-L} LENGTH
            [{-s|-S} START] [{-i|-I} INCREMENT]
            [{-m|-M} DEVICE_NUMBER]
            [{-d|-D} WORD_WIDTH]

physmemFillWithAddress [{-m|-M} DEVICE_NUMBER]
                       [{-d|-D} WORD_WIDTH]
```

Description The `physmemFill` program opens the chosen ***physmem*** device, finds out the physical base address and the size of assigned memory, maps to the whole assigned ***physmem*** memory, writes a pattern into a section of the memory according to the parameters, unmaps the assigned memory, and closes the corresponding device.

The physmemFillWithAddress program fills the words of whole chosen ***physmem*** device with its own physical address. It can be useful when one to check if a program really changed the contains of the ***physmem*** memory.

Parameters:

- OFFSET: this parameter defines the start of the section to be filled. It is given as offset in bytes (flag -o) or in words (flag -O) relative to the user base address of the assigned memory. Values starting with *0x* or *0X* are interpreted as hexadecimal values.
- PHYSICAL_ADDRESS: this parameter gives the physical address of the start of the section to be filled. Either OFFSET or PHYSICAL_ADDRESS must be specified.
- LENGTH: this parameter defines the length of the section to be filled. It is given in bytes (flag -l) or in words (flag -L). The size of the word is given by the WORD_WIDTH parameter. Values starting with *0x* or *0X* are interpreted as hexadecimal values.
- START: the section is filled with incremental data words. This parameter defines the starting value. The default value is *0*.
- INCREMENT: the section is filled with incremental data words. This parameter defines the increment value. The default value is *1*.
- DEVICE_NUMBER: this parameter defines the minor device number. The default value is 0 indicating device ***/dev/physmem0***.
- WORD_WIDTH: this parameter specifies whether the fill values are 4- or 8-byte long. This size is taken at the calculation of world offset (flag -L). For 32 bit architecture machine only 4-byte length is allowed. The default value is *4*.

Example

```
> /date/physmem/Linux/physmemFill -o 0 -l 1000 -m 1 -i 0
```

Fills the first 1000 bytes with *0* of the ***/dev/physmem1*** memory.

physmemDump

Synopsis

```
physmemDump {{-o|-O} OFFSET | {-p|-P} PHYSICAL_ADDRESS}
            [{-l|-L} LENGTH]
            [{-m|-M} DEVICE_NUMBER]
            [{-d|-D} WORD_WIDTH]
```

Description

The physmemDump program opens the chosen ***physmem*** device, finds out the physical base address and the size of assigned memory, maps to the whole assigned ***physmem*** memory, prints out the device properties (number of zones, size of zones and gaps, physical and user offsets, etc.), reads a section of the memory according to the parameters and prints it in a formatted way, unmaps the assigned memory, and closes the corresponding device.

If the chosen ***physmem*** device is divided into separate zones, the program skips the area between zones.

Parameters:

- OFFSET: this parameter defines the start address of the section to be read. It is

given as offset in bytes (flag -o) or in words (flag -O) relative to the user base address of the assigned memory. Values starting with *0x* or *0X* are interpreted as hexadecimal values.

- PHYSICAL_ADDRESS: this parameter gives the physical address of the start of the section to be read. Either OFFSET or PHYSICAL_ADDRESS must be specified.
- LENGTH: this parameter defines the length of the section to be read. It is given in bytes (flag -l) or in words (flag -L). The size of the word is given by the WORD_WIDTH parameter. Values starting with 0x or 0X are interpreted as hexadecimal values. If a 0 length is given the program only prints the device properties as the number of zones, the size of zones and gaps, the physical and user offsets.
- DEVICE_NUMBER: this parameter defines the minor device number. The default value is 0 indicating device ***/dev/physmem0***.
- WORD_WIDTH: this parameter specifies whether to dump the memory words 4- or 8-byte long. This size is taken at the calculation of world offset (flag -L). For 32 bit architecture machine only 4-byte length is allowed. The default value is *4*.

Example

```
> /date/physmem/Linux/physmemDump -o 0 -l 1000 -m 1
```

Dumps the first 1000 bytes of the ***/dev/physmem1*** memory.

15.4 Internals of the physmem driver

The ***physmem*** kernel module services the module initialization as well as the cleanup requests and the file operations on the device files with major number 122. The source code of this kernel module can be found in file ***physmem.c*** together with ***physmem.h*** in the ***physmem*** package. During the loading of the ***physmem*** kernel module, the device is registered, the memory end of the Linux region is determined, and the size of the memory beyond this boundary is obtained by gradually mapping 4 KB pages until a write/read test fails on them. Eventually, the ***physmem*** memory is assigned to the devices ***/dev/physmem0*** and ***/dev/physmem1***, (see Figure 15.1). During the unloading of the ***physmem*** kernel module, the device is unregistered. The implemented file operations are `open()`, `close()`, `mmap()`, `munmap()`, and `ioctl()`. These are standard Linux system routines whose synopsis with reference to ***physmem*** kernel module are presented below. The usage of these routines is illustrated by file ***physmemTest.c*** in the ***physmem*** package.

In the case of 64 bit machine architecture the ***physmem*** memory can be divided into separate area, so called memory zones. The reason of this is that the BIOS assigns some memory area to devices like video memory just below of the 4 GB limit. This part of the memory can not be used by the *physmem* devices. In this way one of the ***physmem*** devices (generally ***/dev/physmem1***) can be divided into two ***zones***. Both zones have continuous physical addresses but the “gap” addresses between the zones can not be used.

The file operation `mmap()` can map the ***physmem*** devices in two different ways; **`usable`** or **`full`** mapping. In the first case the user addresses are continuous for the usable part of the device. This means that the last user address of a memory zone is followed by the first user address of the next zone. There is no gap in the user's space. To calculate the physical addresses of a given user address one has to take into account the physical gap between the zones.

In the case of **`full`** mapping, the user address space is continuous, i.e. the gap area contains user address as well. This facilitates the calculation of physical address of a given user address. On the other hand the user can accidentally read or write from and to the forbidden gap area, which can lead to system crash.

The two mapping methods can be selected by an `ioctl()` call before calling `mmap()` operation.

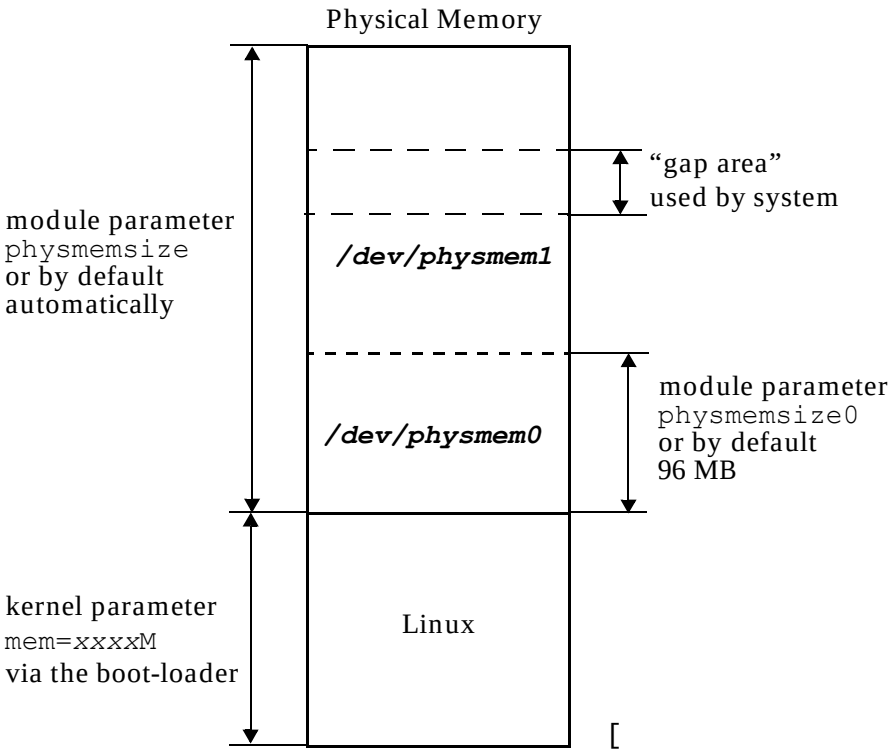

Figure 15.1 Memory layout of physmem

open

C Synopsis

```
#include <unistd.h>
#include <fcntl.h>

int open( const char *pathname, int flags )
```

Description Converts the parameter *pathname*, which has to be the absolute path of the ***physmem*** device file, into a file descriptor to handle subsequent I/O operations. Several processes can open this device file. The kernel module counter is always 0.

Parameters

- `pathname`: pointer to the ***physmem*** device file name
- `flags`***:*** this parameter must include one of the access mode ***O_RDONLY***, ***O_WRONLY*** or ***O_RDWR***. These request opening the ***physmem*** area read- only, write-only, or read / write, respectively.

Returns A new file descriptor (non-negative integer), or -1 if an error occurred.

close

C Synopsis

```
#include <unistd.h>

int close( int fd )
```

Description Closes the file descriptor *fd* that was created by function `open()`. The kernel module counter is always 0.

Parameter `fd`: the file descriptor of the ***physmem*** device.

Returns Zero on success, or -1 if an error occurred.

mmap

C Synopsis

```
#include <unistd.h>
#include <sys/mman.h>

void *mmap( void *start, size_t length, int prot,
            int flags, int fd, off_t offset )
```

Description Maps to ***physmem*** memory for the address range that is specified by parameter `offset` and parameter `length` in bytes. The memory region is mapped in multiples of the page size, 4096 bytes. Several processes are allowed to create a mapping. No initialization of the specified memory region is done. Before calling `mmap()` it is mandatory to call function `ioctl()` with parameter value ***PHYSMEM_GETSIZE*** or ***PHYSMEM_GETFULLSIZE*** (see below the description of `ioctl`). The `ioctl()` call informs the driver of the mapping mode: **`usable`** or **`full`** mapping (see the introductory part of Section 15.4)

Parameters

- `start`: defines a preferred value for the pointer to be returned, but it is usually left to 0.It must be a multiple of the page size.
- `length`***:*** size of the mapped memory area in bytes.
- `prot`: this argument describes the desired memory protection (and must not conflict with the open mode of the device) is either **`PROT_NONE`** or the bitwise OR of one or more of the other **`PROT_*`** flags:
  PROT_EXEC : Pages may be executed.
  PROT_READ : Pages may be read.
  PROT_WRITE : Pages may be written.

PROT_NONE : Pages may not be accessed.

- `flags:` this parameter specifies the type of the mapped object (bits ***MAP_FIXED***, ***MAP_SHARED***, or ***MAP_PRIVATE***, whereas the latter two bits are exclusive).
- `fd:` this parameter is the file descriptor that was produced by the function `open()`
- `offset:` start address in bytes of the mapped physmem memory range. It should be a multiple of the page size.

Returns A pointer to the mapped area, or -1 if an error occurred. The ***physmem*** memory can be accessed with this returned pointer.

munmap

C Synopsis

```
#include <unistd.h>
#include <sys/mman.h>

int munmap( void *start, size_t length )
```

Description Releases the mapping to ***physmem*** memory for the address range that is specified by the parameter `start` and the parameter `length` in bytes. The region is also automatically unmapped when the process terminates. On the other hand, closing the file descriptor does not unmap the region.

Parameters

- `start`: start address of the mapped physmem area, which is the pointer returned by the function `mmap()`.
- `length:` size of the mapped memory area in bytes.

Returns Zero on success, or -1 if an error occurred.

ioctl

C Synopsis

```
#include <sys/ioctl.h>
#include "physmem.h"

int ioctl( int fd, int request, void *argument )
```

Description The sizes and the physical base addresses of the ***physmem*** memory and the usable memory zones can be obtained with this device-specific function. It is mandatory to call this function with parameter `request` set to ***PHYSMEM_GETSIZE*** or ***PHYSMEM_GETFULLSIZE***, before calling `mmap()`. The call not only returns the requested value, but also informs the driver of the mapping mode: **`usable`** or **`full`** mapping (seeSection 15.4).

Parameter

- `fd`: the file descriptor that was created by function `open()`.

- `request`: this parameter defines the out parameter `argument` to be returned. It can be set to the values: ***PHYSMEM_GETADDR, PHYSMEM_GETSIZE, PHYSMEM_GETFULLSIZE, PHYSMEM_GETNUMMEMZONES*** and ***PHYSMEM_GETMEMZONES***.
- `argument`: pointer to the out parameter. Its type, size and value are encoded in the `request` parameter. According to the `request`, the returned values are:

- ***PHYSMEM_GETADDR***: the parameter `argument` returns the physical address of the beginning of the physmem device. Its type is ***unsigned long***.

- ***PHYSMEM_GETSIZE***: the parameter `argument` returns the usable size of the physmem device where the (possible) holes are not taken into account. Its type is ***unsigned long***.

- ***PHYSMEM_GETFULLSIZE***: the parameter `argument` returns the total size of the physmem device, where the (possible) holes are taken into account, even if the memory located there cannot be used. Its type is ***unsigned long***.

- ***PHYSMEM_GETNUMMEMZONES***: the parameter `argument` returns the number of memory zones used by the device. Its type is ***unsigned int***.

- ***PHYSMEM_GETMEMZONES***: to be called with an allocated array of "number of memory zones used by the device" elements of type ***struct memZoneStruct***. The call will populate the given array with the physical start addresses, physical end addresses and sizes of the different used memory zones.

Returns Zero on success, or -1 if an error occurred.

15.5 Physmem application library

In the following we present the synopsis of the C routines and functions useful to build programs using physmem area. The ***physmem_lib.c***, ***physmem_lib.h*** and ***physmem_lib.o*** files are part of the ***physmem*** package.

Before calling any of the following routines one has to include a header file:

```
#include "physmem_lib.h"
```

This file contains the type definition of the structures referred in the routines and the routine's prototypes. It also has reference to ***physmem.h*** header file, which contains the definitions special to physmem driver.

The physmem memory area can be divided into memory ***zones*** as described in Section 15.4. Only the memory parts inside the zones can be used. A zone is characterized by its starting and ending addresses. The start address is the address of the first byte of the zone, the end address is the address of the byte following the last address of the zone. Several structures are defined in the header files to contain the physical and the user addresses and offsets of the zones. (the offsets are the differences of a given address and the start address of the ***physmem*** area.)

All of the following routines uses the ***physmem handler***. It is defined by the open routine and contains the necessary informations for the correct executions of the other routines.

physmemOpen

Synopsis

```
#include physmem_lib.h

int physmemOpen(physmemHandler_t  *device,
                int                minor,
                int                full_size)
```

Description The routine opens the device ***/dev/physmem<minor>*** and maps the area according the value of `full_size`. It also gets the size, the user and physical address of ***physmem*** area, the coordinates (start and end addresses, and size) of zones and gaps, and fills this information into the handler structure pointed by `device`.

Parameters:

- `device`: this parameter points to device handler to be filled by the routine in case of successful open.
- `minor`: this parameter defines the minor number of the physmem device. Its value could be `0` or `1`.
- `full_size`: this parameter defines the mapping method of the area.

  A value of `0` means:

  `usable` mapping: the user addresses are continuous for the whole device. This means that last user address of a memory zone is followed by the first user address of the next zone. There is no gap in the user's space.

  A value of `1` means:

  `full` mapping: the user address space is continuous, i.e. the gap area has user addresses as well.

Return value 0 if the open and mapping was successful, -1 in any other case

physmemClose

Synopsis

```
#include physmem_lib.h

int physmemClose(physmemHandler_t  device)
```

Description The routine removes the memory mapping of the ***PHYSMEM*** area referred by `handler device` and closes area.

Parameters:

- `device`: the handler of the ***physmem*** device defined by a previous call of `physmemOpen()` routine.

Return value 0 if the unmapping and close were successful, -1 in any other case.

physmemPrintZones

Synopsis

```
#include physmem_lib.h

void physmemPrintZones(physmemHandler_t device)
```

Description The routine displays the start- and end-offsets of all the memory zones of the ***physmem*** area referred by the handler `device`. The offsets are the differences of the address of the given zone and the address of the start of the physmem area.

Parameters:

- `device`: the handler of the ***physmem*** device defined by a previous call of `physmemOpen()` routine.

physmemBlockIsNotContinuous

Synopsis

```
#include physmem_lib.h

int physmemBlockIsNotContinuous(
      volatile unsigned int        *block_start,
      volatile unsigned int        *block_end,
      physmemHandler_t             device,
      int                          *start_status,
      int                          *end_status)
```

Description The routine investigates if a memory block's start and end are in the same ***physmem*** memory zone or not. It returns the zone's (or gap's) number of the start and end of block. The memory zones are numbered staring from 0, while the gaps are numbered by negative numbers starting from -1.

An example of the memory zone and gap numbering:

```
gap -1 [--zone 0--] gap -2 [--zone 1--] gap -3 [--zone 2--]
```

i.e. the memory zone N is surrounded by gaps (N-2) and (N-3).

Parameters:

- `block_start`: user address of the block start.
- `block_end`: user address of the block end (last byte of the block +1).
- `device`: this parameter points to device handler to be filled by the routine in case of successful open.
- `start_status`: returns the number of memory zone (>= 0) or number of gap (<0) where the block starts
- `end_status`: returns the number of memory zone (>= 0) or number of gap (<0) where the block ends

Return value 0 if the start and end of the block is in the same zone, -1 in any other case.

physmemPhysOffset

Synopsis

```
#include physmem_lib.h

unsigned long physmemPhysOffset(unsigned long user_offset,
                                physmemHandler_t   device)
```

Description This utility routine calculates the physical memory offset from the user offset.

Parameters:

- `user_offset`: user memory offset value.
- `device`: this parameter points to device handler to be filled by the routine in case of successful open.

Return value the physical offset, or -1 if no physical address in the ***physmem*** area belongs to the given user offset. It happens if the user address belonging to the user offset is outside of the ***physmem*** memory zones.

physmemUserOffset

Synopsis

```
#include physmem_lib.h

unsigned long physmemUserOffset(unsigned long phys_address,
                                physmemHandler_t   device)
```

Description This utility routine calculates the user memory offset from the physical address.

Parameters:

- `user_address`: physical memory address value.
- `device`: this parameter points to device handler to be filled by the routine in case of successful open.

Return value the user offset, `or -1` if no user offset in the physmem area belongs to the given physical address. It happens if the physical address is outside of the ***physmem*** memory zones.

16 Utility packages

This chapter describes the DATE utility packages ***banksManager***, ***bufferManager***, ***simpleFifo*** and ***recordingLib***, their architecture, the API and the associated procedures and utilities. These packages are included in the standard DATE kit and are mainly used internally by DATE.

16.1 The banks manager package

16.1.1 Introduction

All DATE actors need some support for memory banks. These are handled by the DATE ***banksManager*** package which provides a common, configurable and flexible interface that includes features such as dynamic sizing and automatic initialization.

16.1.2 Architecture

The base of the ***banksManager*** package is the banks database as defined by the DATE ***database*** package. In the database are listed, role by role, all the banks that needed to run together with their characteristics (size, type, support). At each start of run, the ***rcServer*** daemon creates (if needed) and initializes the banks required by the role according to the runtime configuration of the DATE system.

Banks are mapped on each actor running on the given machine in strict order. A unique ***ID*** is given to each bank. Different actors may see the same banks mapped at different virtual addresses: for this reason, exchange of absolute pointers to entities contained in the banks is forbidden. Inter-actors exchanges should always be done using the offset between the beginning of the bank and the pointer plus the identifier of the bank itself. These two entities are guaranteed to map to the same object from any process using the ***banksManager*** package. A set of macros and symbols are given to facilitate the procedure.

Code using the ***banksManager*** package should be compiled in a DATE environment, using the DATE symbols and via the DATE makefiles rules. The include file ***${DATE_BANKS_MANAGER_DIR}/banksManager.h*** should be included and the library ***${DATE_BANKS_MANAGER_BIN}/libBanksManager.a*** should be used for linking. The library makes intensive use of the central definition file ***${DATE_COMMON_DEFS}/event.h***. It is included automatically if the DATE compilation rules are used.

16.1.3 Entries and symbols

dateInitControlRegion

C Synopsis

```
#include "banksManager.h"

int DATE_INIT_CONTROL_REGION( hostRole )
int dateInitControlRegion( hostRole, eventId, rcShmId )
```

Description Initialize the control region. This entry is used only by ***rcServer***. The `hostRole` parameter is an ID for the role assumed by the machine. `eventId` and `rcShmId` are the unique IDs for the DATE event and run control block as defined by the common DATE definitions: they are used to validate the structures of the various modules wishing to map to the control buffer. The two IDs are written in the

control structure and are verified when any attempt is made to connect to it. On mismatch (code compiled on different stages or using incompatible DATE distribution kits) the whole operation is refused. This entry does not allocate memory banks other than the one needed for the run control block. The macro ***DATE_INIT_CONTROL_REGION*** can be used to call `dateInitControlRegion` with the appropriate parameters.

Returns ***TRUE*** for success, ***FALSE*** on error.

dateMapBanks

C Synopsis

```
#include "banksManager.h"

int DATE_MAP_BANKS( hostRole )
int DATE_MAP_AND_INIT_BANKS( hostRole )
int dateMapBanks( hostRole, eventId, rcShmId, initTheBanks )
```

Description Map all banks needed for the given role. If `initTheBanks` is ***TRUE***, the banks are also initialized (the ***rcServer*** is the process that does this automatically at each start of run). `eventId` and `rcShmId` are symbols defined by the DATE central include file and are used to match the IDs of the event control structure and of the run control structure as defined during the compilation phase. This ensures that all actors accessing the banks share the same data structures. Mapping can fail if these two IDs do not match the value found in the control block (as defined by `dateInitControlRegion`).

The macro `DATE_MAP_BANKS` can be used to call `dateMapBanks` with the appropriate IDs. Similarly, `DATE_MAP_AND_INIT_BANKS` can be used to do the same and to request to initialize the banks.

Returns ***TRUE*** for success, ***FALSE*** on error.

BO2V/BV2O

C Synopsis

```
#include "banksManager.h"

void *BO2V( int bankId, int offset )
int BV2O( int bankId, void *address )
```

Description Macros used to manipulate virtual addresses and their offsets. `BO2V` takes a bank ID plus an offset and returns the corresponding virtual address. `BV2O` takes a bank ID plus a virtual address to return an offset.

rcShm/readoutReady/readoutFirstLevel/readoutSecondLevel/readoutData/edmReady/eventBuilderReady/eventBuilderData/hltReady/hltDataPages

C Synopsis

```
#include "banksManager.h"

rcShm
   rcShmO
   rcShmSize
   rcShmBank
readoutReady
   readoutReadyO
   readoutReadySize
   readoutReadyBank
readoutFirstLevel
   readoutFirstLevelO
   readoutFirstLevelSize
   readoutFirstLevelBank
readoutSecondLevel
   readoutSecondLevelO
   readoutSecondLevelSize
   readoutSecondLevelBank
readoutData
   readoutDataO
   readoutDataSize
   readoutDataBank
edmReady
   edmReadyO
   edmReadySize
   edmReadyBank
eventBuilderReady
   eventBuilderReadyO
   eventBuilderReadySize
   eventBuilderReadyBank
eventBuilderData
   eventBuilderDataO
   eventBuilderDataSize
   eventBuilderDataBank
hltReady
   hltReadyO
   hltReadySize
   hltReadyBank
hltDataPages
   hltDataPagesO
   hltDataPagesSize
   hltDataPagesBank
```

Description Pointers, offsets, sizes and banks for the entities handled by the ***banksManager*** package. For the definition of the banks, please refer to the DATE ***database*** package. If the entity is not present, the pointer will be ***NULL***, offset and bank will be set to -1, and the size will be set to 0.

edmInput/recorderInput

C Synopsis

```
#include "banksManager.h"

void *edmInput
void *recorderInput
```

Description FIFOs used as input to the corresponding actor. Their actual value depends on the runtime configuration of DATE.

physmemAddr/physmemBank

C Synopsis

```
#include "banksManager.h"

void *physmemAddr
int physmemBank
int physmemNumZones
struct memZonesStruct *physmemZones
```

Description Physical address, bank ID, number of zones and descriptions of the memory zones used for the ***PHYSMEM*** driver. Respectively ***NULL***, -1, 0 and ***NULL*** if not loaded.

When `physmemNumZones` is zero or one, there are no gaps in the address range of the ***PHYSMEM*** zone. When `physmemNumZones` is > 1 s (e.g. when the system has more than 4 GB of RAM and where the starting physical address of the ***PHYSMEM*** memory block is below the 4 GB memory limit) then special care must be taken to avoid the gaps in the address range of ***PHYSMEM***.: in this case, the description of the memory zones within ***PHYSMEM*** is given in the array `physmemZones` (see Section 15.4).

16.1.4 Internals

The ***banksManager*** package is self-contained in the ***${DATE_BANKS_MANAGER_DIR}*** directory. This directory contains the buffer manager module, a utility to dump the runtime configuration and the makefile for the package. No special setup is required. The ***banksManager*** package needs the ***database***, ***runControl***, ***physmem***, ***bufferManager*** and ***simpleFifo*** packages to compile and link.

When running in environments with multiple ***PHYSMEM*** zones (e.g. when the system has more than 4 GB of RAM and where the starting physical address of the ***PHYSMEM*** memory block is below the 4 GB memory limit) special care must be taken to handle the memory hole created by the BIOS between the 3.3 GB and the 4 GB marks. In these cases, the variable `physmemNumZones` contains a value > 1 and the array physmemZones contains the description of each zone.

Compilation of the package is straightforward and requires no additional setup. The compilation symbol ***XTRA_CHECKS*** can be used to run more intensive run-time checks on the parameters and on the buffer itself. This symbol should not be defined for production.

The `dumpBanks` utility (see Section 4.3.6) can be used to inspect the runtime configurations of the DATE banks.

16.2 The bufferManager package

16.2.1 Introduction

The DATE ***bufferManager*** package provides the support for allocation and deallocation of memory coming from a common buffer via a lightweight protocol. The DATE architecture needs an efficient single-producer, multiple-consumers buffer manager. The DATE ***bufferManager*** package fulfills these requirements.

The basic element of a ***bufferManager*** entity is a shared memory block to support the whole mechanism. No other resources (from DATE or from the Operating System) are needed. Inter-process synchronization is made via this shared memory block and it is based on linear test-and-set procedures. The package itself is never spin locking: this is left - if needed - at the caller's level.

16.2.2 Architecture

The ***bufferManager*** package uses the resources provided by the caller level, namely a memory block to be managed. This block must be shared between all actors who need access to it. A buffer can have only one producer (allocating memory blocks) and multiple consumers (deallocating memory blocks). A producer process can also act as a consumer process, although this feature is not part of the user requirement and may therefore be dropped in future releases of the package. External packages (e.g. the DATE ***simpleFifo*** package) must be used to transfer pointers to the memory blocks allocated by the ***bufferManager*** package between processes.

A memory block allocated by this package can be used as conventional dynamic memory (same features as for memory allocated via the `malloc` system call). Any kind of data can be stored in these memory blocks. The only constraint is on memory pointers. As the Operating System may map on multiple processes the same memory block at different virtual addresses, special care must be taken to avoid sharing of virtual pointers. If the exchange of pointers is required, memory offsets (difference between the virtual address to exchange and the start virtual address of the memory block) must be specified together with a protocol to indicate the proper start virtual address. Here DATE assumes that all shared memory blocks are allocated via the ***banksManager*** package and a set of standard calls and macros are available to facilitate the task. If this is not the case, it is up to the caller to establish an alternate protocol to exchange virtual pointers.

When standard DATE buffers are used as support for the ***bufferManager*** package, the ***dateBanks*** package will automatically initialize them at run time as soon as they are created. If the ***bufferManager*** package has to work on non-standard DATE buffers, special care must be taken to initialize the buffer in the appropriate sequence and to avoid out-of-sequence accesses to the same object, an action that may give unpredictable results.

With the exception of `bmInit`, all entries where a buffer pointer is given as a parameter assume that this pointer has been used for a previous, successful call to `bmInit`. Little or no run-time checks are made on the validity of this and other parameters.

The routine bmInit handles memory blocks allocated in ***PHYSMEM*** in such a way to avoid gaps in the address range (if any). The procedure used is to initialise the buffer and immediately allocate a memory block that spans across the memory gap(s) eventually present in the memory block. DATE actors will therefore never receive a memory block that includes any location belonging to these gap(s).

The library implemented by the ***bufferManager*** package provides three sets of entries: one set for the producer process (who allocates blocks), one set for the consumer processes (who deallocate blocks) and one set common between all classes of processes. As the producer may also act as a consumer process, all calls relative to consumers are also available to the producer. Special care must be taken to use the set of entries appropriate to the role of each process.

Due to the way compilers work on several architectures, it is recommended to keep all sizes (memory blocks, memory buffers) aligned to 32 bit. Failure to do so may result in raising of several signals (***SIGSEGV***, ***SIGBUS***) in the calling process.

Code using the ***bufferManager*** package should be compiled in a DATE environment, using the DATE symbols and via the DATE makefiles rules. The include file ***${DATE_BM_DIR}/dateBufferManager.h*** should be included and the library ***${DATE_BM_BIN}/libDateBufferManager.a*** should be used for linking.

16.2.3 Common entries

bmGetVersionId

C Synopsis

```
#include "dateBufferManager.h"

char *bmGetVersionId( void )
```

Description Inquiry for the version ID of the package.

Returns Pointer to static string.

bmGetError

C Synopsis

```
#include "dateBufferManager.h"

char *bmGetError( void )
```

Description Inquiry for the string describing the last error occurred in the library. This string is in common between all buffers handled by the package and it is overwritten at each call made to the library.

Returns Pointer to a static string.

16.2.4 Producer entries

bmInit

C Synopsis

```
#include "dateBufferManager.h"

int bmInit( void *buffer, int sizeOfBuffer )
```

Description The memory block pointed to by `buffer` and of size `sizeOfBuffer` is initialized to be used as support for a buffer entity. The buffer pointer should contain the address of a memory block of at least `sizeOfBuffer` bytes. This entry should not be called when the same buffer is already in use as a memory buffer by other processes. For standard DATE buffers, this entry is called automatically when the buffer itself is created by the ***dateBanks*** package. No checks are made on the validity of the pointer or on the availability of the memory block.

Returns ***TRUE*** on successful completion, ***FALSE*** on error.

bmValidate

C Synopsis

```
#include "dateBufferManager.h"

int bmValidate( void *buffer )
```

Description Check the structure of a buffer for correctness. This entry should be used to guarantee (up to a minimal level) the good status of the buffer in case memory corruption(s) are suspected. The check uses system resources and should therefore be avoided in environments where efficiency is an issue.

Returns ***TRUE*** if the buffer looks OK, ***FALSE*** otherwise.

bmAllocate

C Synopsis

```
#include "dateBufferManager.h"

void *bmAllocate( void *buffer, int sizeOfBlock )
```

Description A block of at least `sizeOfBlock` bytes is allocated (if possible) from the given buffer. The library may fail to allocate for a given size - even if the buffer itself has enough space - whenever the data set is split into smaller subsets.

Returns -1 for fatal error, ***NULL*** if there is not enough free space in the buffer. Otherwise a pointer to the allocated memory block.

bmResize

C Synopsis

```
#include "dateBufferManager.h"

int bmResize( void *block, int newSize )
```

Description The block pointed to by `block` is - if possible - resized to `newSize` bytes. The block must have been previously allocated to contain at least `newSize` bytes (see the entry `bmAllocate`).

Returns ***TRUE*** for success, ***FALSE*** on error.

bmDefragment

C Synopsis

```
#include "dateBufferManager.h"

int bmDefragment( void *buffer )
```

Description The buffer pointed by `buffer` is thoroughly defragmented. This procedure can be used during quiescent phases - where the buffer is not in use - to repack distributed blocks of memory. The process is linear, does not move memory around the buffer and does not affect blocks eventually allocated.

Returns ***TRUE*** for success, ***FALSE*** on error.

bmGetBlocksInUse

C Synopsis

```
#include "dateBufferManager.h"

int bmGetBlocksInUse( void *buffer )
```

Description Count the number of blocks currently allocated in the given buffer.

Returns Number of allocated blocks on success, -1 on error.

bmGetTotalSpace

C Synopsis

```
#include "dateBufferManager.h"

int bmGetTotalSpace( void *buffer )
```

Description Count the maximum number of bytes available in the given buffer.

Returns Maximum number of available bytes on success, -1 on error.

bmGetAvailableSpace

C Synopsis

```
#include "dateBufferManager.h"

int bmGetAvailableSpace( void *buffer )
```

Description Count the number of bytes currently available in the given buffer. Due to possible fragmentation of the buffer, this may not be the maximum size that can be allocated via the `bmAllocate` call but only an upper bound.

Returns Number of available bytes on success, -1 on error.

bmGetNumAllocations

C Synopsis

```
#include "dateBufferManager.h"

int bmGetNumAllocations( void *buffer )
```

Description Count the number of allocation requests made to the buffer since the last call to `bmInit`.

Returns Number of allocations on success, -1 on error.

bmGetNumFulls

C Synopsis

```
#include "dateBufferManager.h"

int bmGetNumFulls( void *buffer )
```

Description Count the number of allocation requests rejected due to lack of space made to the buffer since the last call to `bmInit`.

Returns Number of failed allocations on success, -1 on error.

16.2.5 Consumer entries

bmDeallocate

C Synopsis

```
#include "dateBufferManager.h"

int bmDeallocate( void *block )
```

Description The block pointed by `block` is deallocated. The package assumes that the block has been previously allocated via a `bmAllocate` call.

Returns ***TRUE*** on success, ***FALSE*** on error.

16.2.6 Internals

The package is self-contained in the ***${DATE_BM_DIR}*** directory. This folder contains the buffer manager module, a validation program and the makefile for the package. No special setup is required.

Compilation of the package is straightforward and requires no additional setup. The compilation symbol ***XTRA_CHECKS*** can be used to run more intensive run-time checks on the parameters and on the buffer itself. This symbol should not be defined for production.

The ***bufferManager*** package needs the ***database***, ***runControl***, ***physmem***, ***banksManager*** and ***simpleFifo*** packages to compile and link.

The `dateBufferManagerValidate` program can be used to validate the ***dateBufferManager*** library. It should be run without parameters to execute an extensive suite of tests and exit with an appropriate status message. See Listing 16.1 for an example run.

Listing 16.1 Example of dateBufferManagerValidate run

```
1:  > DATE buffer manager validator starting
2:  dateBufferManager.c: single-producer multiple-consumer buffer handler
    V 1.2 compiled Sep 20 2002 17:15:38
3:  DATE buffer manager validator completed
```

16.3 The simpleFifo package

16.3.1 Introduction

Multi-process systems need inter-process synchronization tools. The DATE package requires a fast, lightweight exchange of data between process pairs (one data producer and one data consumer). The ***simpleFifo*** package fulfills this requirement.

The basic element of a ***simpleFifo*** is a memory block to support the whole mechanism. No other resources (from DATE or from the Operating System) are needed. Inter-process synchronization is made via this shared memory block and it is based on linear test-and-set procedures. The package itself is never spin locking: this is left - if needed - at the caller's level.

16.3.2 Architecture

The ***simpleFifo*** package implements a ***simpleFifo*** entity using a shared memory block provided by the caller. This block is partitioned into a control block and a data block. The ***simpleFifo*** entity can then be used to exchange blocks of arbitrary size in a first-in first-out fashion.

A ***simpleFifo*** allows exactly one data producer and one data consumer. It provides a set of calls for the data producer, a set for the consumer and a third set common for producer and consumer.

With the exception of `fifoDeclare`, all entries where a buffer pointer is given as a parameter assume that this pointer has been used for a previous, successful call to `fifoDeclare`. Little or no run-time checks are made on the validity of this parameter.

Due to the way compilers work on several architectures, it is recommended to keep all sizes (memory block, FIFO head, FIFO tail) aligned to 32 bit. Failure to do so may result in raising of several signals (***SIGSEGV***, ***SIGBUS***) in the calling process.

Code using the ***simpleFifo*** package should be compiled in a DATE environment, using the DATE symbols and via the DATE makefiles rules. The include file ***${DATE_SIMPLEFIFO_DIR}/simpleFifo.h*** should be included and the library ***${DATE_SIMPLEFIFO_BIN}/libFifo.a*** should be used for linking.

16.3.3 Common entries

---

fifoGetVersionId

C Synopsis

```
#include "simpleFifo.h"

char *fifoGetVersionId( void )
```

Description Inquiry for the version ID of the package.

Returns Pointer to a static string.

---

fifoDeclare

C Synopsis

```
#include "simpleFifo.h"

int fifoDeclare( void *buffer, int sizeOfBuffer )
```

Description The memory block pointed to by `buffer` and of `sizeOfBuffer` size is initialized to be used as support for a ***simpleFifo*** entity. The buffer pointer should contain the address of a memory block of at least `sizeOfBuffer` bytes. This entry should not be called when the same buffer is already in use as a ***simpleFifo*** by other

processes. No checks are made on the validity of the pointer or on the availability of the memory block.

Returns 0: successful completion, -1 on error.

fifoGetSize

C Synopsis

```
#include "simpleFifo.h"

int fifoGetSize( void *buffer )
```

Description Inquiry for the size of the data partition of the ***simpleFifo*** pointed by `buffer`.

Returns Size in bytes of the data block of the FIFO.

fifoCheck

C Synopsis

```
#include "simpleFifo.h"

int fifoCheck( void *buffer )
```

Description Check the structure of a ***simpleFifo*** for correctness. This entry should be used to guarantee (up to a minimal level) the good status of the ***simpleFifo*** in case memory corruption(s) are suspected. The check uses system resources and should therefore be avoided in environments where efficiency is an issue.

Returns ***TRUE*** if the FIFO is OK, ***FALSE*** otherwise.

fifoGetOccupancy

C Synopsis

```
#include "simpleFifo.h"

int fifoGetOccupancy( void *buffer )
```

Description Get the occupancy of the given FIFO.

Returns Occupancy in percentage (100: completely full, 0: completely empty).

fifoIsEmpty

C Synopsis

```
#include "simpleFifo.h"

int fifoIsEmpty( void *buffer )
```

Description Test the given FIFO for presence of data.

Returns ***TRUE*** if the FIFO is empty, ***FALSE*** otherwise.

16.3.4 Producer entries

---

fifoGetFree

C Synopsis

```
#include "simpleFifo.h"

void *fifoGetFree( void *buffer, int neededSize )
```

Description Get a pointer to a data block for the given size from the head of the FIFO. This block can be used for whatever purposes the caller may need, as long as its maximum size is respected. When done, the producer should validate the block in order to make it available to the consumer.

Returns ***NULL*** if there is no available datablock for the requested size, -1 if the request is not valid for the FIFO, otherwise pointer to a memory block. If the entry returns ***NULL***, it is up to the caller to take appropriate action (retry, sleep and retry, etc.). It is possible for this entry to return ***NULL*** followed by -1.

---

fifoValidate

C Synopsis

```
#include "simpleFifo.h"

void fifoValidate( void *buffer, int actualSize )
```

Description The head of the FIFO previously allocated with `fifoGetFree` is made available to the consumer. This block can be resized to the given actualSize that must be at most as big as the allocated size during `fifoGetFree`. The procedure must take care that all accesses to this location may - from now on - conflict with other processes - including itself.

16.3.5 Consumer entries

---

fifoHasData

C Synopsis

```
#include "simpleFifo.h"

int fifoHasData( void *buffer )
```

Description Poll the FIFO for data.

Returns ***TRUE*** if data is available, ***FALSE*** otherwise.

fifoGetFirst

C Synopsis

```
#include "simpleFifo.h"

void *fifoGetFirst( void *buffer )
```

Description Get the pointer to the tail of the FIFO. This pointer points to a memory block of arbitrary size. The actual size of the element head of the FIFO must be worked out by the caller.

Returns Pointer to the tail of the FIFO, ***NULL*** if the FIFO is empty.

fifoSetFree

C Synopsis

```
#include "simpleFifo.h"

void fifoSetFree( void *buffer, int size )
```

Description The tail of the FIFO is made available to the data producer. The block is assumed to have the given size, as specified by the caller.

16.3.6 Internals

The package is self-contained in the ***${DATE_SIMPLEFIFO_DIR}*** directory. This folder contains the FIFO handler module, a validation package, a simple performance measurement program and the makefile for the package. No special setup is required, the package is entirely self-contained and does not require other packages to compile and link.

Compilation of the package is straightforward and requires no additional setup. The compilation symbol ***DEBUG*** can be used to produce some output on ***stdout*** in case of error. This symbol should not be used for production.

The ***simpleFifoValidate*** program can be used to validate the ***simpleFifo*** library. It should be run on the same machine in two copies, one producer and one consumer. The procedure will run the given number of loops with basic ***simpleFifo*** operations (including some runtime checks) and exit with an appropriate status message. See Listing 16.2 for an example run.

Listing 16.2 Example of ***simpleFifoValidate*** run

```
 1:  > ${DATE_SIMPLEFIFO_BIN}/simpleFifoValidate p 100&
 2:  [1] 9606
 3:  FIFO V 1.03 validation producer side starting
 4:  > ${DATE_SIMPLEFIFO_BIN}/simpleFifoValidate c 100
 5:  FIFO V 1.03 validation consumer side starting
 6:  Producer: consumer started
 7:  Consumer: producer started
 8:  Producer: test completed OK. Sleeps: 2050836.000000. Checks: 3.
 9:  Consumer: test completed OK. Sleeps: 2459685.000000. Checks: 2.
10:  [1]+ Done ${DATE_SIMPLEFIFO_BIN}/simpleFifoValidate p 100
```

16.4 The recording library package

16.4.1 Introduction

Actors running in a DATE environment may need to record raw events on local or remote systems. DATE provides two standard recording libraries. They both allow efficient and tailored output of raw DATE events on a wide set of data channels (raw files, named pipes and network channels) and for all type of DATE events (streamlined, paged, fully-built events).

16.4.2 The low-level recording library

The low-level recording library is used by processes who need full control over their output channels. This is the case, for example, for the ***eventBuilder*** process on the GDC. The low-level recording library is able to handle a set of channels (for multiple parallel output streams) and allows both synchronous and asynchronous output.

16.4.2.1 The callable interface

The callable interface is defined by the C include file ***${DATE_RECORDLIB_DIR}/recordingLib.h***. This file must be used within a standard DATE environment in order to compile the calling program.

The library is capable of handling multiple channels. All operations must indicate which channel they are related to. This is done with an index in the range [0 .. ***maxChannel***-1], where ***maxChannel*** is the value returned by the library call ***recordingLibDeclareDevice***. For some entries, it is possible to use the pre-defined symbol ***ALL_CHANNELS***: in this case the entry operates on all channels that have been declared.

The library allows a completion handler routine to be declared. This will run as a coroutine as soon as any I/O completes.

Each outstanding output can have an optional user pointer. This is the address of an anonymous block of data associated to the channel, usually describing a data structure related to the pending output. The caller program can - at any time - set or get the user pointer associated to any channel. Normal usage is to declare the user

pointer during the setup of a write operation and to retrieve it at the end of the operation.

This library makes use of the DATE ***database*** package. The corresponding library must be included in the linking stage of the user code.

The library makes use of the DATE ***infoLogger*** package. The appropriate library must be included in the linking stage of the user code. Normal log messages are issued using the ***recordingLib*** stream. Error and fatal messages are also recorded onto the ***runLog*** stream.

The shared memory control region is updated by the library. The run parameters ***runNumber***, ***maxFileSize***, ***ldcSocketSize*** and ***maxEventSize*** are used for setup and run-time checks. The counters ***fileCount***, ***bytesRecorded*** and ***eventsRecorded*** are updated at run-time.

All sizes used in this library are expressed in bytes.

recordingLibDeclareDevice

C Synopsis

```
#include "recordingLib.h"

int recordingLibDeclareDevice( char *recordingDevice )
```

Description Declare the output channel(s) according to the given `recordingDevice`. The syntax of the recording device follows the conventions described in Section 10.2.

Returns The number of channels corresponding to the recording device (zero for error).

recordingLibOpenChannel

C Synopsis

```
#include "recordingLib.h"

int recordingLibOpenChannel( int channel )
```

Description Open the channel with the given ID (***ALL_CHANNELS***: all channels are opened). If the call fails, the channel(s) is / are left in **`closed`** state.

Returns ***TRUE*** on success, ***FALSE*** on error.

recordingLibCloseChannel

C Synopsis

```
#include "recordingLib.h"

int recordingLibCloseChannel( int channel )
```

Description Close the given channel (or all the channels if ***ALL_CHANNELS*** is used as channel ID). On error, the channel(s) is / are left in an undefined state (all channels that can be closed are closed).

Returns ***TRUE*** on success, ***FALSE*** on error.

recordingLibSetCallback

C Synopsis

```
#include "recordingLib.h"

int recordingLibSetCallback( void callback( int channel ) )
```

Description Declare the routine to be called on completion for each output. The routine will receive the channel ID as the input parameter.

Returns ***TRUE*** on success, ***FALSE*** on error.

recordingLibStartSOR

C Synopsis

```
#include "recordingLib.h"

int recordingLibStartSOR( void )
```

Description Declare the beginning of the "start of run" phase.

Returns ***TRUE*** on success, ***FALSE*** on error.

recordingLibEndSOR

C Synopsis

```
#include "recordingLib.h"

int recordingLibEndSOR( void )
```

Description Declare the end of the "start of run" phase.

Returns ***TRUE*** on success, ***FALSE*** on error.

recordingLibSetupWrite

C Synopsis

```
#include "recordingLib.h"

int recordingLibSetupWrite( int channel,
```

```
                                    void *buffer,
                                    int length,
                                    void *uptr )
```

Description Setup the write for the given channel of the data in the given buffer for the given size. The given user pointer can be retrieved at any time during and after the output.

Returns ***TRUE*** on success, ***FALSE*** on error.

recordingLibSetupWriteV

C Synopsis

```
#include <sys/uio.h>
#include "recordingLib.h"

int recordingLibSetupWriteV( int channel,
                             struct iovec *iov,
                             int iovcnt,
                             void *uptr )
```

Description Setup the write for the given channel of the data in the given I/O vector of the specified length. The given user pointer can be retrieved at any time during and after the output. For more details on the ***iovec*** structure, see the man page relative to the ***writev*** Unix system call.

Returns ***TRUE*** on success, ***FALSE*** on error.

recordingLibWriteNext

C Synopsis

```
#include "recordingLib.h"

int recordingLibWriteNext( int channel )
```

Description Write the next data on the given channel. If the channel is set as non-blocking, the call writes only what is possible to write without blocking the operation. If the channel is set as blocking, the call will **stall** until the write is completed.

Returns The number of bytes written, zero if none, -1 on error.

recordingLibSetUptr

C Synopsis

```
#include "recordingLib.h"

int recordingLibSetUptr( int channel,
                         void *uptr )
```

Description Set the user pointer associated to the given channel.

Returns ***TRUE*** on success, ***FALSE*** on error.

recordingLibGetUptr

C Synopsis

```
#include "recordingLib.h"

void *recordingLibGetUptr( int channel )
```

Description Get the user pointer associated to the given channel.

Returns The user pointer associated to the channel (either via `recordingLibSetUptr` or during the `recordingLibSetupWrite`/`recordingLibSetupWriteV` call), ***NULL*** if not set or on error.

recordingLibSetPortNumber

C Synopsis

```
#include "recordingLib.h"

int recordingLibSetPortNumber( int port )
```

Description Set the TCP/IP port number to be used for network channels. This call affects only the channels that have not yet been opened.

Returns ***TRUE*** on success, ***FALSE*** on error.

recordingLibSetBlocking

C Synopsis

```
#include "recordingLib.h"

int recordingLibSetBlocking( int blockingMode )
```

Description Set the channels as blocking (`blockingMode` ***TRUE***) or non-blocking (`blockingMode` ***FALSE***).

Returns ***TRUE*** on success, ***FALSE*** on error.

recordingLibGetChannelName

C Synopsis

```
#include "recordingLib.h"

char *recordingLibGetChannelName( int channel )
```

Description Get the name of the given channel.

Returns String containing the name of the channel on success, ***NULL*** on error.

recordingLibGetFd

C Synopsis

```
#include "recordingLib.h"

int recordingLibGetFd( int channel )
```

Description Get the number associated to the given channel as returned by the Unix `open` system call.

Returns Number of file descriptor on success, -1 if the channel is closed or on error.

recordingLibGetNumWrites

C Synopsis

```
#include "recordingLib.h"

int recordingLibGetNumWrites( int channel )
```

Description Get the number of write operations requested on the given channel.

Returns Number of operations requested, -1 on error.

recordingLibGetNumBytes

C Synopsis

```
#include "recordingLib.h"

long64 recordingLibGetNumBytes( int channel )
```

Description Get the number of bytes written through the given channel.

Returns Number of bytes written, -1 on error.

recordingLibGetChannelByGdcId

C Synopsis

```
#include "event.h"
#include "recordingLib.h"

int recordingLibGetChannelByGdcId( eventGdcIdType gdcId )
```

Description Get the ID of the channel associated to the given GDC ID.

Returns ID of the channel, -1 on error.

recordingLibDumpDatabase

C Synopsis

```
#include "recordingLib.h"

void recordingLibDumpDatabase( void )
```

Description Dump the content of the library database via an ***infoLogger*** stream.

16.4.3 The high-level recording library

The high-level recording library provides an abstract access layer to the low-level recording library. The ***recorder*** process running on the LDCs uses the high-level recording library. The high-level recording library can handle only DATE raw events and uses an approach similar to the one implemented in the low-level recording library. Basically, a set of channels is handled all together and many events can be sent simultaneously and asynchronously to any of the open channels. The library then handles the relations with the guest Operating System to queue and perform in parallel all the outstanding transfers. The dynamic resources associated to the calling process are used to adapt to the operating conditions. The library also handles the association of the event to the appropriate output channel.

16.4.3.1 The callable interface

The callable interface is defined by the C include file ***${DATE_RECORDLIB_DIR}/dateRec.h***. This file must be included in a standard DATE environment in order to compile the calling program.

All conventions valid for the low-level recording library (see Section 16.4.2 above) are also valid for the high-level recording library.

This library makes use of the DATE ***database*** package, the DATE ***infoLogger*** package, and the low-level recording library. The file ***${DATE_RECORDINGLIB_BIN}/libDateRec.a*** should be used to link the calling code.

The library issues log messages using the ***infoLogger dateRec*** stream. Errors are also sent to the ***infoLogger runLog*** stream.

The run parameter ***recordingDevice*** is used by this library (in addition to all the parameters and counters handled by the low-level recording library).

dateRecInit

C Synopsis

```
#include "dateRec.h"

int dateRecInit( void )
```

Description Initialize the library. Can be called only once in the lifetime of the calling process.

Returns 0 on success, non-zero on error.

dateRecSetup

C Synopsis

```
#include "dateRec.h"

int dateRecSetup( void *event, void *userPtr )
```

Description Initialize the transfer of the given DATE event. The given user pointer will be returned whenever the transfer is either completed or aborted.

Returns 0 on success, non-zero on error.

dateRecGetCompleted

C Synopsis

```
#include "dateRec.h"

int dateRecGetCompleted( int timeout,
                         void **event,
                         void **userPtr )
```

Description Get the data relative to the next completed transfer (if any). The timeout can be equal to 0 (do not wait), -1 (wait forever) or the minimum amount of milliseconds to wait for the next available completed transfer (the actual wait time could exceed the given value for fragmented outputs or for heavy loaded systems). The event pointer must be provided while the `userPtr` pointer can be ***NULL***.

Returns 0 on success, non-zero on error. If an output has been completed, the event pointer will contain the address of the data written, otherwise ***NULL*** will be returned. The `event` pointer will **always** be overwritten using whatever value was given in the transfer setup routine.

dateRecGetNumPendings

C Synopsis

```
#include "dateRec.h"

int dateRecGetNumPendings( void )
```

Description Get the number of outstanding write operations.

Returns Number of outstanding operations (completed, in progress or pending), 0 if none.

dateRecShutdown

C Synopsis

```
#include "dateRec.h"

int dateRecShutdown( int forceShutdown )
```

Description Close all channels in a graceful way (`forceShutdown` ***FALSE***) or by aborting all outstanding operations (`forceShutdown` ***TRUE***): aborted writes can be retrieved via the `dateRecGetAborted` call described below.

Returns 0 on success, non-zero on error (or on the impossibility of shutting down the system due to pending operations and forceShutdown set to ***FALSE***).

dateRecGetNumAborted

C Synopsis

```
#include "dateRec.h"

int dateRecGetNumAborted( void )
```

Description Get the number of write operations aborted due to errors that can be retrieved via the `dateRecGetAborted` call.

Returns Number of aborted operations, 0 if none.

dateRecGetAborted

C Synopsis

```
#include "dateRec.h"

int dateRecGetAborted( void **event, void **userPtr )
```

Description Get the data associated to the next aborted operation. This call can be iterated to get - one by one - all outstanding aborted operations.

Returns 0 on success, non-zero on error.

dateRecLastError

C Synopsis

```
#include "dateRec.h"

char *dateRecLastError( void )
```

Description Get a string describing the last error encountered by the library.

Returns Pointer to a zero-terminated static string.

16.4.4 Internals

The two recording libraries (high-level and low-level) are self-contained in the ***${DATE_RECORDLIB_DIR}*** directory. This folder contains the low-level and the high-level recording libraries, the two include files and a small validation program.

The `validator.c` program tests the capability of the low-level recording library to handle a set of parallel output local files. It can be called with several optional parameters the most important of which is the target directory (default: "`/tmp/recordingLibValidation`") used to create the test files. This area must be big enough to store several megabytes of raw data that will be created by the validation program and removed upon termination. Run the program with parameter "`-?`" for a complete list of the available options.

Interfaces

17

This chapter discusses the interfaces of DATE with other systems.

17.1 Interface with the Trigger System

The trigger system provides the synchronization between the experiment and the data acquisition. It identifies the events that are supposedly worth to be read out and activates the data–acquisition system. This role of the Trigger is documented in Chapter 8.

The Trigger system is also a source of data that is read-out by the DAQ. The ALICE Central Trigger Processor will generate three data streams to the DAQ:

1. the CTP event fragment sent for every Trigger Level 2 accept (L2a) consists of 8 words carrying the same information that is broadcast to all the participating sub-detectors through the TTC B-channel. The format of the data is given in Figure 17.1.

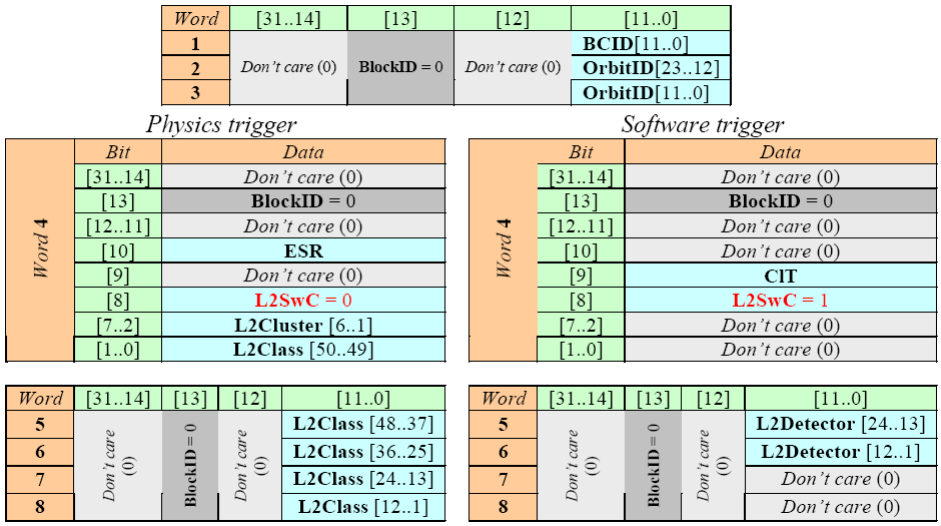

Figure 17.1 The format of the CTP event data.

2. the interaction record (see Figure 17.2) consists of:
   - a two-word header, consisting of an orbit number (first orbit of the record) and an Err flag, asserted if there is a gap just before the record in the continuous sequence of interaction records (under normal circumstances, the DAQ should receive interaction records for all LHC orbits).
   - a maximum of 250 words containing bunch crossing numbers in which interactions have been detected with ***InT*** flag set to zero (0) for peripheral events or set to 1 for semi-central interactions.
   - an optional incomplete record word, present when there are more than 250 interactions, indicated by a virtual bunch crossing number equal to 4095.

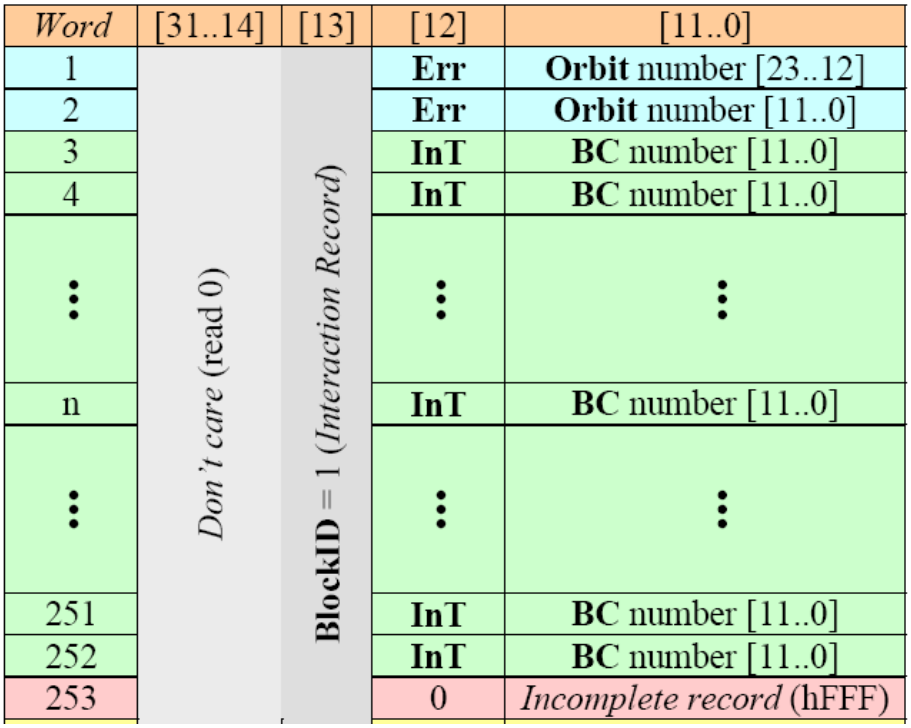

Figure 17.2 The format of interaction records.

The CTP event fragments and the interaction record data shall be generated by the CTP and transmitted to the DAQ via the ALICE DDL. The hardware and the communication procedure shall be standard - identical to the channels that transmit the sub-detector readout. The nature of the data, and the timing and rate of their generation, on the other hand, differ significantly from the sub-detector readout and shall be formatted by a “customized” data format as indicated before.

The CTP Readout will contribute to the event-building task. It is a redundant channel that carries exactly the same information broadcast, at the time of an L2a decision, to all the participating sub-detectors (L2a Message). It will be used by the ALICE data-driven DAQ system to resolve error conditions.

The Interaction Record is an aid to the pattern recognition task. The generation of the record is continuous, rather than “triggered” by any CTP or DAQ action. The data do not “interfere” with any on-line operation - they only need to be archived for the off-line use.

17.2 Interface to the High-Level Trigger

The overall architecture of the Trigger, DAQ and High-Level Trigger (HLT) systems is illustrated in Figure 17.3.

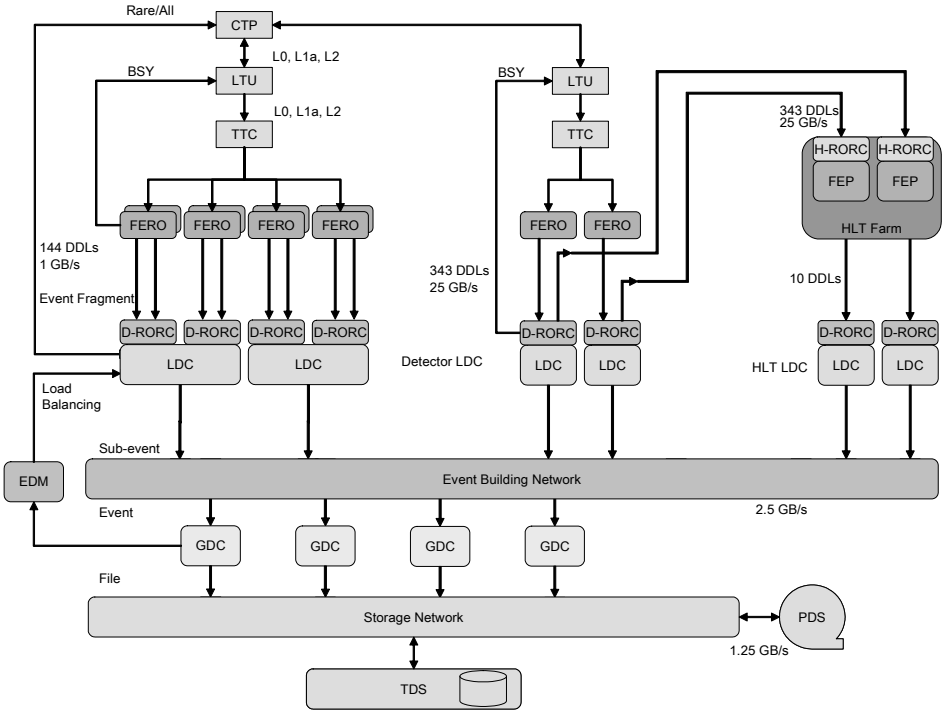

Figure 17.3 TRIGGER-DAQ-HLT overall architecture.

The data-acquisition system takes care of the data flow from the DDL up to the storage of data on the PDS system.

The task of the HLT system is to select the most relevant data from the large input stream and to reduce the data volume by well over an order of magnitude in order to fit the available storage bandwidth, while preserving the physics information of interest. This is achieved by a combination of event selection (triggering), data compression, or selection of Regions of Interest with partial detector readout. While executing either of these tasks, the HLT may also generate data to be attached to or partially replacing the original event.

Care has been taken not to impose any architectural constraints which could compromise the HLT filtering efficiency, knowing that event selection will become more and more elaborated during the experiment lifetime. This way, filtering may be introduced in progressively sophisticated steps without affecting the performance and the stability of the data-acquisition system.

17.2.1 DAQ-HLT interface

A schematic view of the DAQ-HLT interface is illustrated in Figure 17.4.

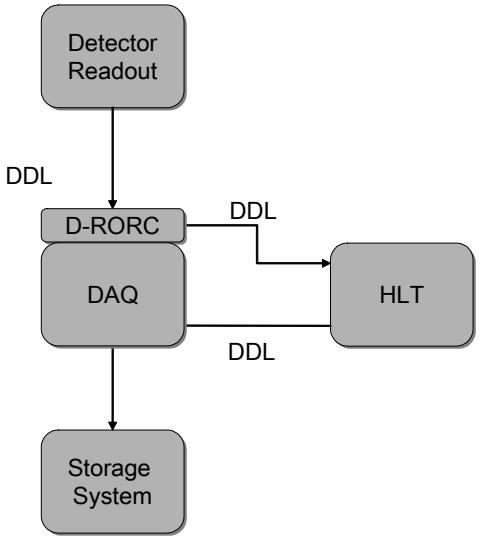

Figure 17.4 DAQ-HLT interface schematic view.

The hardware interface is based on the DDL and its DIU / SIU cards, the same components used to transfer data from the detector electronics to the data-acquisition system.

The DAQ system is implemented within a coherent hardware and software framework, with the HLT system operating as an external system, as shown in Figure 17.5.

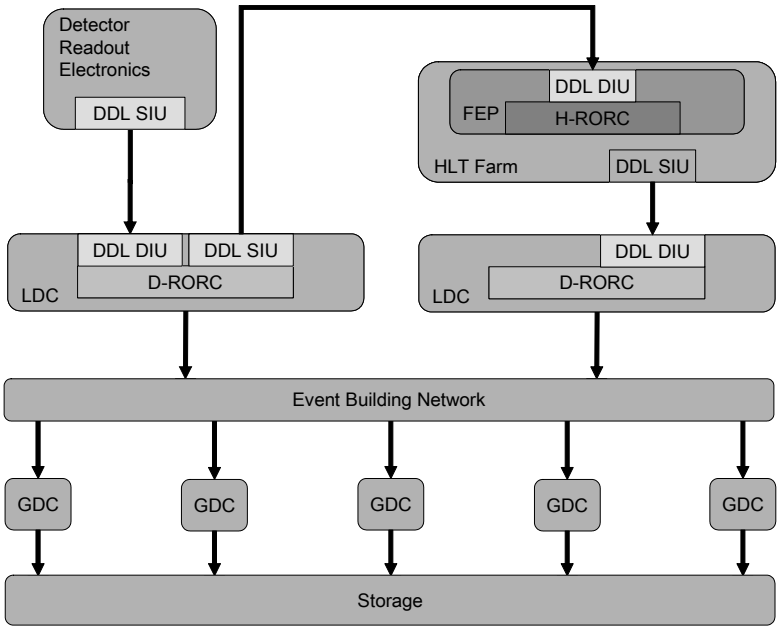

Figure 17.5 DAQ-HLT Data Flow overview.

Every D-RORC sitting in the LDC can host two DIUs. These on-board DIUs can be used in two ways: both can be connected to the front-end electronics and serve as two readout link, or one DIU can be connected to the front-end electronics while the other is able to transfer a copy of all the raw data to the HLT RORC (H-RORC) sitting in the HLT computers, through the standard DDL. The H-RORC receives all

the raw data as it has received from the front-end electronics. All the LDCs dedicated to the detectors which make use of the HLT system are equipped with D-RORCs working in the second mode. These are called Detector LDCs. The interface between the DAQ and the HLT system is the DIU output on the H-RORC.

17.2.2 HLT-DAQ interface

After running the HLT algorithms, the HLT computers transfer the result of the processing, the trigger decisions, and the compressed data to the DAQ system, using again standard DDLs. Here the interface is the SIU input.

The GDCs receive the sub-events from the Detector LDCs and any additional data generated by the HLT computers from the LDCs dedicated to the HLT, called HLT LDCs. The DATE software can accept as many data channels from the LDCs dedicated to the HLT as required, since it handles these channels as additional LDC data paths.

The HLT LDCs will also receive messages specifying whether to discard or accept a given event. Furthermore, for accepted events, the HLT decision can specify the new pattern of sources for a given event, resulting in a partial readout of the raw data. A decision broker process, running on the HLT LDCs, will transfer the HLT information and decision to a decision agent process, running on the detector LDCs, as shown in Figure 17.6.

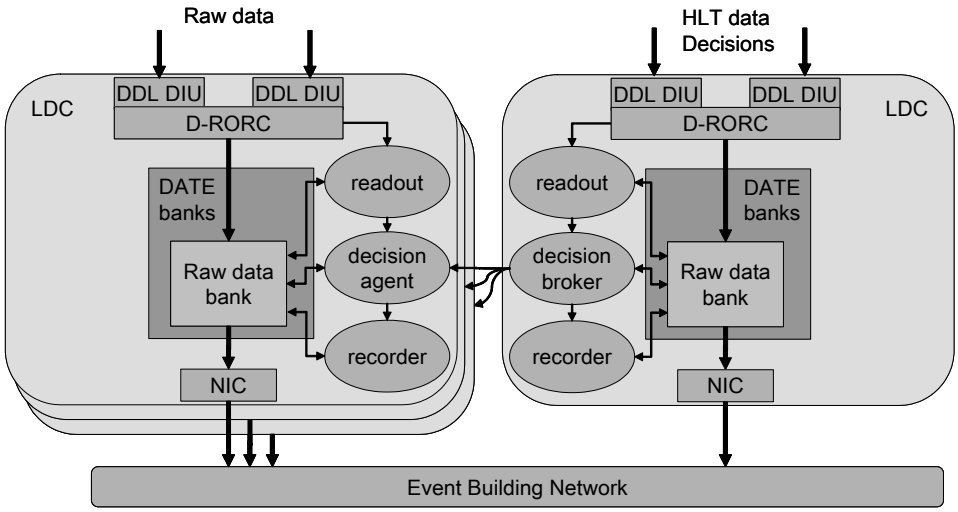

Figure 17.6 Data flow in the LDC in the DAQ system with HLT active.

The functions of the decision broker and of the decision agent are implemented by the ***hltAgent*** process, started on all the LDCs of the DAQ system whenever the DATE resources are configured to run in an environment where the HLT system is active. This process runs a server loop - similar to those of the ***recorder*** and ***eventBuilder*** - waiting for incoming events, sending and receiving HLT decisions and forwarding the result to the ***recorder*** process.

17.2.3 Installation and operation

The ***hltAgent*** requires a system with a minimum of one LDC declared as a HLT LDC and at least one LDC declared as a Detector LDC. It needs the entity ***hltReadyFifo*** (of size equivalent to that of the ***readoutReadyFifo***) and the entity ***hltDataPages***, big enough to contain the headers of the accepted events (this entity - when declared via IPC - can be oversized as the real limiting factor will come from the ***readoutReadyFifo*** and the ***readoutDataPages***).

When active, the ***hltAgent*** will use the ***infoLogger*** package (see Chapter 11) to create log messages with facility set to ***hltAgent***.

The standard output and standard error streams from the ***hltAgent*** are available in the file ***${DATE_SITE_TMP}/${DATE_HOSTNAME}/hltAgent/hltAgent.log***. A set of files are kept available to keep a historical record over a few runs.

The **hltAgent** must be started using the script ***${DATE_HLT_AGENT_BIN}/hltAgent.sh.*** The script needs no parameters and must be run within an adequate DATE environment (normally setup by the ***rcServer*** process of the DATE run control).

The ***hltAgent*** can be configured in two modes: operation and emulation.

When running in operation mode, the incoming events are considered as HLT decisions and / or HLT payloads and are treated as such.

When running in emulation mode, the content of the incoming events is ignored, the ***hltAgent*** assumes that all the events received handled by the HLT LDCs must be considered as a potential HLT decision and it takes decisions based on a static configuration file. This file, called ***${DATE_SITE_CONFIG}/hltEmulation.config***, describes the behavior of the ***hltAgent*** via two text lines containing the following information:

1. Relative ratios of HLT decisions taken between all the ***hltAgent***s running on all the HLT LDCs, expressed as a list of integer values with the number of decisions to be taken by each ***hltAgent*** running on the HLT LDCs, e.g.:

   ```
   10 2 1 1 1
   ```

   This configures the ***hltAgent*** running on the first (as given by the host role ID) HLT LDC to create 10 decisions every 15 events, the ***hltAgent*** running on the second HLT LDC to create 2 decisions every 15 events and the ***hltAgent***s running on the third, fourth and fifth HLT LDC to create one decision every 15 events. It is possible to give more ratios than the HLT LDCs actually in use (the extra ones will be ignored); however, every HLT LDC must have a corresponding entry.

2. For each trigger class, the percentage of events to be fully accepted, of events to be rejected and - for events to be partially accepted - the percentage of active Detector LDCs to accept with a + / - range, e.g.:

   ```
   CENTRAL_TRIGGER 20 30 10 2
   ```

   This tells the ***hltAgent***, for events marked with CENTRAL_TRIGGER, to accept 20% of the incoming traffic, to reject 30% of the incoming traffic and to

accept the remaining 100 - 20 - 30 = 50% of the incoming events using 10% +/- 2% of the active Detector LDCs.

17.2.4 Synchronization between hltAgents

The ***hltAgent***s synchronize themselves using non-blocking TCP/IP channels. When the run starts, each ***hltAgent*** running on a HLT LDC connects to the next ***hltAgent*** using the order defined by the host ID. The ***hltAgent*** running on the HLT LDC with highest host ID connects to the ***hltAgent*** running on the HLT LDC with lower host ID. The result is a circular path that connects sequentially all the HLT LDCs.

Next, all ***hltAgent***s running on the HLT LDCs connect to a subset of the Detector LDCs (the full set of Detector LDCs is split into subsets with about the same size). The result is a tree of depth 1 with the HLT LDCs as root nodes and the Detector LDCs as leaves, as shown in Figure 17.7.

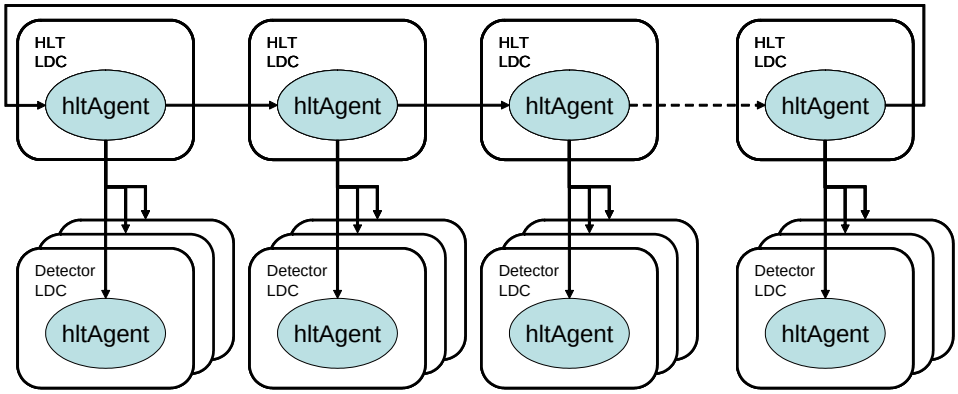

Figure 17.7 Interconnections between hltAgents.

At run-time, when an LDC receives an event, this is given to the ***hltAgent*** who finds out if a HLT decision is due or not. If not, the packet is passed as-is to the next element in the chain (***recorder*** or ***edmAgent***). If instead a HLT decision is due, the ***hltAgent***s running on the HLT LDCs check if the event is a HLT Decision or a HLT Payload. If the event is a HLT Decision, this is decoded, interpreted and the result is sent both to the ***hltAgent*** running on the next HLT LDC and to all the connected Detector LDCs. If instead the event is not a HLT decision or if the ***hltAgent*** runs on a Detector LDC, then the ***hltAgent*** checks if a HLT decision for the given event has been already received, in which case the appropriate action is taken, or if there is no decision yet, and the event is added to a list of "pending" events. The ***hltAgent***s also wait for incoming HLT decisions and - when these are received - they are either applied (if the event is found in the list of "pending" events) or put aside for later use. ***hltAgent***s running on HLT LDCs also forward incoming HLT decisions to the next HLT LDC in the chain. All communications between ***hltAgent***s are handled asynchronously and do not block the agent itself.

17.3 Interface to AliEn and the Grid

Files written by DATE must eventually migrate to the Grid. This implies two separate operations: a copy of the file itself in Permanent Data Storage (PDS) and the registration of the file and its content in ***AliEn***.

17.3.1 Transfer to PDS

The data files, which are written in ROOT format, are transferred to PDS using one of the available protocols. For the PDS currently in use at ALICE (CASTOR) the XROOTD protocol is used (the RFCP protocol, used heavily in the past, is courrently obsoleted).

In PDS, all the files are stored below a common root. The first level of partitioning concerns files written during global runs and files written during standalone runs. For the first class of files, the directory is named "***global***" while the second type of files are catalogued using the name of the detector.

The second level of partitioning is meant to control the number of entries per directory (too many entries would pollute the directory catalogue, creating an unacceptable overload when querying the server). We achieved this objective by creating a tree sorted by year, month, day and hour (GMT) of creation of the individual data files. The information used to create this tree is not meant to be used for reference (for this we have the ***AliEn*** catalogue), it is there just to limit the number of entries in each directory (we could have used any other method for this purpose). In Listing 17.1 we can see an example, with excerpts coming from a snapshot taken in April 2009. Below the root directory in line 3 we can see a separate directory for each detector plus a directory global for the global runs (line 7). Below the global directory we have two directories for 2008 and 2009 (lines 24 and 25). If we expand the 2009 directory and descent into 04 (month of April) and 02 (2^{nd} of April) we see (lines 28 to 37) one directory per each our of acquisition, where the data files (lines 40 and 41) are stored. An identical structure is shown for an example taken from the directory dedicated to ZDC standalone runs (lines 43 to 55) that took place September 18^{th} 2008.

Listing 17.1 Example of directory structure on CASTOR

```
 1: $ rfdir -R /castor/cern.ch/alice/raw
 2:
 3: /castor/cern.ch/alice/raw:
 4: acorde
 5: emcal
 6: fmd
 7: global
 8: hmpid
 9: muon_trg
10: muon_trk
11: phos
12: pmd
13: sdd
14: spd
15: ssd
16: t0
17: tof
18: tpc
19: trd
20: v0
21: zdc
22: [...]
23: /castor/cern.ch/alice/raw/global:
24: 2008
25: 2009
26: [...]
27: /castor/cern.ch/alice/raw/global/2009/04/02:
28: 01
29: 03
30: 06
31: 07
32: 10
33: 12
34: 13
35: 19
36: 20
37: 23
38: [...]
39: /castor/cern.ch/alice/raw/global/2009/04/02/23:
40: 09000067672031.10.root
41: 09000067672032.10.root
42: [...]
43: /castor/cern.ch/alice/raw/zdc/2008/09/18:
44: 08
45: 09
46: 11
47: 12
48: 14
49: 15
50: 16
51: 22
52: 23
53: [...]
54: /castor/cern.ch/alice/raw/zdc/2008/09/18/23:
55: 08000060012031.10.root
```

The filenames have been optimized in agreement with the ***AliEn*** file catalogue, for efficiency and manageability. The syntax of the names is the following:

- two digits for the year
- nine digits for the run number
- three digits for the host ID
- dot

- file sequential ID (arbitrary number of characters, must be anything unique within the run: we use the file sequential number, which is unique within each stream, followed by the stream number)
- the file type: "***.root***" or "***.tag.root***"

The result from the above encoding is a filename which is guaranteed to be unique in the lifetime of ALICE (it merges information coming form the run number, the host ID, the stream ID and the sequential number within the stream) and, at the same time, allows searches based on various key parameters (to ease operator's intervention on the ***CASTOR*** namespace). See Listing 17.1, lines 40, 41 and 55, for examples of the syntax.

During the ***ROOT***ification procedure, a file of file type "***.root.guid***" containing the **GUID** information (sort of unique identifier for the data stored in the events) is created by ***AliRoot*** for each ***ROOT*** file. The content of the `GUID` file is used during the registration procedure with ***AliEn*** and then the file itself is removed.

Another file created by the ***ROOT***ification procedure is the **TAG** file (filetype "***.tag.root***"). A **TAG** file is created for each run. This file is copied to ***CASTOR*** using the same syntax as for the associated ***ROOT*** file. It is also registered in ***AliEn*** (without **GUID** as this information does not apply to **TAG** files).

Once a file it has been moved, it must get registered in the AliEn files catalogue. For this procedure we need the following information:

- file size (in bytes)
- full filename in ***CASTOR***
- file creation time
- ***LHC*** period associated to the run of the file
- **MD5** checksum of the file (calculated either during the transfer of the file or manually after the transfer)
- **GUID** information of the file (***ROOT*** files only)

All the above information is stored in a stamp file that cal be handled in two different ways: via the **AliEn registration gateway** or via the **alienspoold daemon**. The two mechanisms can be used indivividually or together. The recommended one is the **AliEn registration gateway**.

The **AliEn registration gateway** is a machine that can be reached from the DAQ TDSM daemons via the SCP protocol and that has access to the network where the ***AliEn*** files catalogue is connected (usually GPN). At the end of the transfer of a file, the TDSM Mover copies the associated stamp file into a dedicated directory local to the ***AliEn*** registration gateway, where a dedicated software daemon, developed and maintained by the ALICE Offline project, handles it. It is possible to define multiple ***AliEn*** gateways: the TDSM Mover hot-switches to the first available node (an error message will be raised periodically to inform the operator of the abnormal status of the "broken" ***AliEn*** gateway(s)). Whenever no gateway is available, the stamp file is stored in a dedicated TDS area and its registration will be retried by a dedicated ***AliEn*** interface process via the same protocol used by the TDSM Movers. The DAQ/ECS operator handles all events up to the copy of the file into the gateway (including installation and backup of the gateway(s)) while the Offline operator must take care of all events related to the

handling of the information stored within the stamp file and its registration in the AliEn catalogue.

The **alienspoold daemon** works via dedicated directory on TDS. This directory is periodically polled by a dedicated process named ***alienspoold***. The daemon, developed and maintained by the ALICE Offline project, discovers the stamp file, takes care of the registration procedure using the information stored therein and removes the stamp file when done. For obvious reasons, the ***alienspoold*** process must run on a machine that has access to both the TDS network (where the stamp file is created) and to whatever network is used to host the ***AliEn*** files catalogue (usually GPN). Files with incomplete or invalid syntax are rejected by `alienspoold` and stored in a special ".garbage" sub-directory, where they can be retrieved for post-mortem analysis. The health status of the `alienspoold` process is continuously verified by the TDSM package, that takes care of saving the logs and of restarting the daemon if necessary, up to the notification to the DAQ/ECS operator whenever needed. It is the responsibility of the ECS/DAQ operator to handle events such as rejected registrations or failures in the protocol from/to the ***AliEn*** catalogue server. Periodic warnings may be issued by the TDSM package whenever stamp files keep piling up, pointing to a failure somewhere in the registration chain.

17.4 File Exchange Server

The File Exchange Server (sometime named FES or FXS) is a transient storage to exchange information between DAQ and other systems (e.g. Offline, DCS, HLT). Some DAQ processes may copy files to the server, which stores them until they are picked up (e.g. by an external system). Each file is supposed to be read once only after it has been stored, by a single consumer, and will then be removed. The File Exchange Server is not meant to publish a file to be used by multiple remote consumers. The paradygm used is: store from DAQ, read by a single remote consumer, and then delete.

The File Exchange Server consists of a server hosting the files and related meta information stored in a MySQL database, and a client API to access the files from the DAQ.

We describe in this chapter the DAQ File Exchange Server, i.e. a mechanism to export files from DAQ to other systems. DCS and HLT have implemented their own File Exchange Server based on this architecture to export their files.

The server part relies on:

- a local directory, which must be accessible remotely by password-less `scp` or `sftp` for a given user, from the DAQ nodes where File Exchange Server access is needed. To enforce maximum security in production areas, and make sure the server is not used for other purpose than copying files, this remote user login is usually setup with the restricted shell implemented by the `rssh` package. It is however not mandatory, and the DAQ user needed to store the files can be defined as a normal interactive user. The DAQ user should have write access to the storage area. The external systems reading the files are given only read access to the files.

- a database to store the metada associated to the hosted files. This is used as a directory entry to know what files are available, and to flag the files once they have been used remotely. The information is stored in a single table, described in Table 17.1. The DAQ user should have write access to the table, whereas external systems need only read access, and update on the ***time_processed*** column to flag the files which they have retrieved.

The DAQ File Exchange Server is primarily used to export results from the Detector Algorithms (see Chapter 22) to the Offline Shuttle.

Each file stored in the File Exchange Server has a unique name as defined by the field ***filePath*** which is derived from other identifiers to make it unique.

Table 17.1 File Exchange Server ***daqFES_files*** table

Field	Description
run	The run number during which the file was created.
detector	The code of the detector creating the file.
fileId	The local file Id, which should be unique for a given process (***DAQsource***) in a given run. However, similar processes running in parallel on different DATE roles may use the same fileId. For example, the different instances of the same pedestal DA running on each LDC of a given detector may all export a file with the same ***fileId*** like ***pedestal.root***.
DAQsource	The DATE role where the file was created, as defined by the ***DATE_ROLE_NAME*** environment variable set at runtime by the ***runControl*** launching the process.
filePath	The unique identifier of the file. It is built from a combination of ***run***, ***detector***, ***fileId*** and ***DAQsource*** to ensure unicity.
time_created	The time at which the file was stored in the File Exchange Server.
time_processed	The time at which the file is flagged to be deleted. When an external process reads a file from the File Exchange Server, it should update this field to notify the File Exchange Server that the file has been retrieved and can be removed.
time_deleted	The time at which the file has eventually been removed from the File Exchange Server, after it has been retrieved and flagged as such by a non-***NULL*** `time_processed` field.
size	The file size.
fileChecksum	The MD5 checksum of the file.

Access to the File Exchange Server from a DAQ process requires that the following two environment variables are defined (in the environment section of the DATE configuration database, with ***editDb***, seeChapter 4.4):

- ***DATE_FES_DB***: access parameters to the File Exchange Server database. This variable should be defined with the syntax `user:password@hostname:dbname` where `user` / `password` are the credentials to access the File Exchange Server database named `dbname` running on `hostname`.
- ***DATE_FES_PATH***: access parameters to the File Exchange Server file repository.

This variable should be defined with the syntax `user@host:path` so that a command like `scp myfile user@hostname:path` would copy `myfile` to the File Exchange Server repository directory on `hostname`.

The database may be created using the ***${DATE_INFOLOGGER_DIR}/daqFES_create.sql*** SQL command script.

The File Exchange Server may be cleaned by running periodically (with appropriate rights) a script as the example provided ***${DATE_INFOLOGGER_DIR}/daqFES_clean*** which should be adapted according to the needs (e.g. not destroying the files right away, but maybe moving them to an archive of the last ones as disk space allows). The database table may also be used for consistency checks (file size and MD5 sum), or to identify orphan entries (e.g. a file on disk with no database entry, or a database entry with no corresponding file on disk). Upon deletion of a File Exchange Server file, the files on disk are removed, but thecorresponding entries in the database should be left for logging and statistics purposes.

The following command line tools are available in ***${DATE_INFOLOGGER_DIR}*** to access the File Exchange Server from a DAQ node:

- `daqFES_ls` : list the files available in the File Exchange Server. The environment variables ***DATE_RUN_NUMBER*** and ***DATE_DETECTOR_CODE*** may be defined (and then used as filters).
- `daqFES_store mylocalfile myId`: store a local file named `mylocalfile` on the File Exchange Server using identification `myId`. The environment variables ***DATE_RUN_NUMBER***, ***DATE_ROLE_NAME*** and ***DATE_DETECTOR_CODE*** should be defined.
- `daqFES_get filePath`: get the file named `filePath` from the File Exchange Server (or all of them if `filePath` not provided) and mark them as used so that they may afterwards be deleted. The environment variables ***DATE_RUN_NUMBER*** and ***DATE_DETECTOR_CODE*** may be defined (and then used as filters).

17.5 Interface to the Shuttle

The Shuttle is an Offline process that collects some information from DAQ after a run is finished in order to populate the Offline Condition DataBase (OCDB). This is meant in particular to retrieve the calibration results produced by the Detector Algorithms (see Chapter 22).

The Shuttle is triggered either internaly by a timeout (periodical checks for new runs), or by the DAQ service providing end of run notification by DIM. This service (among others) is implemented in the ***logbookDaemon*** and named, as defined in ***DAQlogbook.h***, ***/LOGBOOK/SUBSCRIBE/ECS_EOR***. The service is updated with a run number each time a run is completed (ECS completion).

A dedicated logbook table named ***logbook_shuttle*** (defined in ***logbook_create.sql***) is populated by the ECS to tell the Shuttle which detectors are active in a run, and to keep the status of Shuttle processing for each of

them. The Shuttle gets the list of new runs from this table, and updates it accordingly when it processes them.

A special flag ***test_mode*** may be set manually when needed to identify some runs where it is not wished that the results are taken into account by the Shuttle to populate OCDB. This may be the case to test new Detector Algorithms.

The Shuttle accesses the logbook database and the DAQ File Exchange Server by direct SQL queries; there is no API for this purpose.

Part II

ALICE Experiment Control System Reference Manual

November 2010

ALICE ECS Project

ECS & ACT

Preface

The ALICE Experiment Control System (ECS) coordinates the activities performed on the particle detectors when running the experiment. These activities concern the operation and control of the detectors from the hardware point of view, the acquisition of experimental or calibration data, the Trigger system, and the High Level Trigger. They are called ‘online systems’.

The ECS has been designed and implemented as a layer of software on top of the existing 'online systems' controlling the different activities. The ECS imposes only one constraint to these systems: they must provide status information and eventually accept commands through interfaces based on Finite State Machines (FSMs).

The FSM package used in ALICE is ***SMI++*** [4]. The ECS heavily relies on it and on the DIM [3] communication package both for the implementation of the interfaces between ECS and 'online systems' and for the implementation of the ECS major components.

This part of the manual describes the ECS.

- The integration between the ECS and the different 'online systems' is not equally developed.
- Some of the detectors did not implement yet a DCS based on FSM: therefore the DCS states of these detectors are not included yet in the ECS.
- Some of the detectors have not yet developed their calibration procedures.
- Some information on the configuration of the ‘online systems’, such as the definition of the Trigger classes, is not available yet and therefore the ECS uses now some temporary definitions.

The ECS will evolve to include the above issues as soon as they will be available. Its architecture has been tested during beam tests, and proved to be solid and flexible enough to include all the future extensions.

ECS Overview

18

This chapter describes the architecture of the Experiment Control System (ECS), its various components and their interactions.

18.1 Introduction

The ALICE experiment consists of several particle detectors. Running the experiment implies performing a set of activities with these detectors. In ALICE these activities are grouped into four activity domains: Detector Control System (DCS), Data Acquisition (DAQ), Trigger (TRG) and High Level Trigger (HLT).

Every activity domain requires some form of coordination and control: independent control systems have been developed for all of them. These systems, called 'online systems', are independent, may interact with all the particle detectors, and allow partitioning. Partitioning is the capability to concurrently operate groups of ALICE detectors called partitions.

Before being operated together to collect physics data in the ALICE final setup, detectors were prototyped, debugged, and tested as independent objects. While this operation mode, called 'stand-alone mode', was absolutely vital in the commissioning and testing phase, it is also required during the operational phase to perform calibration procedures on individual detectors. Therefore it remains essential during the whole life cycle of ALICE.

The Experiment Control System (ECS) coordinates the operations of the 'online systems' for all the detectors and within every partition. It permits independent, concurrent activities on part of the experimental setup by a same or different operators.

The components of the ECS receive status information from the 'online systems' and send commands to them through interfaces based on Finite State Machines. The interfaces between the ECS and the 'online systems' contain access control mechanisms that manage the rights granted to the ECS: the 'online systems' can either be under the control of the ECS or be operated as independent systems. In the second case the 'online systems' provide status information to the ECS, but do not receive commands from it.

18.2 Partitions

A partition is a group of particle detectors. From the ECS point of view, a partition is defined by a unique name that makes it different from other partitions and by two lists of detectors: the list of detectors assigned to the partition and the list of detectors excluded from the partition.

The first list, called ***assigned detectors list***, contains the names of the ALICE detectors that can be active within the partition. This static list represents an upper limit for the partition: only the detectors listed in the ***assigned detectors list*** can be active in the partition, but they are not necessarily active all the time. The ***assigned detectors lists*** for different partitions may overlap: a same detector can appear in different ***assigned detectors lists***. ***Assigned detectors lists*** cannot be empty.

The second list, called ***excluded detectors list***, contains the names of the ALICE detectors that have been assigned to the partition, but are currently not active in the

partition. This dynamic list is a subset of the ***assigned detectors list***. It can be empty.

Although a given detector appears in the ***assigned detectors list*** of many partitions, it cannot be active in several partitions at the same time, but only in one or none of them: the ***excluded detectors list*** of a partition contains the names of the detectors that are not active in the partition because they are active in another one, or because they are operated in stand-alone mode, or because of an explicit operator request. Explicit operator requests are subject to restrictions: the structure of a partition cannot be changed by the exclusion and inclusion of detectors during the data-taking phase.

Two types of operations can be performed in a partition: operations involving all the active detectors and operations involving only one active detector. The operations of the first type are called global operations; those of the second type are called individual detectors operations.

The ECS handles the global operations watching the DCS status of all the active detectors and interacting with the ***runControl*** process that steers the data acquisition for the whole partition, with the ***Trigger Partition Agent (TPA)*** that links the partition to the Central Trigger Processor (CTP), and with the HLT proxy that controls the HLT operations for the partition. When a global operation starts, it inhibits all the individual detectors operations.

The ECS handles an individual detector operation watching the DCS status of the detector and interacting with the ***runControl*** process that steers the data acquisition for that particular detector,with the Local Trigger Units (LTU) associated to it, and with the HLT proxy that controls the HLT operations for that detector.. When an individual detector operation starts, it inhibits the global operations, but it does not inhibit individual detector operations executed on other detectors: these individual detector operations, such as calibration procedures, can be concurrently performed within the partition.

18.3 Stand-alone detectors

A stand-alone detector is a detector operated alone and out of any partition. The operations performed with a stand-alone detector are equal to the ***i***ndividual detector operations that can be done on the detector when this one is active in a partition: the ECS handles these operations watching the DCS status of the detector and interacting with the ***runControl*** process that steers the data acquisition for that detector, with the LTU associated to it, and with the HLT proxy that controls the HLT operations for the detector.

The major difference between a stand-alone detector and a partition with only one single detector is that the partition with only one detector is linked to the CTP by a ***TPA***, whereas the stand-alone detector only interacts with its LTU.

18.4 ECS architecture

Every detector operated in stand-alone mode or assigned to a partition is controlled by a process called ***Detector Control Agent (DCA)*** and every partition is controlled by a process called ***Partition Control Agent (PCA)***.

When a detector is operated in stand-alone mode, its ***DCA*** accepts commands from an operator that issues commands from a ***DCA Human Interface***. Several ***DCA Human Interfaces*** can coexist for the same ***DCA***, but only one can send active commands at a given time: the others can only get information.

When a detector is active in a partition, its ***DCA*** accepts commands only from the ***PCA*** controlling the partition. Operators can still open ***DCA Human Interfaces,*** but only to get information and not to send active commands.

A ***PCA Human Interface*** provides to an operator the full control of a partition. Many ***PCA Human Interfaces*** can coexist for the same ***PCA***, but only one has the control of the partition at a given time and can be used so send active commands.

DCAs and ***PCA***s get status information from the 'online systems' and eventually send commands to components of these systems through interfaces based on Finite State Machines.

This chapter describes the components of the ECS and the interface between the ECS and the 'online systems'.

18.5 Detector Control Agent (DCA)

There is a ***DCA*** for every detector operated in stand-alone mode or assigned to a partition. The main tasks performed by this process are the following:

- It handles stand-alone data-acquisition runs for the detector working alone. This function requires the coordination and the synchronization of the detector's hardware controlled by the DCS, of the detector's Front End Read-Out (FERO), of a ***runControl*** process steering the data acquisition for the given detector only, of the HLT activities performed for the detector, and of the LTU associated to this detector. This function is implemented for all the detectors but not in the same way because detectors have specific requirements.
- It handles detector specific procedures, such as calibration procedures. These procedures are by definition detector dependent and therefore their implementation is different for each detector.

The ***DCA*** is implemented as an SMI domain [4]. The name of the domain is given by the detector name suffixed with '_DCA'. For example, the ***DCA*** controlling the HMPID detector is implemented as an SMI domain whose name is ***HMPID_DCA***.

In addition to the objects required to perform the main tasks described above, the SMI domain contains other objects that allow the following features:

- When the detector is active in a partition and as long as a global action is being

executed in the partition, the ***PCA*** controlling the partition keeps the ***DCA*** informed about the global action going on: the ***DCA*** goes in an ***INHIBITED*** status and waits for the global action to terminate. The information flow goes from the ***PCA*** to the ***DCA***.

- When the detector is active in a partition and as long as an individual detector operation is being executed for the detector, the ***DCA*** keeps the ***PCA*** controlling the partition informed about the action going on: the ***PCA*** goes in an ***INHIBITED*** status and waits for the action to terminate. The information flow goes from the ***DCA*** to the ***PCA***.

The ***DCA*** accepts commands from one master operator at a time: either a ***PCA*** or a ***DCA Human Interface***.

18.6 The DCA Human Interface

An operator can control the detector in stand-alone mode with a ***DCA Human Interface*** having got the mastership of a ***DCA***. He / she can send commands to the ***DCA***, can change the rights granted to the ***DCA***, and can send commands directly to objects in the HLT, DAQ, and TRG 'online systems'. Without the mastership of the ***DCA***, the ***DCA Human Interface*** can only get information, but it cannot issue active commands.

A detailed description of the ***DCA Human Interface*** can be found in the ALICE DAQ WIKI.

18.7 Partition Control Agent (PCA)

There is a ***PCA*** per partition. The main tasks performed by this process are the following:

- It handles PHYSICS and TECHNICAL runs using all the detectors active in the partition. This function requires the coordination of the status, from the hardware and FERO point of view, of all the active detectors, of a ***runControl*** process steering the data acquisition for the whole partition, of the ***TPA*** associated to the partition and of the HLT proxy controlling the HLT activities for the partition. This function is implemented in a same way for all the partitions.
- It delegates individual detectors operations to the ***DCA***s controlling the detectors active in the partition.
- It handles the partition structure allowing the inclusion / exclusion of detectors in / from the partition, whenever these operations are compatible with the data-taking going on for individual detectors or for the whole partition.

The ***PCA*** is implemented as an SMI domain [4]. The name of the domain is given by the partition name suffixed with '***_PCA***'. For example, the ***PCA*** controlling the ITS partition is implemented as an SMI domain whose name is ***ITS_PCA***.

In addition to the objects required to perform the main tasks described above, the SMI domain contains other objects that allow the following features:

- When a detector is active in a partition and as long as a global action is being executed in the partition, the ***PCA*** keeps the ***DCA*** controlling the detector informed about the global action going on: the ***DCA*** goes in an ***INHIBITED*** status and waits for the global action to terminate. The information flow goes from the ***PCA*** to the ***DCA***.
- When a detector is active in a partition and as long as an individual detectors operation is being executed for the detector, the ***DCA*** controlling the detector keeps the ***PCA*** informed about the action going on: the ***PCA*** goes in an ***INHIBITED*** status and waits for the action to terminate. The information flow goes from the ***DCA*** to the ***PCA***.

The ***PCA*** accepts commands from one ***PCA Human Interface*** at a time.

18.8 The PCA Human Interface

An operator can control a partition with a ***PCA Human Interface*** having got the mastership of a ***PCA***. He/she can send commands to start global and individual detectors operations, can change the rights granted to the ***PCA***, can change the structure of the partition excluding or including detectors, and can send commands directly to objects in the HLT, DAQ, and TRG 'online systems'. Without the mastership of the ***PCA***, the ***PCA Human Interface*** can only get information, but it cannot issue active commands.

A detailed description of the ***PCA Human Interface*** can be found in the ALICE DAQ WIKI.

18.9 ECS/DCS Interface

The DCS describes the ALICE experiment as a hierarchy of particle detectors and of infrastructure services. Its model of ALICE is based on Finite State Machines and is implemented as a tree structured set of SMI domains and objects. Within this tree every detector is represented by a different sub-tree of SMI objects. The status of the objects being the roots of these sub-trees are the status of the different detectors seen from the DCS point of view.

The interface between the ECS and the DCS mainly consists of one object per detector: the roots of the sub-trees described above and representing the detectors within the DCS. These objects provide status information to the central DCS and, at the same time, to the ECS. A second object, called Run Control Unit, informs the ECS about the availability of the detector for running (i.e. even a READY detector may want to be excluded from runs).

Figure 18.1 is an example where two detectors, named 'y' and 'z' are active in an ECS partition named 'A'. The figure shows the double role of the SMI objects that provide status information for the two detectors both to the DCS and to the ECS.

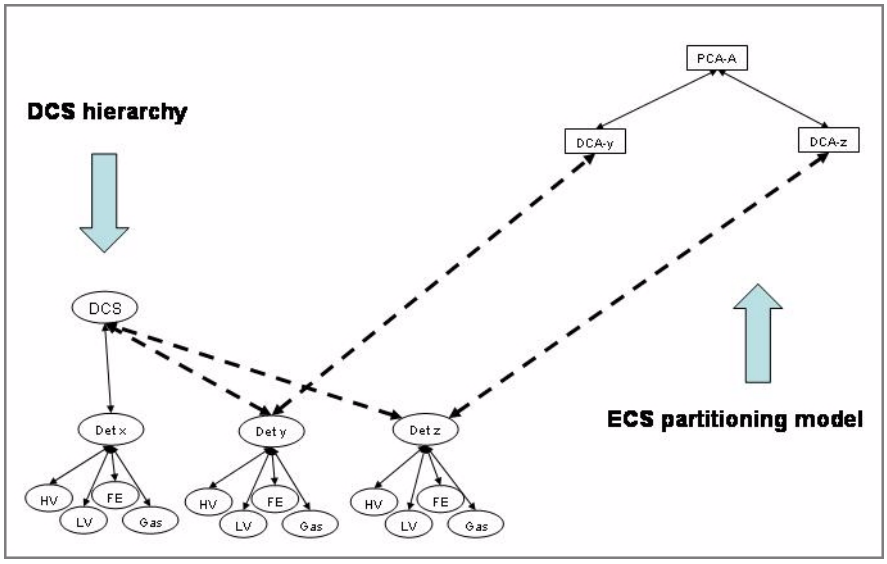

Figure 18.1 ECS/DCS interface.

18.10 ECS/DAQ Interface

The interface between the ECS and the DAQ is made of SMI objects representing ***runControl*** processes:

- A ***runControl*** process per detector: every ***runControl*** process steers the data acquisition for a given detector and for that detector only. The name assigned to the process is equal to the detector name.
- A ***runControl*** process per partition: it steers the data acquisition for the whole partition with data produced by all the active detectors. The name assigned to the process is equal to the partition name prefixed with '***ALL***'. If, for example, the partition name is ALICE, then the name of the ***runControl*** process is ***ALLALICE***.

18.11 ECS/TRG Interface

An SMI domain named '***TRIGGER***' contains the objects describing the basic Trigger components: the LTUs associated to the detectors and the CTP. These SMI objects are associated to processes, called proxies, that actually drive the LTUs and the CTP.

All the detectors active in a partition produce raw data when a global operation is performed; the generation of raw data by the detectors is done under the control of their associated LTUs. These LTUs are synchronized by the CTP.

There is one CTP, but many partitions can be operated at the same time and all of them need access to the CTP. The ***Trigger Partition Agents (TPA**s**)*** associated to the different partitions solve the access conflicts. There is one ***TPA*** per partition. The ***TPA***s are implemented as SMI objects in SMI domains. The name of these domains is made by the partition names suffixed by '***_TRG***'. The ***TPA*** for a partition named ALICE is an SMI object named TPA in an SMI domain named ***ALICE_TRG***. The ***TPA*** interacts with CTP and LTUs.

When a detector is operated in stand-alone mode, the ***DCA*** controlling it directly interacts with the LTU associated to the detector. The CTP is ignored.

When a detector is active in a partition and an individual detectors operation is executed on it, the ***PCA*** delegates the operation to the ***DCA*** controlling the detector. The ***DCA*** again interacts with the LTU associated to the detector. The CTP is ignored.

When a global operation is performed in a partition, the ***PCA*** controlling the partition interacts with the ***TPA*** that in turn interacts with CTP and LTUs. The ***PCA*** has no direct interaction with CTP and LTUs.

Figure 18.2 shows the ECS / TRG interface.

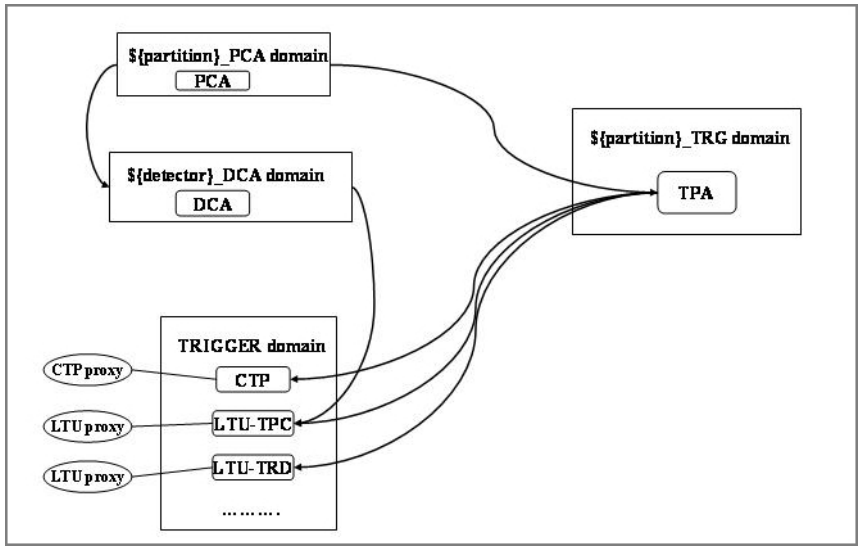

Figure 18.2 ECS/TRG interface.

18.12 ECS/HLT Interface

The interface between the ECS and the HLT is made of SMI objects representing ***HLT proxy*** processes:

- An ***HLT proxy*** process per detector: every ***HLT proxy*** process steers the

HLT activities for a given detector and for that detector only.

- An ***HLT proxy*** process per partition: it steers the HLT activities for the whole partition with data produced by all the active detectors.

18.13 logFiles

The ECS components use the DATE ***infoLogger*** package to record error and information messages.

The `stdout` files created by ***DCA***s and ***PCA***s are stored in a directory pointed to by the environment variable ***ECS_LOGS***. The name of the files are self explanatory.

When working with dummy versions of HLT, DCS and Trigger (for debugging purposes) the `stdout` files created by the dummy processes are stored in the directories pointed to by the environment variables ***HLT_LOGS***, ***DCS_LOGS*** , and ***TRG_LOGS***.

18.14 Database

The ECS components require configuration data. This information is stored in a MySQL database (see the ALICE DAQ WIKI). The database also contains additional runtime information available through a Web interface (see the ALICE DAQ WIKI). In particular, this interface shows the activities being performed for each detector and for each partition.

18.15 Interactions with other systems

In addition to its main activity (i.e. the synchronization of the ‘online systems’ to perform runs), the ECS interacts with other components of the ALICE software. In particular:

- Sends SOR and EOR commands to the central ALICE DCS at the beginning and at the end of every standalone or global run.
- Sends SOR and EOR commands to the LHC_MON process at the beginnig and at the end of every global run.
- Sends to the Alice Configguration Tool (ACT) requests to lock / unlock configuration items to prevent configuration changes during runs.
- Stores in the ALICE eLogbook information about all the performed runs.

18.16 Auxiliary processes

All the **DCA**s and all the **PCAs** require the presence of some auxiliary processes:

- **ecs_timeout** used to interrupt SMI commands after reasonable delays.
- **ecs_counter** to count the number of iterations performed during some calibration procedures or the number of elements in some sets.
- **stringsProxy** required to compare SMI parameters.
- **ecs_logger** to store infoLogger messages.
- **ecs_operator** required to start some special operator commands, such as starting the migration of data.
- **ecs_daq_db_handler** handling all the interactions with the DAQ and ECS databases.

The **PCAs** require more auxiliary processes:

- **pca_updateDB** to keep track of the detectors excluded / included from / in partitions.
- **pca_updateTIN** to update the list of detectors used as trigger detectors for a partition.

19 ALICE Configuration Tool

The ALICE Configuration Tool (ACT) is the first step to achieve a high level of automation, implementing automatic configuration of the different detectors and online systems. Having already contributed to the reduction of the size of the shift crew needed to operate the experiment, the ACT is a central actor in ALICE's activities, allowing the Run Coordination and the Shift Leaders to operate the experiment in a global way. This chapter describes the architecture of the ACT and its different components, the interfaces with the different online systems and the Web-based Graphical User Interface.

19.1 Architecture

19.1.1 Overview

The operation of the ALICE experiment over several years to collect billions of events acquired in well defined conditions requires repeatability of the experiment's configuration. Appropriate software tools are therefore needed to automate daily operations, minimizing human errors and maximizing the data-taking time. The ALICE Configuration Tool (ACT) fulfills these requirements, allowing the automatic configuration of the different systems and detectors.

The base concept of the ACT is to serve as a configuration repository to which the different ALICE systems can access to extract their currently selected configuration. As shown in Figure 19.1, the ACT is operated both by the Run Coordination and the different system experts via a Web-based Graphical User Interface (GUI). A relational database (DB) serves as a data repository and an Application Programming Interface (API), implemented in C, provides numerous functionalities to the different components.

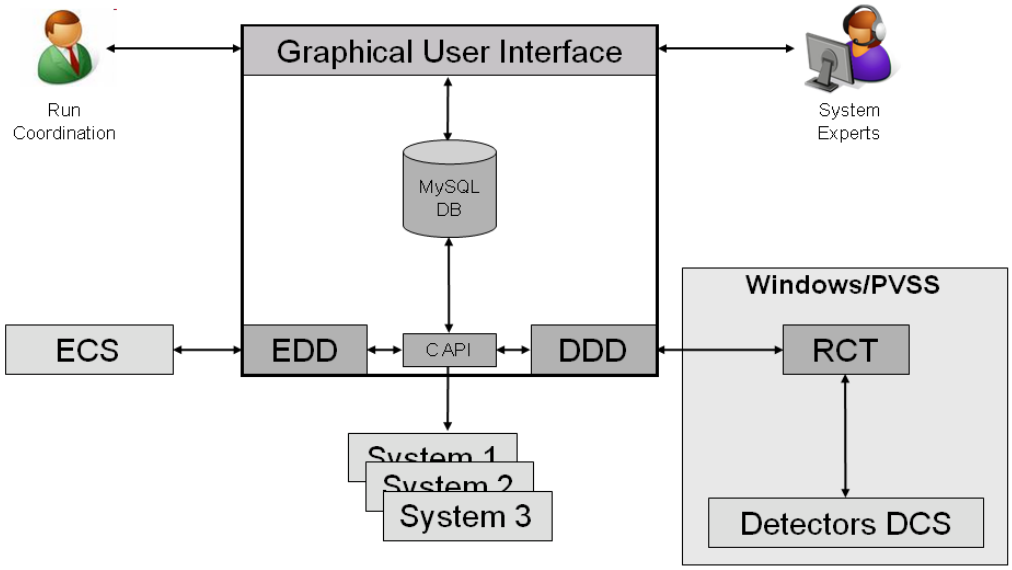

Figure 19.1 The architecture of the ACT and its interfaces with the different online systems and detectors.

A publish / subscribe mechanism, based on the Distributed Information Management (DIM) system, is also available. Two dedicated modules, running as daemon processes, use this mechanism:

- ***ECS Dedicated Daemon*** (***EDD***): interacts with the Experiment Control System (ECS), ***pushing*** the selected configurations for the online systems.
- ***DCS Dedicated Daemon*** (***DDD***): interacts with the ***Run Control Tool*** (***RCT***), ***pushing*** the selected configurations for the different ALICE detectors. The ***RCT*** then makes the configurations available to each individual Detector Control System (DCS) where the detector configuration is executed.

19.1.2 Taxonomy

In order to define the different systems and detectors components to configure, the ACT introduces the following concepts:

- ***System***: an ACT ***system*** represents a physical or logical element of the ALICE experiment. Each ***system*** normally has several configurable components. Examples of **V\VWHPV** are: detectors, online systems, ECS partitions.
- ***Item***: an ACT ***item*** corresponds to a configurable component of a specific ACT ***system***. Each ***item*** normally has several possible configurations defined. Examples of an ***item*** are: 'partition PHYSICS_1 HLT Mode', 'TPC DCS configuration', 'CTP L0 inputs'.
- ***Instance***: an ACT ***instance*** defines a possible predefined configuration for a specific ACT ***item***. At any given time, only one ***instance*** can be activated for each ***item***.

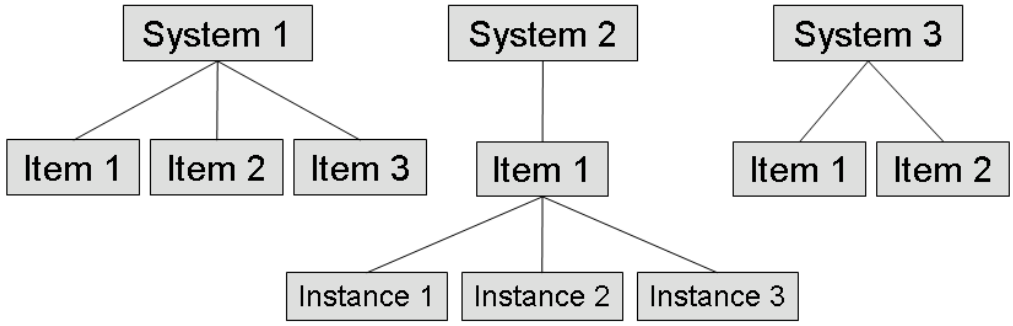

Figure 19.2 ACT hierarchy.

19.1.2.1 Items Locking

A locking mechanism prevents the configuration of ***items*** that are either being configured or used by an online system (e.g. a detector being part of a running partition).

For configuration, the ***items*** are locked by the corresponding daemon (***EDD*** or ***DDD***). If being used by an online system, the ***items*** are locked by that system.

19.1.2.2 Items Status Mismatch

The status mismatch flag allows to identify ***items*** whose configuration has changed outside the control of the ACT. It is the external tool's responsibility to flag the changed ***item***.

An example of this behavior is the inclusion / exclusion of detectors from an ECS partition using the ECS's human interfaces.

19.1.2.3 Items Activation Status

At a given time, each ***item*** is in a specific state, represented by its activation status. There are four possible values:

- 'update requested': a configuration has been requested for the ***item***.
- 'applying': a configuration is being executed for the ***item***.
- 'active': the ***item*** is configured as requested.
- 'update failed': an error occurred while configuring the ***item***.

Figure 19.3 shows the state diagram for the ***items*** activation status.

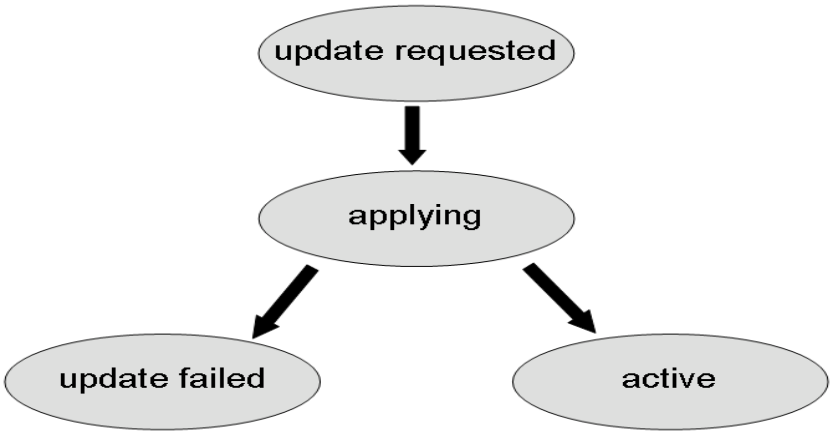

Figure 19.3 Items activation status state diagram.

19.1.3 ACT Update Request Server

The ACT Update Request Server is a DIM server implementing the ***ACT_UPDATE*** DIM service and several DIM commands. The most important are:

- ACT_UPDATE_REQUEST: an update has been requested.
- ACT_TIMEOUT_REQUEST: an abort has been requested.

When a command is received, it updates the ***ACT_UPDATE*** service.

19.1.4 Interfaces

Below is a list of the different ACT interfaces.

19.1.4.1 ACT-ECS interface

The interface between ACT and ECS is implemented by the ***EDD*** daemon process. When an update request is received by ***EDD*** (via the ***ACT_UPDATE*** DIM service), it checks which ECS ***items*** have been marked for update and propagates the corresponding configuration to ECS.

19.1.4.2 ACT-DAQ interface

Communication between ACT and DAQ is implemented by the ***EDD*** daemon process. When an update request is received by ***EDD*** (via the ***ACT_UPDATE*** DIM service), it checks which DAQ ***items*** have been marked for update. Then, depending on the ***item***, two different paths may be followed:

- for parameters which are also controlled by the ECS (e.g recording mode), the configuration is propagated to the ECS.
- for parameters which are only controlled by the DAQ (e.g. number of GDCs), the configuration is propagated to the DAQ.

At Start of Run, several DAQ modules (e.g. TPCC) also download their selected configuration directly from the DB using the C API.

19.1.4.3 ACT-HLT interface

Communication between ACT and HLT is performed via the ECS, which then sends the relevant changes to the HLT system.

19.1.4.4 ACT-CTP interface

Communication between ACT and CTP is performed in two different ways.

For the CTP partition configuration, at Start of Run the selected configuration is downloaded directly from the DB by the CTP system using the C API.

For the CTP global configuration, when an update request is received by ***EDD*** (via the ***ACT_UPDATE*** DIM service), it is transmitted to CTP, which downloads the selected configuration directly from the DB using the C API. The CTP is then restarted to load the new configuration.

19.1.4.5 ACT-Detector interface

The interface between ACT and the ALICE detectors is implemented by two modules: the ***DDD*** daemon process and the ***RCT***. When an update request is received by ***DDD*** (via the ***ACT_UPDATE*** DIM service), it checks which ***items*** have been marked for update and propagates the corresponding information (via DIM) to the ***RCT***.

The ***RCT*** then updates its internal PVSS datapoints for the corresponding detectors and sends them an FSM `CONFIGURE` command. When the detector finishes its configuration, it updates a dedicated datapoint with an acknowledgment message, which is then propagated back to ***DDD*** (via DIM). Based on this message, ***DDD*** then changes the updated ***items*** activation status to either '***active***' or '***update failed***'.

More technical details concerning ***RCT*** can be found in [20].

19.1.5 Workflow

As seen in Figure 19.4, the ACT workflow starts with the user (usually the Shift Leader) selecting the desired configuration via the GUI. When finished, the user will submit an update request, which will change the selected ***items*** activation status to '***update requested***' and trigger the execution of the `ACT_UPDATE_REQUEST` DIM command by the ***ACT Update Request Server***. This command will result in an update of the ***ACT_UPDATE*** DIM service, thus signaling to both ***EDD*** and ***DDD*** that an update was requested.

Items related with ECS, DAQ, HLT and CTP (***item*** categories equal to '***partition***', '***DAQ config***','***HLT config***', and '***CTP config***', respectively) are handled by ***EDD***. Upon receiving the update request, ***EDD*** first locks the ***items*** and sets their activation status to 'applying'. Then, depending on the ***item***, changes will be performed in the corresponding online systems to reflect the new configuration. Finally, the ***items*** are set to either 'active' (on success) or 'update failed' (on failure) and unlocked.

Items related with the DCS (***item*** categories equal to 'DCS config') are handled by ***DDD***. Upon receiving the update request, ***DDD*** first locks the ***items*** and sets their activation status to 'applying'. Then the ***items*** are grouped by detector, and their name and the value of their active ***instance*** concatenated in a string which is passed via DIM to ***RCT*** (by updating the ***ACT_RCT_CONF_DET*** DIM service where DET is replaced by the corresponding 3-letter detector code). ***RCT*** then decodes the received string and populates its internal PVSS datapoints, after which it sends a CONFIGURE command to the detector's FSM. After executing this command, the detector's FSM updates a datapoint with the reply to the configuration, which is sent back to ***DDD*** via DIM. Finally, the ***items*** are set to either '***active***' (on success) or '***update failed***' (on failure) and unlocked by ***DDD***.

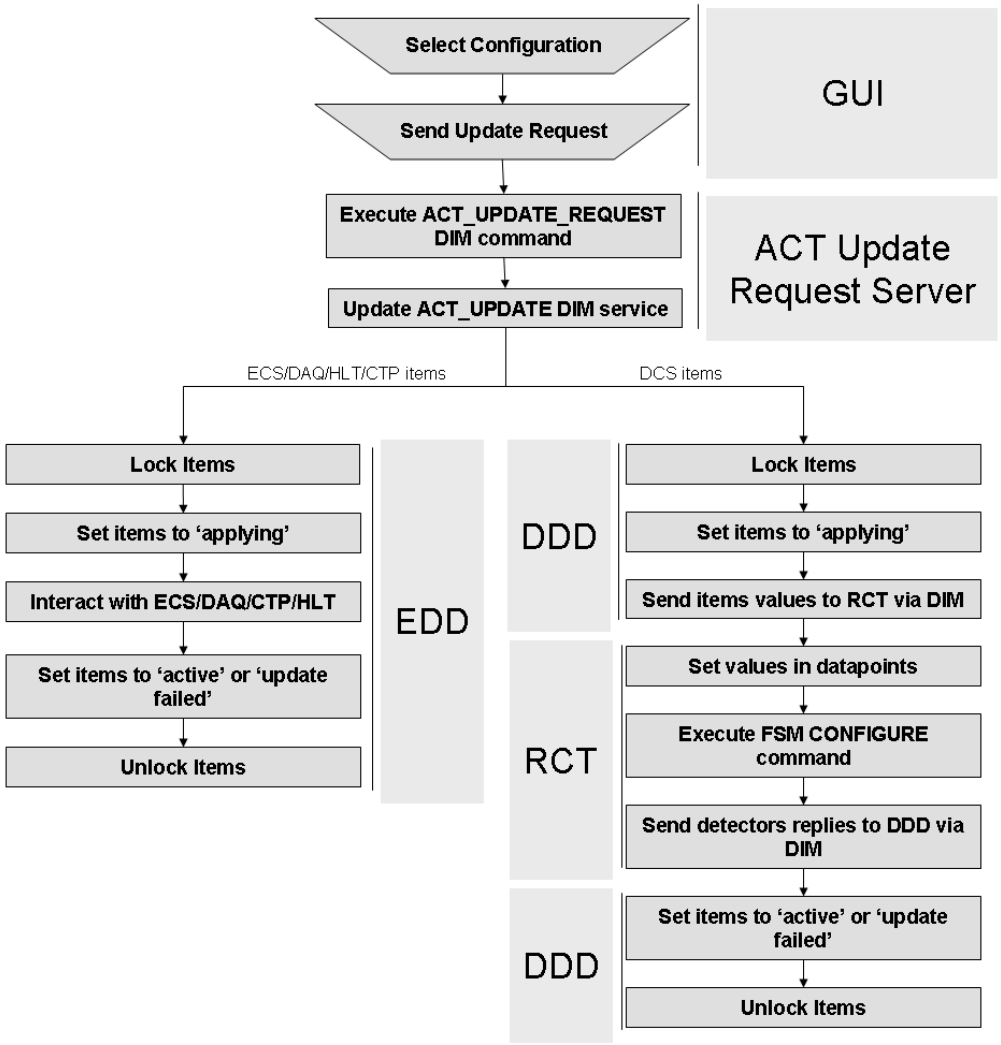

Figure 19.4 ACT workflow diagram.

19.2 Database

19.2.1 Overview

The DB, running on a MySQL Server, is used to store the definition of the different elements of the ACT. InnoDB is used as a storage engine for its support of both transactions and foreign keys constraints.

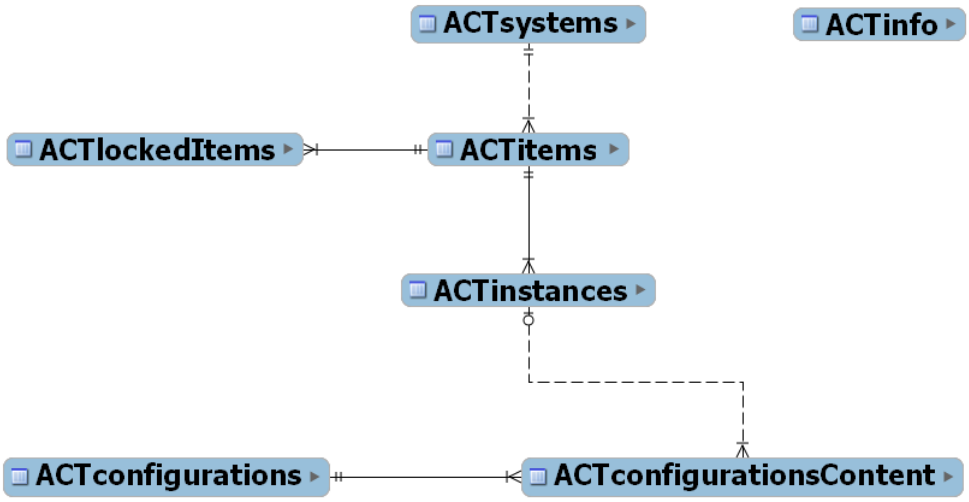

Figure 19.5 ACT database schema.

Daily backups are performed to a RAID 6 disk array and the CERN Advanced STORage manager (CASTOR).

19.2.2 Table description

Below is a description of the ACT's tables.

19.2.2.1 `ACTsystems` table

This table defines ALICE configurable systems, such as the online systems, the ECS partitions or the different detectors.

Table 19.1 `ACTsystems` table

Field	Description
`system`	System name
`description`	System description
`ECScomponent`	Name of the corresponding ECS detector or partition (if applicable)
`systemCategory`	System type, if any ('`partition`', '`detector`')

Table 19.1 *ACTsystems* table

Field	Description
enabled	Flag indicating if system is enabled in ACT
isTriggerDetector	Flag indicating if system is a trigger detector
isReadoutDetector	Flag indicating if system is a readout detector
updateTimeout	Update request timeout in seconds

19.2.2.2 *ACTitems* table

This table defines, for each ***system***, the list of configuration ***items***.

Table 19.2 *ACTitems* table

Field	Description
item	Item name
system	Item's system
description	Item description
itemCategory	Item category, if any ('***CTP config***', '***TRG config***', '***DCS config***', '***HLT config***', '***DAQ config***', '***partition***')
enabled	Flag indicating if item is enabled in ACT
activationStatus	Item's activation status report ('***update requested***', '***applying***', 'αχτιωε', '***update failed***')
statusTimestamp	Timestamp of the latest activation status update
statusMismatch	Flag indicating if item is not in the requested configuration (only meaningful when the item's activation status is equal to '***active***')
activationComment	Comments of latest activation status update

19.2.2.3 *ACTinstances* table

This table defines, for each ***item***, the list of possible (predefined) configurations.

Table 19.3 *ACTinstances* table

Field	Description
instance	Instance name
item	Instance's item
version	Instance version number
description	Instance description
author	Instance author name
creationTime	Instance creation timestamp

Table 19.3 ***ACTinstances*** table

Field	Description
isValidated	Flag indicating if instance is validated (marked as ready to be used)
changeLog	Instance change log
value	Instance value
isActive	Flag indicating if instance is selected (active) for the corresponding item
dependOnDetector	ECS name of detector this instance may depend on at runtime

19.2.2.4 ***ACTlockedItems*** table

This table stores the list of locked configuration ***items***.

Table 19.4 ***ACTlockedItems*** table

Field	Description
item	Item name
lockSource	Name of element which is locking the item
runNumber	Run number which is locking the item (if applicable, zero otherwise)
eventCount	Number of subevents collected by ***readout***
bytesInjected	Size of data collected by ***readout*** in bytes
time_update	Database row update date/time

19.2.2.5 ***ACTconfigurations*** table

This table defines reusable configurations of one or more configuration ***items***, allowing users to configure several ***items*** in one action.

Table 19.5 ***ACTconfigurations*** table

Field	Description
id	Configuration ID
name	Configuration name
target	Target to which the configuration can be applied (e.g. partition name)
wildcards	CSV list of wildcards to be applied
obsolete	Flag indicating if configuration is obsolete
author	Configuration author name
creationTime	Configuration creation timestamp

Table 19.5 *ACTconfigurations* table

Field	Description
description	Configuration description

19.2.2.6 *ACTconfigurationsContent* table

This table stores the content of the reusable configurations defined in the ***ACTconfigurations*** table.

Table 19.6 *ACTconfigurationsContent* table

Field	Description
id	Configuration ID
item	Item name
instance	Instance name
version	Instance version number

19.2.2.7 *ACTinfo* table

This table stores internal ACT information (e.g. version number).

Table 19.7 *ACTinfo* table

Field	Description
variable	Variable name
value	Variable value
description	Variable description

19.3 Application Programming Interface

19.3.1 Overview

Read / write access is available via a C API.

19.3.2 Environment variables

The following environment variables are available to configure the behavior of the API:

- ***ACT_DB:*** sets the credentials to access the DB. The format is ***"USERNAME:PASSWORD@HOSTNAME/DBNAME"***.

- ***ACT_VERBOSE:*** sets the logging level. Possible values are:
  - 0: no messages.
  - 1: error messages.
  - > 2: same as 1 + all SQL queries.

  If not set, the default value is 0.

19.3.3 Data types

Below is a list of the available data types.

ACT_handle

Description Handle to an ACT DB connection.

Listing 19.1 ***ACT_handle*** type definition

```
1:  struct _ACT_handle {
2:    MYSQL *db;       /* Handle to MySQL connection */
3:    char verbose;    /* Flag set to 1 for verbose logs */
4:  };
5:  typedef struct _ACT_handle * ACT_handle;
```

ACT_system

Description Structure defining an ACT ***system***.

Listing 19.2 ***ACT_system*** type definition

```
1:  typedef struct _ACT_system {
2:    char *system;                  /* system name */
3:    char *ECScomponent;            /* corresponding ECS
4:                                     component, if any (or NULL) */
5:    ACT_t_systemCategory category; /* system category */
6:  } ACT_system;
```

ACT_t_systemCategory

Description Enumerated type defining the ***system*** categories to which a ***system*** can belong.

Listing 19.3 ***ACT_t_systemCategory*** type definition

```
1:  typedef enum {
2:    ACT_system_partition,
3:    ACT_system_detector,
4:    ACT_system_none,      /* category undefined */
5:    ACT_system_any        /* used for search, match any of the
    above */
6:  } ACT_t_systemCategory;
```

ACT_t_systemParams

Description Enumerated type defining the parameters available in a ***system***.

Listing 19.4 ***ACT_t_systemParams*** type definition

```
 1:  typedef enum {
 2:    ACT_system_param_updateTimeout
 3:  } ACT_t_systemParams;
```

ACT_item

Description Structure defining an ACT ***item***.

Listing 19.5 ***ACT_item*** type definition

```
 1:  typedef struct _ACT_item {
 2:    char *item;                          /* item name */
 3:    char *system;                        /* system it belongs to */
 4:    ACT_t_itemCategory category;         /* item category */
 5:    ACT_instance *activeInstance;        /* instance currently active,
 6:                                            may be NULL */
 7:  } ACT_item;
```

ACT_t_itemCategory

Description Enumerated type defining the ***item*** categories to which an ***item*** can belong.

Listing 19.6 ***ACT_t_itemCategory*** type definition

```
 1:  typedef enum {
 2:    ACT_item_CTPconfig,
 3:    ACT_item_TRGconfig,
 4:    ACT_item_DCSconfig,
 5:    ACT_item_HLTconfig,
 6:    ACT_item_DAQconfig,
 7:    ACT_item_Partition,
 8:    ACT_item_none,      /* category undefined */
 9:    ACT_item_any,       /* used for search, match any of the above
    */
10:  } ACT_t_itemCategory;
```

ACT_t_itemActiveStatus

Description Enumerated type defining the activation status in which an ***item*** can be.

Listing 19.7 ***ACT_t_itemActiveStatus*** type definition

```
 1:  typedef enum {
 2:    ACT_activeState_updateRequested,
 3:    ACT_activeState_applying,
 4:    ACT_activeState_active,
 5:    ACT_activeState_updateFailed,
 6:    ACT_activeState_none,              /* undefined */
 7:    ACT_activeState_any,               /* used for search, match
    any of the above */
 8:  } ACT_t_itemActiveStatus;
```

ACT_instance

Description Structure defining an ACT ***instance***.

Listing 19.8 ***ACT_instance*** type definition

```
 1:  typedef struct _ACT_instance {
 2:    char *item;                    /* item name */
 3:    char *instance;                /* instance name */
 4:    void *value;                   /* value content (BLOB) */
 5:    int size;                      /* size of value, in bytes */
 6:    ACT_t_itemCategory category;   /* instance category */
 7:    ACT_t_itemActiveStatus status; /* instance activation status */
 8:    char *dependOnDetector;        /* additional detector
 9:                                      dependence, if any */
10:    char isActive;                 /* 1 if instance active, 0
11:                                      otherwise */
12:  } ACT_instance;
```

19.3.4 Database connection functions

Below is a list of functions providing basic connection functionality to the ACT database.

ACT_open

Synopsis

```
#include "act.h"

int ACT_open(const char *cx_params, ACT_handle *h)
```

Description Open a MySQL connection. Credentials should be given via the ***cx_params*** parameter in the ***"USERNAME:PASSWORD@HOSTNAME/DBNAME"*** format. If an empty string is passed the credentials are taken from the ***ACT_DB*** environment variable. If both are empty, an error will be returned.

If successful, an handle to the DB connection will be stored in the *h* parameter.

Returns Upon successful completion, this function will return a value of zero. Otherwise, an error code with a value equal to the line number where the error occurred will be returned.

ACT_close

Synopsis

```
#include "act.h"

int ACT_close(ACT_handle h)
```

Description Close a MySQL connection and release previously used resources.

Returns Upon successful completion, this function will return a value of zero. Otherwise, an error code with a value equal to the line number where the error occurred will be returned.

19.3.5 API cleanup functions

Below is a list of functions providing memory cleanup. They should be used by client programs to ensure efficient memory usage.

ACT_destroySystem

Synopsis

```
#include "act.h"

int ACT_destroySystem(ACT_system *i)
```

Description Cleanup memory associated with a ***system***.

Returns Upon successful completion, this function returns a value of zero. Otherwise, an error code with a value equal to the line number where the error occurred will be returned.

ACT_destroySystemArray

Synopsis

```
#include "act.h"

int ACT_destroySystemArray(ACT_system *i, int size)
```

Description Cleanup memory associated with an array of ***systems***. The parameter `size` defines the number of ***systems*** to be destroyed.

Returns Upon successful completion, this function returns a value of zero. Otherwise, an error code with a value equal to the line number where the error occurred will be returned.

ACT_destroyItem

Synopsis

```
#include "act.h"

int ACT_destroyItem(ACT_item *i)
```

Description Cleanup memory associated with an ***item***.

Returns Upon successful completion, this function returns a value of zero. Otherwise, an error code with a value equal to the line number where the error occurred will be returned.

ACT_destroyItemArray

Synopsis

```
#include "act.h"

int ACT_destroyItemArray(ACT_item *i, int size)
```

Description Cleanup memory associated with an array of ***items***. The parameter `size` defines the number of ***items*** to be destroyed.

Returns Upon successful completion, this function returns a value of zero. Otherwise, an error code with a value equal to the line number where the error occurred will be returned.

ACT_destroyInstance

Synopsis

```
#include "act.h"

int ACT_destroyInstance(ACT_instance *i)
```

Description Cleanup memory associated with an ***instance***.

Returns Upon successful completion, this function returns a value of zero. Otherwise, an error code with a value equal to the line number where the error occurred will be returned.

ACT_destroyInstanceArray

Synopsis

```
#include "act.h"

int ACT_destroyInstanceArray(ACT_instance *i, int size)
```

Description Cleanup memory associated with an array of ***instances***. The parameter `size` defines the number of ***instances*** to be destroyed.

Returns Upon successful completion, this function returns a value of zero. Otherwise, an error code with a value equal to the line number where the error occurred will be returned.

19.3.6 ACT READ access functions

Below is a list of functions providing READ access to the ACT database. All functions should receive as parameter `h` an handle to the DB connection previously created by a call to the `ACT_open` function.

ACT_getSystems

Synopsis

```
#include "act.h"

int ACT_getSystems(ACT_handle h, ACT_t_systemCategory
category, ACT_system **systemsArray, int *systemsNumber)
```

Description Retrieve the list of all ***systems***. The ***systems*** are stored in the `systemsArray` parameter (`NULL` if no ***systems*** are found) and the number of retrieved ***systems*** in the `systemsFound` parameter.

The `category` parameter can be used to restrict the retrieved ***systems*** to a given ***system*** category. To retrieve all ***systems***, use ***ACT_system_any***.

NOTE: After being used, the ***systems*** should be destroyed via the `ACT_destroySystemArray` function.

Returns Upon successful completion, this function will return a value of zero. Otherwise, an error code with a value equal to the line number where the error occurred will be returned.

ACT_getSystemsToUpdate

Synopsis

```
#include "act.h"

int ACT_getSystemsToUpdate(ACT_handle h, ACT_t_itemCategory
category, ACT_system **systemsArray, int *systemsFound)
```

Description Retrieve the list of ***systems*** with ***items*** for which an update has been requested. The ***systems*** are stored in the `systemsArray` parameter (***NULL*** if no ***systems*** are found) and the number of retrieved ***systems*** in the `systemsFound` parameter.

The `category` parameter can be used to restrict the considered ***items*** to a given ***item*** category. To query all ***item*** categories, use ***ACT_item_any***.

Disabled ***systems*** and ***items*** are ignored.

NOTE: After being used, the ***systems*** should be destroyed via the `ACT_destroySystemArray` function.

Returns Upon successful completion, this function will return a value of zero. Otherwise, an error code with a value equal to the line number where the error occurred will be returned.

ACT_getSystemParamInt

Synopsis

```
#include "act.h"

int ACT_getSystemParamInt(ACT_handle h, const char *system,
ACT_t_systemParams param, int *value)
```

Description Retrieve an integer parameter of a given ***system***. The corresponding value is stored in the `value` parameter.

Returns Upon successful completion, this function will return a value of zero. Otherwise, the following value will be returned:

1: parameter's value is ***NULL***.

< 0: error while retrieving the parameter's value.

ACT_getItem

Synopsis

```
#include "act.h"

int ACT_getItem(ACT_handle h, const char *item, ACT_instance
**instancesArray, int *instancesNumber)
```

Description Retrieve the list of defined ***instances*** of a given ***item***.The ***instances*** are stored in the `instancesArray` parameter (***NULL*** if no ***instances*** are found) and the number of retrieved ***instances*** in the `instancesNumber` parameter.

Disabled ***systems*** and ***items*** are ignored.

NOTE: After being used, the ***instances*** should be destroyed via the `ACT_destroyInstanceArray` function.

Returns Upon successful completion, this function will return a value of zero. Otherwise, an error code with a value equal to the line number where the error occurred will be returned.

ACT_getActiveItem

Synopsis

```
#include "act.h"

int ACT_getActiveItem(ACT_handle h, const char *item,
ACT_instance **instance)
```

Description Retrieve the active ***instance*** of a given `item`. The ***instance*** is stored in the `instance` parameter (***NULL*** if no active ***instance*** is found).

Disabled ***systems*** and ***items*** are ignored.

NOTE: After being used, the ***instance*** should be destroyed via the `ACT_destroyInstance` function.

Returns Upon successful completion, this function returns a value of zero. Otherwise, an error code with a value equal to the line number where the error occurred will be returned.

ACT_getActiveItem_bySystem

Synopsis

```
#include "act.h"

int ACT_getActiveItem_bySystem(ACT_handle h, const char
*system, ACT_t_itemCategory category, ACT_instance
**instancesArray, int *instancesNumber)
```

Description Retrieve the list of active ***instances*** of a given ***system***. The ***instances*** are stored in the `instancesArray` parameter (***NULL*** if no active ***instances*** are found) and the number of retrieved ***instances*** in the `instancesNumber` parameter.

If the `system` parameter is ***NULL***, all ***systems*** are queried.

The `category` parameter can be used to restrict the considered ***items*** to a given ***item*** category. To query all ***item*** categories, use ***ACT_item_any***.

Disabled ***systems*** and ***items*** are ignored.

NOTE: After being used, the ***instances*** should be destroyed via the `ACT_destroyInstanceArray` function.

Returns Upon successful completion, this function will return a value of zero. Otherwise, an error code with a value equal to the line number where the error occurred will be returned.

ACT_getItemsToUpdate

Synopsis

```
#include "act.h"

int ACT_getItemsToUpdate(ACT_handle h, const char *system,
ACT_t_itemCategory category, ACT_item **itemsArray, int
*itemsNumber)
```

Description Retrieve the list of ***items*** (ordered by ***system*** name and ***item*** name) for which an update has been requested. The ***items*** are stored in the `itemsArray` parameter (***NULL*** if no ***items*** are found) and the number of retrieved ***items*** in the `itemsNumber` parameter.

The `system` parameter can be used to restrict the considered ***items*** to a given ***system***. To query all ***systems***, use ***NULL***.

The `category` parameter can be used to restrict the considered ***items*** to a given ***item*** category. To query all ***item*** categories, use ***ACT_item_any***.

Disabled ***systems*** and ***items*** are ignored.

NOTE: After being used, the ***items*** should be destroyed via the `ACT_destroyItemArray` function.

Returns Upon successful completion, this function returns a value of zero. Otherwise, an error code with a value equal to the line number where the error occurred will be returned.

ACT_isLockedItem

Synopsis

```
#include "act.h"

int ACT_isLockedItem(ACT_handle h, const char *item, int
*countLocks)
```

Description Check if an `item` is locked. The `countLocks` parameter stores the number of existing locks for the `item`, if zero the `item` is not locked.

Returns Upon successful completion, this function returns a value of zero. Otherwise, an error code with a value equal to the line number where the error occurred will be returned.

19.3.7 ACT WRITE functions

Below is a list of functions providing WRITE access to the ACT database. All functions should receive as parameter *h* an handle to the DB connection previously created by a call to the ***ACT_open*** function.

ACT_updateActivationStatus

Synopsis

```
#include "act.h"

int ACT_updateActivationStatus(ACT_handle h, const char
*item, ACT_t_itemActiveStatus status, const char *comment)
```

Description Update the activation status of a given ***item*** (***activationStatus*** field of the ***ACTitems*** table). The allowed status transitions are:

- '***update requested***' => '***applying***'
- '***applying***' => '***active***'
- '***applying***' => '***update failed***'

The optional *comment* parameter will update the `activationComment` field of the `ACTitems` table.

NOTE: If the given ***item*** is disabled, an error will be returned.

Returns Upon successful completion, this function returns a value of zero. Otherwise, an error code with a value equal to the line number where the error occurred will be returned.

ACT_updateStatusMismatch

Synopsis

```
#include "act.h"

int ACT_updateStatusMismatch(ACT_handle h, const char *item,
int statusMismatch)
```

Description Update the mismatch flag of a given ***item*** (`statusMismatch` field of the `ACTitems` table). The `statusMismatch` parameter can have the following values:

- 0: the `item` is in the desired configuration
- 1: the `item` is not in the desired configuration

NOTE: If the given `item` is disabled, an error will be returned.

Returns Upon successful completion, this function returns a value of zero. Otherwise, an error code with a value equal to the line number where the error occurred will be returned.

ACT_lockItem

Synopsis

```
#include "act.h"

int ACT_lockItem(ACT_handle h, const char *item, const char
*source, unsigned int run)
```

Description Lock an `item` (create a new row in the `ACTlockedItems` table). The `source` and `run` parameters define the element which is locking the `item`. If a run number is not applicable, zero should be used.

Returns Upon successful completion, this function returns a value of zero. Otherwise, an error code with a value equal to the line number where the error occurred will be returned.

ACT_unlockItem

Synopsis

```
#include "act.h"

int ACT_unlockItem(ACT_handle h, const char *item, const char
*source, unsigned int run)
```

Description Unlock an ***item*** (delete one row from the `ACTlockedItems` table). The `source` and `run` parameters define the element for which the lock should be removed.

Returns Upon successful completion, this function returns a value of zero. Otherwise, an error code with a value equal to the line number where the error occurred will be returned.

19.4 Tools

Below is a list of the available command-line tools providing miscellaneous ACT functionalities.

act_check_daemons.csh

Synopsis `act_check_daemons.csh`

Description Check if the different ACT DIM services and the corresponding DIM Name Servers are available and reachable.

Prior to execution, the following environment variables must be set:

- `DIMDIR`: DIM root directory.
- `DIM_DNS_NODE`: ECS DIM Name Server node.
- `DCS_DIM_DNS_NODE`: DCS DIM Name Server node.

Upon successful completion, the tool will print a list of semicolon separated values indicating the status of each checked service:

- -1: unknown
- 0: not running / reachable
- 1: running and reachable

The order of the printed values correspond to the following DIM services / Name Servers:

- 1: ECS DIM Name Server
- 2: ACT_UPDATE_REQUEST_SERVER (***ACT Update Request Server***)
- 3: ecsDedicatedDaemon (***EDD***)
- 4: ECS DIM Name Server
- 5: dcsDedicatedDaemon (***DDD***)
- 6: ACT_UPDATE (ACT Bridge)
- 7: PVSSSys211Man4:DIMHandler (DCS Run Control Tool)

Returns Upon successful completion, this command returns a value of zero. Otherwise, 1 will be returned.

act_compare_partitions

Synopsis `act_compare_partitions`

Description Check if partition definitions in ECS and ACT are consistent.

Prior to execution, the following environment variables must be set:

- ***ECS_DB_MYSQL_HOST***: ECS database host.
- ***ECS_DB_MYSQL_DB***: ECS database name.
- ***ECS_DB_MYSQL_USER***: ECS database username.
- ***ECS_DB_MYSQL_PWD***: ECS database password.
- ***ACT_DB***: ACT database credentials in the ***"USERNAME:PASSWORD@HOSTNAME/DBNAME"*** format.

Upon successful completion, the tool will print the discrepancies found, if any.

Returns Upon successful completion, this command returns a value of zero. Otherwise, the following values will be returned:

-1: a mandatory environment variable is not set.

-2: an error occurred while retrieving information from the ACT DB.

act_ddd_dummy_dcs_rct

Synopsis `act_ddd_dummy_dcs_rct`

Description Emulates the ***RCT***, allowing test setups to be fully functional.

Prior to execution, the following environment variable must be set:

- `DCS_DIM_DNS_NODE`: DCS DIM Name Server node.

Returns Upon successful completion, this command returns a value of zero. Otherwise, the following values will be returned:

1: `DCS_DIM_DNS_NODE` environment variable is not set.

2: `DIM_DNS_NODE` environment variable could not be set to the value provided in `DCS_DIM_DNS_NODE`.

19.5 Graphical User Interface

19.5.1 Overview

The ACT's Web-based GUI was developed using modern Web technologies, including PHP5, Javascript and Cascading Style Sheets (CSS). It uses the PHP Zend Framework to implement a Model-View-Controller (MVC) architecture.

It is hosted on an Apache web server and can be accessed from the experimental area (inside the experiment's technical network), the CERN General Purpose Network (GPN) and the internet.

19.5.2 Authentication and Authorization

Authentication is implemented via the CERN Authentication central service, providing Single Sign On (SSO) and removing the effort of authenticating the users from the ACT software. This way, when a user tries to access the GUI, he is redirected to the CERN Login page where it has to provide his credentials. If successful, he is then redirected back to the GUI.

Authorization is based on CERN's egroups. In order to access the GUI, users must be members of the ***ALICE-ACT*** egroup.

19.5.3 Expert Mode

The Expert Mode section is used to populate the ACT DB, allowing system experts and ACT administrators to create and modify ***systems***, ***items*** and ***instances***. It also allows the execution of different actions and the display of different status tables.

19.5.3.1 Actions

The following actions are available:

- Send Configuration Request: send a configuration request by executing the ***act_update_request*** command. This will affect all ***items*** with activation status equal to '***update requested***'.
- Put all Detectors in Standalone: activate, for all detectors, the ***instance*** that defines the detector as being in standalone.
- Unlock all Items: remove all ***item*** locks (delete all entries of the ***ACTlockedItems*** table).
- Remove all Status Mismatch: remove all status mismatch flags (set to zero the ***statusMismatch*** field of the ***ACTitems*** table for all ***items***)
- Put all Items in "Active": change the activation status of all ***items*** to '***active***'.
- Disable all Partition CTP Configurations: deactivate, for all ***items*** defining a CTP configuration for a given partition, the currently active ***instance***.

19.5.3.2 Status

The following status reports are available:

- Activation Status: display, for each ***item***, the current activation status. Additional ***item*** information, such as the lock status and the currently active ***instance***, is also displayed.
- Locked Items: display the list of locked ***items***.
- Running Partitions: display the list of running partitions.
- Status Mismatch: display the list of ***items*** with the status mismatch flag set.

19.5.4 Run Coordination Mode

The Run Coordination Mode section is used to configure ALICE during data taking periods, allowing the Run Coordination and the Shift Leaders to globally configure the different ALICE sub-detectors and systems.

19.5.4.1 Partitions

The Partitions subsection allows users to configure the different ECS partitions using a graphical wizard. These configurations can be saved for later reuse, thus allowing for an easier and faster ACT usage). The following modes are available:

- Fully: all partition components (readout detectors, trigger detectors, CTP, DAQ and HLT).
- Readout Detectors: only the partition's readout detectors (including the selected detectors DCS configuration).
- Readout Detectors (without DCS Configuration): only the partition's readout detectors (without including the selected detector's DCS configuration).
- CTP: only the partition's CTP configuration (including the corresponding trigger detectors DCS configuration).
- CTP (without Trigger Detectors): only the partition's CTP configuration (without including the corresponding trigger detectors DCS configuration).
- HLT: only the partition's HLT configuration.
- DAQ: only the partition's DAQ configuration.

Additionally, the following actions can also be executed for each defined partition:

- Change CTP Configuration source to "ACT database" / "Local file": toggle the CTP configuration mode between ACT and local.
- Enable / Disable "Ignore ACT Pending Actions": enable / disable the "***Ignore ACT pending actions***" option in the partition's ***PCA Human Interface***.

NOTE: while a partition is running, no configurations nor actions can be executed (a configuration in mode "Fully" can still be defined and saved).

19.5.4.2 Detectors

The Detectors subsection allows users to individually configure the different ALICE detectors. The following actions are available:

- Put in Standalone: put the detector in standalone (if included in an ECS partition, the detector will be excluded).
- Change Partition: include the detector in an ECS partition (or put it in standalone).
- Configure: configure the detector.

NOTE: while a detector is running or being configured, no actions can be executed.

19.5.4.3 CTP

The CTP subsection allows users to configure the CTP. At the end of the configuration the CTP processes will be restarted, therefore this configuration can only be performed when no runs are active.

NOTE: this subsection executes a global CTP configuration and is therefore different from the CTP partition configuration described in Section 19.5.4.1.

Part III

DDL and D-RORC software Reference Manual

November 2010

ALICE DAQ Project

DDL and D-RORC

20

DDL and D-RORC stand-alone software

This chapter describes several software tools that allow using the RORC device in a stand-alone manner.

20.1 Introduction

The DATE kit provides the readout software to perform long-term high-volume data taking with several RORC devices in an LDC (see Chapter 6 and Chapter 7). Some software is also provided to use the RORC device in a stand-alone manner, which is useful to facilitate the installation procedure, to help debugging in case of problems, and to exploit the supplementary features of the RORC as a test device for DATE. This stand-alone software covers four areas:

- The various test programs for the DDL and the RORC allow the user to identify the DDL and RORC components, to reset them, to check their status, and to execute a simple data taking task. The most important utility programs are described in Section 20.2.
- The ***Front-end Control and Configuration*** (***FeC2***) interpreter program allows the user to utilize the "backward" channel of the RORC device to send commands and data blocks to the front-end electronics. The short description of this program along with the review of the ***FeC2*** script language is described in Section 20.3.
- The DDL ***Data Generator*** (***ddg***) program allows the user to operate the RORC as a device to generate simulated event fragments as they would be produced from some front-end electronics. The handling of this program is described in Section 20.4. The DDG software is used for testing the DATE system.
- All functionalities of the of the stand-alone software are available as C API as well. One application is to call these C routines to configure and control the front-end electronics instead of using ***FeC2*** scripts. This might be the better choice when aiming at more complex and interactive software for testing detector electronics. The C API documentation is available in Chapter 21.

The precondition to run this stand-alone software is a loaded ***physmem*** and ***rorc_driver*** kernel modules. The installation procedure is explained in Section 20.5. When the DDL and RORC software is installed via the DATE kit, all the programs and scripts are located in the directories ***/date/rorc/Linux*** and ***/date/physmem/Linux***.

If several RORC devices are in place, the DATE readout program and the stand-alone programs can run simultaneously on different channels. This is possible because two ***physmem*** devices (see Chapter 15) are used: DATE memory banks access this memory via the ***/dev/physmem1*** device, and the stand-alone RORC programs via the ***/dev/physmem0*** device. Some programs have an option to access ***/dev/physmem1*** device as well; this option can be used only when DATE is not running.

The different parts of DDL (RORC, DIU, SIU) are described in Chapter 7. Further documentation the can be found at the Web site ***http://cern.ch/ddl/ddl_docu.html***. Besides, each program prints a short explanation of usage via the `-h` or `--help` options.

20.2 Test programs for the RORC, DIU and SIU

In this chapter the most important test stand alone programs are presented. A very detailed description of all test programs can be found in RORC Library User's Manual at ***http://cern.ch/ddl/rorc_docu.html*** page.

rorc_find and rorc_qfind

Synopsis

`rorc_find`

`rorc_qfind`

Parameters: none

Description

The ***rorc_find*** and ***rorc_qfind*** programs list the type and hardware identification of the RORC cards plugged in the machine and of the DIUs, either plugged on or integrated in the RORCs. The `rorc_find` program tries to open all the RORC devices and reads from their configuration EPROM the hardware identification. If one RORC channel is in use, it can not be opened, so its feature will not be listed. The ***rorc_qfind*** program reads the `/proc/rorc_map` process file prepared by the `rorc_driver` in boot time. It shows all RORC channels and the process id's as well if a channel is in use.

The type of the RORC device could be pRORC (PCI revision: 1), D-RORC (PCI rev: 2), integrated D-RORC (PCI rev: 3), D-RORC version 2 (PCI-X version, rev: 4) and PCI Express D-RORC (PCI rev: 5).

Examples

Listing 20.1 Example of rorc_find program

```
> rorc_find
The following device(s) found:
-------------------------------------------------------------------------
 Minor Channel Device type and HW identification
-------------------------------------------------------------------------
   0       0   integ. DRORC2   DRORC2 2v1 INT. LD: EP2S30  S/N: 03034
               embedded DIU
           1   embedded DIU
   1       0   integ. DRORC2   DRORC2 2v1 INT. LD: EP2S30  S/N: 04021
               embedded DIU
           1   embedded DIU
-------------------------------------------------------------------------
 4 RORC channel(s) not in use was found.
 RORC driver reported 2 RORC device(s).
```

Listing 20.2 Example of rorc_qfind program

```
> rorc_qfind
 The following device(s) found:
 ------------------------------------------------------
 Minor PCI_rev Com/Status Speed Hw_s/n Fw_ID PID_0 PID_1
 ------------------------------------------------------
   0      4    0x04100147  100   03034  2.12     0     0
   1      4    0x04100147  100   04021  2.12     0     0
 ------------------------------------------------------
 2 RORC device(s) with 4 channel(s) was found.
```

rorc_reset

Synopsis `rorc_reset [-{M|m} <RORC_minor>] [-{C|c} <DDL_channel>] [-D|d|B|b|S|s|F|f|O|o|E|e|N|n]`

Description The ***rorc_reset*** program initializes a RORC and / or a DDL channel. Depending on the given program options, the program resets the different parts of the RORC, the DIU or the SIU. Resetting the RORC means emptying all its FIFOs, clearing all error bits, and putting all programmable features to their reset value. Resetting the

DIU or the SIU means cutting the DDL link and putting all programmable features to their reset value; afterwards the DDL link automatically re-establishes itself.

Parameters and switches:

- the parameter *RORC_minor* defines the minor device number of the RORC in case there are several cards. The associated device file is ***/dev/prorcN***, where *N* is the minor device number starting from *0*. The default minor device number is 0.
- the parameter *DDL_channel* chooses the channel (0 or 1) in case of an integrated D-RORC. The default channel is 0.
- the switch `-D` or `-d` resets the DIU.
- the switch `-B` or `-b` resets both the RORC and the DIU.
- the switch `-S` or `-s` resets the SIU.
- the switch `-F` or `-f` clears the `rorcFreeFifo`.
- the switch `-O` or `-o` clears the other FIFOs of the RORC.
- the switch `-E` or `-e` clears the error bits of the RORC.
- the switch `-N` or `-n` clears the byte counters of the RORC.
- in case no reset switch is given, only the RORC is reset.

Before exiting the program writes “RORC reset OK”. It means the requested reset command is sent, but the success of the command is not checked. The user can test how successful the reset was by calling the ***rorc_status, diu_status,*** or ***siu_status*** program.

rorc_id, diu_id, siu_id

Synopsis

```
rorc_id [-{M|m} <RORC minor>] [-{C|c} <DDL channel>]
[-V <major version> -v <minor version>
-{P|p} <PLD version> -{S|s} <serial#> -{N|n} <channels>]
[-{D|d}] [-{T|t} <time-out>]


diu_id [-{M|m} <RORC_minor>] [-{C|c} <DDL_channel>]
[-V <major version> -v <minor version>
-P <PLD version> -B <speed version> -S <serial#>]


siu_id [-{M|m} <RORC_minor>] [-{C|c} <DDL_channel>]
[-V <major version> -v <minor version>
-P <PLD version> -B <speed version> -S <serial#>]
```

Description

The ***rorc_id*** program reads and displays the type, the hardware and the firmware identification of the RORC. It can display the DIU or SIU firmware identification as well. Finally the program informs whether the program library version and the RORC firmware are compatible.

The type of the RORC is written as “RORC revision number”. It is a number read from the PCI configuration space. If its value is 1, the device is a pRORC, if 2 then the device is a D-RORC having one DDL channel, if 3 then the device is a dual channel integrated D-RORC, if 4 then the device is a version 2 D-RORC (PCI-X version), and if 5 then the device is a PCI Express D-RORC.

The hardware identification word of the RORC contains the hardware release date and version number. The firmware identification word of the RORC contains the firmware release date, the firmware version, and the size of the `rorcFreeFifo`. The DIU or SIU firmware identification words contain the firmware release date and version number.

To get the DIU or SIU firmware identification the program sends a command. The SIU firmware identification can be asked only if the link is up. After waiting as many microseconds for the answer as specified in *`time-out`* parameter, the program interprets and displays the reply.

The ***`rorc_id`*** program can be used for writing the RORC hardware identification into the RORC with the help of `-V`, `-v`, `-P`, `-S` and `-N` switches. This feature is intended for RORC developers only. For writing the hardware identification a special resistor must be soldered. If this resistor is soldered out, the hardware identification cannot be changed. Because of this the usage of these switches will not be explained here.

The ***`diu_id`*** program reads and displays the hardware identification words of the DIU. The DIU hardware identification word contains the card major and minor version numbers (e.g. 2.0), the PLD version code (e.g. 20K60E), the card speed version (e.g. 2125 Mbps), and the card serial number. If the major version number is 1 then the card is a prototype (old) DDL card, if it is 2 then the card is the final (new) card.

Note: For embedded DIUs, i.e. when the DIUs are integrated onto the D-RORC card, the DIU does not have separated hardware identification. It is identified as a one channel of the RORC card.

The ***`siu_id`*** program reads and displays the hardware identification words of the SIU. The SIU hardware identification word contains the card major and minor version numbers (e.g. 2.0), the PLD version code (e.g. 20K60E), the card speed version (e.g. 2125 Mbps), and the card serial number. If the major version number is 1 then the card is a prototype (old) DDL card, if it is 2 then the card is the final (new) card. The SIU hardware identification words can be read only if the link is up.

The ***`diu_id`*** and ***`siu_id`*** programs can be used for writing this information into the DIU’s or SIU’s memory as well. This feature is made for DDL developers only. For writing the hardware identification a special resistor must be soldered in the card. If this resistor is soldered out, the hardware identification cannot be changed. Because of this the usage of `-V`, `-v`, `-P`, `-B` and `-S` switches will not be explained here.

Parameters and switches:

- the parameter *`RORC_minor`* defines the minor device number of the RORC in case there are several cards. The associated device file is ***`/dev/prorcN`***, where *`N`* is the minor device number starting from *`0`*. The default minor device number is 0.
- the parameter *`DDL_channel`* chooses the channel (0 or 1) in case of an integrated D-RORC. The default channel is 0.

- the switch `-D` or `-d` displays in addition the hardware and firmware identifications of the DIU and SIU.
- the parameter *`time-out`* defines the waiting time in microseconds for the DIU and SIU responds. The default value is 1000 microseconds.

rorc_status, diu_status, siu_status

Synopsis

```
rorc_status [-{M|m} <RORC_minor>] [-{C|c} <DDL_channel>]

diu_status [-{M|m} <RORC_minor>] [-{C|c} <DDL_channel>]
 [-{T|t} <time-out>] [-{V|v} <diu_version>]

siu_status [-{M|m} <RORC_minor>] [-{C|c} <DDL_channel>]
[-{T|t} <time-out>] [-{V|v} <diu_version>]
```

Description

The ***`rorc_status`*** program besides displaying the same information as *`rorc_id,`* reads the type of the RORC, the control/status and error registers and displays information about RORC status (e.g. working mode, `rorcFreeFifo` status, link status, flow control status) and errors.

The type of the RORC is written as “RORC revision number”. It is a number read from the PCI configuration space. If its value is 1, the device is a pRORC, if 2 then the device is a D-RORC having one DDL channel, if 3 then the device is a dual channel integrated D-RORC, if 4 then the device is a version 2 D-RORC (PCI-X version), and if 5 then the device is a PCI Express D-RORC.

The ***`diu_status`*** program sends a command to the DIU, waits for its reply and displays the DIU status. The program displays the hardware and firmware identifications of the DIU as well.

The ***`siu_status`*** program sends a command to the SIU, waits for its reply and displays the SIU status. The SIU receives the commands and replies only if the link is up. The program displays the SIU hardware and firmware identifications as well.

Parameters and switches:

- the parameter *`RORC_minor`* defines the minor device number of the RORC in case there are several cards. The associated device file is ***`/dev/prorcN`***, where *`N`* is the minor device number starting from *`0`*. The default minor device number is `0`.
- the parameter *`DDL_channel`* chooses the channel (0 or 1) in case of an integrated D-RORC. The default channel is 0.
- the parameter *`time-out`* defines the waiting time in microseconds for the DIU responds. The default value is 1000 microseconds.
- the parameter *`diu_version`* chooses the version of the DIU. The value can be 1 for the prototype version, or 2 for the final version (plugged or embedded). The default value is 2.

rorc_receive

Synopsis

```
rorc_receive
[[{-M|-m|--minor} <RORC_minor>]
|{-r|--revision} <revision> {-n|--serial} <serial>]
[{-C|-c|--channel} <DDL_channel>]
[-v|--verbose]
[{-G|-g|--generator} <loop-back mode>]
[-D|-d|--no_scatter]
[{-R|--reset_lev} <reset_level>]
[{-X|-x|--check} <check_level>]
[-Y|-y|--DDL_header]
[-Z|-z|--no_RDYRX]
[{-P|--phys_minor} <physmem_minor>]
[{-B|-b|--page} <page_length>]
[{-U|-u|--physmem} <useable_memory>]
[{-O|-o|--offset} <memory_offset>]
[{-E|-e|--events} <events>]
[{-I|-i|--init_word} <init_word>]
[{-p|--pattern} {c|a|0|1|i|d|<mif_file_name>}]
[{-S|-s|--stat_file} <stat file>]
[{-L|-l|--length} <data_length>]
[{-J|-j|--rand_len} <random_seed>]
[{-N|-n|--init_count} <initial_count>]
[{-F|--max_fifo} <max_FIFO>]
[{-f|--min_fifo} <min_FIFO>]
[{-T|--sleep_time} <sleep_time>]
[{-t|--load_sleep} <load_sleep_time]
[{-W|-w|--resp_wait} <wait_time>]
[{-Q|--byte_print} <GBs_to_print>]
[{-q|--page_print} <pages_to_print>]
[{-K|--output_file} <output_file>]
[{-k|--binary_output} <binary_output_file>]
[{-A|-a|--front_end} <FEE_address>]
```

Description

The ***rorc_receive*** program receives fragments from the DDL link or the internal data generator of the RORC. It uses the ***physmem*** package for allocating the memory blocks where the data is stored. The program compares word by word the received data with its expected value. It also checks whether the fragment length in the DTSW word matches the actual fragment length. This program provides a functional test of the RORC and DDL hardware / software.

The ***rorc_receive*** program can be executed for several RORCs in parallel. In this situation a distinct region of the ***physmem*** memory must be assigned to each running ***rorc_receive*** process. The assignment of physmem memory can be done via program options (with the switches `-U` and `-O`).

Parameters and switches:

- the parameter *RORC_minor* defines the minor device number of the RORC in case there are several cards. The associated device file is ***/dev/prorcN***, where *N* is the minor device number starting from *0*. The default minor device number is 0.

- the parameter *revision* is the RORC's PCI revision number. It must be < 6.
- the parameter *serial* is the RORC's hw serial number. If given, the RORC is identified by the *revision* and *serial* not by the parameter RORC minor.
- the parameter *DDL_channel* chooses the channel (0 or 1) in case of an integrated D-RORC. The default channel is 0.
- the switch -v prints details for debugging (verbose mode). Note that the switch -V is not implemented.
- the switch -G or -g enables the internal data generator of the RORC and the parameter loop-back mode selects the in loop-back location. The accepted values are:
  0: do not loop-back, thus sent data via the link.
  1: set the loop-back inside the DIU.
  2: set the loop-back inside the SIU.
  any other value: set the loop-back inside the RORC.
  The default value for the parameter loop-back mode is 0.
- the switch -D or -d enforces that the received fragments are not scattered in the ***physmem*** memory. Every event page will be written on the same physical address, hence pages will be overwritten by each other.
- the parameter *reset_level* defines which RORC and DDL elements are reset. The accepted values are:
  0: do not reset the RORC, neither the DIU nor the SIU.
  1: reset the RORC only.
  2: reset the RORC and the DIU, but not the SIU
  3: reset the RORC, the DIU and the SIU before collecting data.
  The default value for the parameter reset level is 3.
- the parameter *check_level* defines which parts of the received fragment are checked. The pattern for checking is given by the parameter *pattern*. The accepted values are:
  -1: do not stop if DTSTW problem occurs and do not check the fragment
  0: do not check the fragment
  1: check the first word of the fragment
  2: check the fragment, expect the first word
  3: check the whole fragment
  The default value for the parameter *check_level* is 3.
- the switch -Y defines that the received fragment contains the Common Data Header (see Section 3.9). The contents of this header is not checked.
- the switch -Z prevents sending the RDYRX and EOBTR commands. This switch is implicitly set when the -G or -A switch is used.
- the parameter *physmem_minor* defines the minor device number of the physmem memory device. It can be 0 for ***/dev/physmem0*** device, or 1 for ***/dev/physmem1***. The latter can be used only when DATE is not running in the same RORC channel. The default physmem device is ***/dev/physmem0***.
- the parameter *page_length* defines the size in bytes of the memory blocks (data pages). The default page size is 4096 bytes.
- the parameter *usable_memory* defines the amount in MB of the memory requested from the ***/dev/physmemN*** (N = 0 or 1) device. The default amount is 30 MB.
- the parameter *memory_offset* defines the offset in MB of the memory

requested from the ***/dev/physmemN*** (N = 0 or 1) device relative to its base address. The default offset is 0 MB.

- the parameter *events* defines the number of fragments to be read, or to be generated if the switch -G is used. The value 0 specifies an unlimited number of fragments, which is also the default value.
- the parameter *init_word* sets the second word of each fragment when the switch -G or -g is used. The default value for this parameter is 0. The first word of each fragment is an incrementing counter value starting from 1.
- the switch -p or --pattern sets the data pattern of each fragment when the switch -G or -g is used. The accepted possibilities are:
  'c': use constant data given by parameter init word.
  'a': use alternating data.
  '0': use flying 0 data starting from 0xfffffffe.
  '1': use flying 1data starting from 0x00000001.
  'i': use incremental data starting with parameter init word.
  'd': use decremental data starting with parameter init word.
  mif file name: Memory Initialization File (.mif). It can define a complete fragment. For the MIF description see e.g.
  http://www.mil.ufl.edu/4712/docs/mif_help.pdf
  The default character for the pattern is 'i'.
- the parameter *stat_file* defines the name of the file where the number of bytes transferred is written. If given, and the file already exist, the program adds the number of transferred bytes to the value already in the file.
- the parameter *data_length* defines either the size of the expected largest fragment, or the maximum size of the generated fragment by the internal data generator of the RORC when the switch -G is set. The size is given in words (4 bytes). The default size is 524287 words.
- the parameter *random_seed* defines the seed value for the generation of fragments of random length by the internal data generator of the RORC when the switch -G is set. If the parameter is set, the minimum length is 1 and the maximum length is the parameter data length rounded down to the nearest integer of power of 2. Using 0 as parameter value, the fragments have constant size. This is also the default value.
- the parameter *initial_count* defines the event count (first data word) of the first fragment. The default value for this parameter is 1.
- the parameter *max_FIFO* controls the filling of the rorcFreeFifo. It defines the maximal number of entries, hence it stops filling if the number of entries is reaches this value. The maximal value is 128, which is also the default value.
- the parameter *min_FIFO* controls the filling of the rorcFreeFifo. It defines the minimal number of entries, hence it starts filling if the number of entries is lower then this value. The default value for this parameter is 127.
- the parameter *sleep_time* defines the waiting period in milliseconds after each received fragment in order to simulate a loaded LDC. The default period is 0 milliseconds.
- the parameter *load_sleep_*time defines the waiting period in milliseconds before each time new physmem address is loaded into the RORC's Ready FIFO. The default period is 0 milliseconds.
- the parameter *wait_time* defines the waiting period for command responses

in microseconds. The default period is 1000 microseconds.

- the parameter *GBs_to_print* specifies that whenever this number of received data is transferred, the total number of received GB is printed. The default value is 1 GB.
- the parameter *pages_to_print* specifies that whenever this number of received pages is transferred, the total number of received pages is printed. The default value is 0, hence no page printout.
- the parameter *output_file* defines the name of the file where the received fragments are dumped as a text file. Each fragment starts with comment lines (indicated by the '#' character) that contains the event number (i.e. event fragment number) followed by the block number and the block length in 4-byte words, and then follow lines where each one shows a data word in hexadecimal format. This sequence repeats for each block and event fragments. The data words are not checked, hence this parameter implies the switch `-x 0`.
- the parameter *binary_output_file* defines the name of the file where the received fragments are dumped in binary format. The format of the fragments is the same as that of the text file (see the previous parameter). The comment lines are dumped as ASCII (together with the new line characters), while the data is dumped in binary. So the binary file is much smaller than the text one. At the same time in hexadecimal dump the event fragments can be easily found. The data words are not checked, hence this parameter implies the switch `-x 0`.
- the parameter *FEE_address* enforces that the data reading is carried out with the Start Block Read (STBRD) command. It defines the front-end address which is part for the STBRD command. This parameter is mandatory when the `-A` or `-a` switch is set.

Examples

```
> rorc_receive -m 1 -c 0 -g 3 -p i -l 1000 -x 3
```

Uses the internal data generator of the RORC with minor device number 1 and channel 0. The generated incremental data words have a fixed size of 1000 words, whereas the whole fragment is checked.

```
> rorc_receive -m 2 -c 1 -o 60 -z -r 2 -K /tmp/data.raw
```

Reads fragments from channel 1 of the dual channel D-RORC with minor device number 2. The RORC and the DIU are reset, but not the SIU. The RDYRX is not sent. The data words are dumped to the file ***/tmp/data.raw*** without checks. The ***physmem*** memory is utilized between 60 MB and 90 MB relative to its base address.

20.3 Front-end Control and Configuration (FeC2) program

20.3.1 General description of the FeC2 program

FeC2

Synopsis

```
FeC2 [-{M|m} <Rorc_minor> |
-{R|r} <revision> -{N|n} <serial>]
[-{C|c} <DDL_channel>] [-{P|p} <phys_minor>]
[-{F|f} <FeC2_script_file>] [-{L|l} <log_file>]
[-{O|o} <mem_offset>] [-{U|u} <mem_size>]
[-{T|t} <DDL_timeout>] [-S|-s] [-v] [-H|-h]
```

Description

The ***FeC2*** program can be used for controlling and configuring the Front-end Electronics (FEE) via the DDL. It downloads commands and data blocks to the FEE, and it reads status and data blocks from the FEE. The user needs to write a script file following the FeC2 syntax.

Parameters and switches:

- the parameter *RORC_minor* defines the minor device number of the RORC in case there are several cards. The associated device file is ***/dev/prorcN***, where *N* is the minor device number starting from *0*. The default minor device number is *0*.
- the parameter *revision* is the RORC's revision number. It must be < 6.
- the parameter *serial* is the RORC's hardware serial number. If given, the RORC is identified by the *revision* and *serial*, not by the parameter *RORC_minor*.
- the parameter *DDL_channel* chooses the channel (0 or 1) in case of an integrated D-RORC. The default channel is 0.
- the parameter *physmem_minor* defines the minor device number of the physmem memory device. It can be 0 for ***/dev/physmem0*** device, or 1 for ***/dev/physmem1***. The latter can be used only when DATE is not running in the same RORC channel. The default physmem device is ***/dev/physmem0***.
- the parameter *FeC2_script_file* defines the name of the script file to be interpreted. The default name is ***FeC2.scr***. The syntax of FeC2 script files is described in Section 20.3.2.
- the parameter *log_file* defines the name of the log file. If not given, the standard output (stdout) stream is used.
- the parameter *mem_offset* defines the offset in MB of the memory requested from the ***/dev/physmem0*** device relative to its begin. The default offset depends on the parameters RORC minor and DDL channel in the following way: `mem offset = (RORC_minor * 2 + DDL_channel) * 8`. Hence for each channel a separate block of 8 MB ***physmem*** memory is assigned, which allows to run several FeC2 scripts in parallel on the same machine.

- the parameter *mem_size* defines the size in MB of the memory requested from the ***/dev/physmem0*** device. The default amount is 8 MB in accordance with the above scheme of the default value for the parameter *mem_offset*.
- the parameter *DDL_timeout* defines the waiting time in microseconds for the DDL commands. The default value is 1000 microseconds.
- the switch `-S` or `-s` enables the use of shared memory to accelerate the download of data blocks that have been written beforehand into files. When this switch set, each file is stored into shared memory, so that the next time the file will be used, it will be retrieved from memory. The usage of the shared memory is as follows:
  - Each DDL channel has its own shared memory segments.
  - The maximum number of channels per LDC is 16.
  - The maximum number of files per channel is 15420.
  - The maximum length of a file name is 255 characters.
  - The maximum number of shared memory segments per channel is 127.
  - Each shared memory segment can host 4 MB minus 8 bytes for administration.

  The following utility program can be used to remove the shared memory segments, where the switch `-x` just scans them:

  ```
  clean_shm [-m <RORC_minor>] [-c <DDL_channel>] [-x]
  ```
- the switch `-v` prints details for debugging (verbose mode). Note that the switch `-V` is not implemented.
- the switch `-H` or `-h` prints a short help message.

20.3.2 Syntax of script files for the FeC2 program

The instruction and its parameters can be separated by space(s) or tabulator(s). Any parameter can be an environment variable. In this case the name must start with a $ character. Each instruction should be written in one line. Any number of empty lines is allowed. Lines starting with a '#', '*' or ';' character are considered as comment. After the character ';' or '//' the remaining part of any line is considered as in-line comment. Comment lines, in-line comments and empty lines can be used in data files as well. All instructions will be executed sequentially up the end of the script file, or until reaching a return/stop command, or till the occurrence of an error.

20.3.2.1 FeC2 instructions related to the DDL

---

reset

Synopsis `reset [RORC | DIU | SIU]`

Description Reset the given element of the DDL link. If no parameter is given, then the RORC is reset.

read_DDL_status

Synopsis `read_DDL_status`

Description Reads and prints the DIU and SIU status. This command is executed only if the `-v` option is switched on.

write_RDYRX

Synopsis `write_RDYRX`

Description Send an RDYRX command to the FEE.

write_EOBTR

Synopsis `write_EOBTR`

Description Send an EOBTR command to the FEE.

write_command

Synopsis `write_command <command_code>`

Description Send a DDL command to the FEE.

Parameter:
- the parameter *`command_code`* is a hexadecimal number (maximum 19 bits).

write_block

Synopsis `write_block <address> <file_name> [<format>]`

Description First send the address to the FEE, and then send the block of data to the FEE.

Parameters:
- the parameter *`address`* is a hexadecimal number (maximum 19 bits) within the address space of the FEE to which the data block is sent.
- the parameter *`file_name`* is the name of the file where the block of data is stored. The maximum length of this file is (2^19 -1 = 524287) words.
- the parameter *`format`* specifies in C style format (e.g. "`%x`") the reading mode of the words from the file. If omitted, the binary mode is used.

write_block_multiple

Synopsis `write_block_multiple <poll_address> <status> <mask> <time-out> <FEE_address> <block_size> <file_name> [<format>]`

Description First read the data from the file called *file_name* and divide it into sub-blocks of *block_size* words length. For each sub-block send the incremented address to the FEE followed by the data, thus the first sub-block goes to *FEE_address*, the second sub-block to *FEE_address + block_size*, the third sub-block to *FEE_address + 2 * block_size*, and so forth. At the end of each sub-block send a status read request to the *poll address* and compare the reply (after applying *mask* as bitwise AND operation) with the value *status*. Repeat the status read request until an exact match happens or the *time-out* is expired. In the latter case stop looping and set the "check_fail" flag (see Section 20.3.2.2). The length of the file needs to correspond with the length expected for the given FEE address. The maximum length allowed is 2^19 -1 = 524287 words.

Parameters:

- the parameter *poll_address* is a hexadecimal number (maximum 19 bits) within the address space of the FEE to which the status read request command is sent.
- the parameter *status i*s the compare value (maximum 19 bits) for the check.
- the parameter *mask* is applied as bitwise AND operation to the received value from the FEE before the comparison against the parameter *status i*s done.
- the parameter *time-out* defines the maximum duration in microseconds to repeat the polling operation.
- the parameter *FEE_address* is a hexadecimal number (maximum 19 bits) within the address space of the FEE to which the data block is sent.
- the parameter *block_size* is the number of words of sub-blocks to be sent before the next status check.
- the parameter *file_name* is the name of the file where the block of data is stored. The maximum length of this file is (2^19 -1 = 524287) words.
- the parameter *format* specifies in C style format (e.g. "%x") the reading mode of the words from the file. If omitted, the binary mode is used.

read_and_print

Synopsis `read_and_print <address> <format> [<stream>]`

Description Send a command to the FEE and print the received value.

Parameters:

- the parameter *address* is a hexadecimal number (maximum 19 bits) within the address space of the FEE to which the status read request command is sent.
- the parameter *format* specifies in C style format (e.g. "%x") the printing mode of the received value.
- the parameter *stream* defines the file name where to append the output. If omitted, the parameter *log_file* from the FeC2 calling sequence is used,

otherwise the standard output is used.

read_and_check

Synopsis `read_and_check <address> <status> <mask>`

Description Send a command to the FEE and check the received value. If the check fails, the "check_fail" flag is set (see Section 20.3.2.2).

Parameters:

- the parameter *address* is a hexadecimal number (maximum 19 bits) within the address space of the FEE to which the status read request command is sent.
- the parameter *status* is the compare value (maximum 19 bits) for the check.
- the parameter *mask* is applied as bitwise AND operation to the received value from the FEE before doing the comparison with the parameter *status*.

read_until

Synopsis `read_until <address> <status> <mask> <time-out>`

Description Send a command to the FEE and check the received value. This polling operation is repeated until the check is successful or the time-out is reached. In the latter case, the "check_fail" flag is set (see Section 20.3.2.2).

Parameters:

- the parameter *address* is a hexadecimal number (maximum 19 bits) within the address space of the FEE to which the status read request command is sent.
- the parameter *status* is the compare value (maximum 19 bits) for the check.
- the parameter *mask* is applied as bitwise AND operation to the received value from the FEE before doing the comparison with the parameter *status*.
- the parameter *time-out* defines the maximum duration in microseconds before repeating the polling operation.

read_block

Synopsis `read_block <address> <file_name> [<format>]`

Description First send the address to the FEE, and then read the block of data from the FEE. The received words are written to the file. The length of the block of data is under the control of the FEE.

Parameters:

- the parameter *address* is a hexadecimal number (maximum 19 bits) within the address space of the FEE from where the block of data is read.
- the parameter *file_name* is the name of the file where the received words are written.

- the parameter *format* specifies in C style format (e.g. "%x") the writing mode of the words to the file. If omitted, the binary mode is used.

read_and_check_block

Synopsis read_and_check_block <address> <file name> [<format>]

Description First send the address to the FEE, and then read the block of data from the FEE. The received words are compared with the ones in the file. The length of the block of data is under the control of the FEE. If the check fails, the "check_fail" flag is set (see Section 20.3.2.2)

Parameters:
- the parameter address is a hexadecimal number (maximum 19 bits) within the address space of the FEE from where the block of data is read.
- the parameter file name is the name of the file which contains the words for comparison.
- the parameter format specifies in C style format (e.g. "%x") the reading mode of the words from the file. If omitted, the binary mode is used.

20.3.2.2 FeC2 instructions related to the program flow

define

Synopsis define <name> <value>

Description Whenever *name* occurs as parameter of an FeC2 instruction, the *value* is used instead. The definition of *name* must appear before its first use. To distinguish between *name* and numbers, the *name* must start with a letter, whereas a hexadecimal constants must start with 0x.

loop_on, loop_off

Synopsis
```
loop_on <loop_number> <loop_uswait> <loop_cont_if_error>
FeC2 instruction(s)
loop_off
```

Description All FeC2 instructions (with some exception) which are between loop_on and loop_off will be repeated loop_number times. The following FeC2 instructions will not be repeated even if they are between loop_on and loop_off: reset, write_RDYRX, write_EOBTR, define, loop_on, loop_off, wait, call_file, return, stop_if_failed, stop. The loop_on can not be nested. A second call of loop_on overwrites the previous loop parameters.

Parameters:
- the parameter *loop_number* defines how many times the FeC2 instruction will

be repeated. If its value is less then 2 the command is equivalent to a `loop_off` command.

- the parameter *`loop_uswait`* defines how many microseconds to wait at the end of each loop.
- the parameter *`loop_cont_if_error`* specifies whether the loop should be interrupted if a check in the instruction inside the loop fails. If its value is 0, the looping continues. For any other values the program jumps out of the loop. This concerns only the FeC2 instructions `read_and_check`, `read_and_check_block`, and `read_until`.

Example

```
loop_on 2 0 0
instruction 1
instruction 2
instruction 3
loop_off
```

is equivalent with the following sequence:

```
instruction 1
instruction 1
instruction 2
instruction 2
instruction 3
instruction 3
```

wait

Synopsis `wait <usecs>`

Description The execution is suspended for a period of `usecs` microseconds.

call_file

Synopsis `call_file <file_name>`

Description The execution jumps to the FeC2 script file whose name is `file_name`. If this file is not found, the execution is stopped. Recursive calls are not allowed.

return

Synopsis `return`

Description The execution of the current FeC2 script is stopped and if possible the control is returned one level higher.

stop_if_failed

Synopsis `stop_if_failed [<exit_code>]`

Description The execution is stopped with the given *exit_code* (the default one is 1) if in the previous instruction the check has failed. This concerns the following FeC2 instructions: `read_and_check`, `read_and_check_block`, `read_until`, and `write_block_multiple`.

stop

Synopsis `stop [<exit_code>]`

Description The execution is stopped with the given *exit_code* (the default one is 0).

20.3.2.3 Example of an FeC2 script

Listing 20.3 shows a FeC2 script used to carry out some basic tests on the FEE. Some symbolic names are defined (lines 15-18) and the RORC as well as the SIU are reset (lines 20-21) at the beginning. A command is sent (line 23) to initialize the FEE and the effect is verified (lines 24-25). The status of some registers is read and copied into a file (line 27-28). Finally, a block of data is sent to the FEE (line 30) and read back (line 31) for the purpose of testing. If all tests are passed, the exit code of this script is 0 (line 34).

Listing 20.3 Example of an FeC2 script

```
13:  # FeC2 script
14:
15:  define PATGEN 0x0
16:  define EVLEN  0x100
17:  define STATUS status.out
18:  define PROBA  proba.hex
19:
20:  reset RORC
21:  reset SIU
22:
23:  write_command 0x10f
24:  read_until EVLEN 0x0f 0xff 10000000
25:  stop_if_failed -1
26:
27:  read_and_print PATGEN "Patgen status: "x%x" STATUS
28:  read_and_print EVLEN "Evlen status: 0x%x" STATUS
29:
30:  write_block 0x600 PROBA "%x"
31:  read_and_check_block 0x600 PROBA "%x"
32:  stop_if_failed -2
33:
34:  stop
```

20.4 DDL Data Generator (DDG) program

20.4.1 General description of the DDG program

ddg

Synopsis

```
ddg [-{F|f} <config_file>] [-{L|l} <log_file>]
[-{P|p} <physmem_minor>] [-{S|s} <SMI_object>
[-{T|t} <time-out>] [-{N|n}] [-v] [-H|-h]
```

Description

The `ddg` program is able to supply data (simulated events) to the dual channel D-RORC card, which acts as data generator for the DDL channels. The program reads the fragments from data files, which need to be generated in advance. It can handle up to 12 DDL channels per machine. Files cannot be shared between the channels.

The program can also generate the Common Data Header (see Section 3.9). If this header is enabled, then the event identifiers of all fragments will be synchronized. Several replica of the `ddg` program can run parallel on the same or on different machines. The CDH of the corresponding fragments (generated with different program replica) remain synchronized.

Parameters and switches:

- the parameter *`config_file`* defines the name of the configuration file, which contains all the parameters describing the fragments to be generated by the program (see Section 20.4.3). The default name is ***`ddg.conf`***.
- the parameter *`log_file`* defines the name of log file. If not given, the standard output (stdout) stream is used.
- the parameter *`physmem_minor`* defines the minor device number of the ***`physmem`*** memory device. It can be 0 for ***`/dev/physmem0`*** device, or 1 for ***`/dev/physmem1`***. The latter can be used only when DATE is not running in the same RORC channel. The default ***`physmem`*** device is ***`/dev/physmem0`***.
- the parameter *`SMI_object`* defines the associated SMI object in the form <domain_name>::<object_name>. The default SMI object is ***`DDG::DDG`***.
- the parameter *`time-out`* defines the waiting time in microseconds for the DDL commands. The default value is 1000 microseconds.
- the switch `-N` or `-n` enforces that the fragments are not scattered in the ***`physmem`*** memory. Only one buffer is used.
- the switch `-v` prints details for debugging (verbose mode). Note that the switch `-V` is not implemented.
- the switch `-H` or `-h` prints a short help message.

20.4.2 Behavior of the DDG program

The ddg program works in the following way:

1. The program parses the configuration file. Then it initializes the ***physmem*** memory, the DIM and the SMI packages. It sets the SMI state to “IDLE”, opens the requested DDL channels, and reads the fragments from the data files into its buffer. Alternatively it generates the fragments, if they are not read from a file.
2. The program waits for the RDYRX command from the DDL channels. It should receive them only when the DATE system at the receiving side has been started and is ready to receive data. If configured, the program resets the specified channels.
3. The program starts sending data upon reception of the SMI command “START”. It sets the SMI state to “RUNNING”.
4. While sending data the program pushes the fragment parameters (buffer’s physical address and length) into the RORC. It checks the status of the ROROC’s FIFOs. When a fragment is transmitted, the program reads the next from file to the memory buffer and pushes the fragment’s parameters into the RORC.
5. After receiving the SMI command “STOP”, the program stops sending data and terminates.

20.4.3 Syntax of the DDG configuration file

Any number of empty lines is allowed. Lines starting with a ‘#’, ‘*’ or ‘;’ character are considered as comment. After the character ‘;’ or ‘//’ the remaining part of any line is considered as in-line comment.

Each keyword must be written in a separate line. The keywords and their (optional) parameters can be separated by space(s), tabulator(s) or equal sign(s). The use of the keywords is not mandatory. Each keyword has a default value, which is used when the keyword is not specified in the DDG configuration file. The configuration file may even be empty, and in this case all the default values are used. If the same keyword occurs more than once in the configuration file, then the last value is used. This rule applies also for conflicting keywords.

20.4.3.1 Channel independent keywords

The following DDG configuration file keywords do not depend on the D-RORC channel.

PHYSMEM_OFFSET

Synopsis `PHYSMEM_OFFSET [<memory_offset>]`

Description The parameter *memory_offset* defines the offset of memory in MB requested from the ***/dev/physmem0*** device relative to its base address.The default offset is 0.

PHYSMEM_LENGTH

Synopsis `PHYSMEM_LENGTH [<useable_memory>]`

Description The parameter *usable memory* defines the amount of memory in MB requested from the ***/dev/physmem0*** device. The default amount is 32 MB.

DDL_COMMANDS

Synopsis `DDL_COMMANDS`

Description Wait for the RDYRX commands before sending data.

NO_DDL_COMMANDS

Synopsis `NO_DDL_COMMANDS`

Description Do not wait for the RDYRX commands before sending data. If neither the keyword DDL_COMMANDS nor the keyword NO_DDL_COMMANDS is present, then the configuration is done with the keyword NO_DDL_COMMANDS.

HEADER

Synopsis `HEADER`

Description Generate the Common Data Header (CDH) for each fragment. The configuration of the CDH is specified with a set of DDG keywords (see Section 20.4.3.3).

NOHEADER

Synopsis `NOHEADER`

Description Do not generate the Common Data Header (CDH) for the fragments. If neither the keyword HEADER nor the keyword NOHEADER is present, then the configuration is done with the keyword NOHEADER.

BUNCH_CROSSING_START

Synopsis `BUNCH_CROSSING_START <start_value>`

Description The parameter *start_value* is used to calculate the starting value for the orbit_number (24 bits) and the bunch_crossing_number (12 bits) for the CDH of the first fragment for all channels:

- orbit_number_first = *start_value* / 3564
- bunch_crossing_number_first = *start_value* % 3564

If not present, the parameter *start_value* is 1.

BUNCH_CROSSING_INCREMENT

Synopsis BUNCH_CROSSING_INCREMENT <min_value> <max_value>

Description The parameters *min_value* and *max_value* are used to calculate the range of the increment values for the bunch_crossing_number:

- bc_min_increment = 2^*min_value* - 1
- bc_max_increment = 2^*max_value* - 1

The range for these parameters is between 0 and 31. If not present, the value for the parameter min value is 0, and the value for the parameter max value is 20.

BUNCH_CROSSING_SEED

Synopsis BUNCH_CROSSING_SEED <seed_value>

Description The parameter *seed_value* is used to initialize the random number generator RANDOM for the calculation of the bunch_crossing_number and orbit_number for the CDH of the fragments (except the first one) for all channels:

- bc_increment = RANDOM (bc_min_increment, bc_max_increment)
- bunch_crossing += (bc_increment % 3564)
- orbit_number += (bc_increment / 3564)
- ```
  if (bunch_crossing >= 3564)
  {
    orbit_number++
    bunch_crossing %= 3564
  }
  ```

If not present, the value for the parameter seed value is 1.

MAX_EVENT

Synopsis MAX_EVENT <max_event_number>
  ```

Description The parameter *max_event_number* defines the maximum number of fragments per channel to be generated. No limitation is given when 0 is used, hence the DDG program terminates when the SMI command “STOP” is received. If not present, the value for the parameter *max_event_number* is 0.

20.4.3.2 Channel dependent keywords

The following DDG configuration file keywords depend on the D-RORC channel. The group of keywords that characterize one channel is introduced by the keyword RORC_CHANNEL.

RORC_CHANNEL

Synopsis `RORC_CHANNEL <minor> <channel>`

Description The parameter `minor` defines the minor device number of the RORC in case there are several cards. The default minor device number is 0. The parameter `channel` chooses the channel (0 or 1) of a dual channel D-RORC. The default channel is 0.

DATA_FILE

Synopsis `DATA_FILE <file name>`

Description The parameter `file name` defines the DDG data file (see Section 20.4.4). If this keyword is given, the data words of the generated fragments are supplied from this file.

DATA_PATTERN

Synopsis `DATA_PATTERN <pattern>`

Description The parameter `pattern` sets the data pattern for the generated fragments. The accepted characters of this parameter are the following:
- ‘`c`’: constant data pattern.
- ‘`a`’: alternating data pattern.
- ‘`0`’: flying 0 data pattern.
- ‘`1`’: flying 1data pattern.
- ‘`i`’: incremental data pattern.
- ‘`d`’: decremental data pattern.

The default data pattern is ‘`i`’. If both keywords DATA_FILE or DATA_PATTERN are present, the later one is used. If neither is present, then the configuration is done with the keyword DATA_PATTERN.

INIT_WORD

Synopsis `INIT_WORD <start value>`

Description The parameter `start value` in hexadecimal format sets the second data word of each generated fragment when the keyword DATA_PATTERN is present. The default value depends on the selected pattern:
- `0xfffffffe`: if the pattern is '`d`'.
- `0x00000001`: if the pattern is '`i`'.
- `0x0`: for the other patterns.

The first word of each generated fragment is an incrementing counter value starting from 1.

DATA_LENGTH

Synopsis `DATA_LENGTH <maximum length>`

Description The parameter `maximum length` defines the length in words (4 bytes) of the largest generated fragment when the keyword DATA_PATTERN is present. The range for this parameter is between 1 and 2^24-1. If not present, the value is 2^19-1 = 524287.

RANDOM

Synopsis `RANDOM`

Description Generate fragments with random length. Their minimal length is 0, and their maximum length is given either by the parameter of the keyword DATA_LENGTH or by the specified length of the fragments in a DDG data file (see Section 20.4.4).

NORANDOM

Synopsis `NORANDOM`

Description Generate fragments with constant length. If neither the keyword RANDOM nor the keyword NORANDOM is present, then the configuration is done with the keyword RANDOM.

RESET

Synopsis `RESET`

Description Reset the DDL channel before generating fragments.

NORESET

Synopsis `NORESET`

Description Do not reset the DDL channel before generating fragments. If neither the keyword RESET nor the keyword NORESET is present, then the configuration is done with the keyword RESET.

20.4.3.3 Common data header keywords

The following DDG configuration file keywords control the generation of the Command Data Header (CDH) when the keyword HEADER is present.

BLOCK_LENGTH

Synopsis `BLOCK_LENGTH`

Description Fill the “block length” field in the CDH with the length for each generated fragment.

NO_BLOCK_LENGTH

Synopsis `NO_BLOCK_LENGTH`

Description Fill the “block length” field in the CDH with the value `0xffffffff` for each generated fragment. If neither the keyword BLOCK_LENGTH nor the keyword NO_BLOCK_LENGTH is present, then the configuration is done with the keyword BLOCKLENGTH.

MINI_EVENT_ID

Synopsis `MINI_EVENT_ID`

Description Fill the “mini-event ID” field in the CDH with the `bunch crossing number` for each generated fragment.

NO_MINI_EVENT_ID

Synopsis `NO_MINI_EVENT_ID`

Description Fill the “mini-event ID” field in the CDH with the `bunch crossing number` with some random errors for each generated fragment. If neither the keyword MINI_EVENT_ID nor the keyword NO_MINI_EVENT_ID is present, then the configuration is done with the keyword MINI_EVENT_ID.

FORMAT_VERSION

Synopsis `FORMAT_VERSION <version>`

Description Fill the “format version” field in the CDH for each generated fragment with the parameter `version`. It is a hexadecimal number (8 bits). If not present, the version number 1 is used.

L1_TRIGGER

Synopsis `L1_TRIGGER <L1 trigger message>`

Description Fill the “L1 trigger message” field in the CDH for each generated fragment with the parameter `L1 trigger message`. It is a hexadecimal number (10 bits). If not present, the message is 0.

SUB_DETECTORS

Synopsis `SUB_DETECTORS <participating sub-detectors>`

Description Fill the “participating sub-detectors” field in the CDH for each generated fragment with the parameter `participating sub-detectors`. It is a hexadecimal number (24 bits). If not present, the value is 0.

ATTRIBUTES

Synopsis `ATTRIBUTES <block attributes>`

Description Fill the “block attributes” field in the CDH for each generated fragment with the parameter `block attributes`. It is a hexadecimal number (8 bits). If not present, the value is 0.

STATUS_BITS

Synopsis `STATUS_BITS <status and error bits>`

Description Fill the “status and error bits” field in the CDH for each generated fragment with the parameter `status and error bits`. It is a hexadecimal number (8 bits). If not present, the value is 0.

TRIGGER_CLASS_LOW

Synopsis `TRIGGER_CLASS_LOW <trigger class low bits>`

Description Fill the “trigger class low” field (bits 1-31 of the trigger class information) in the CDH for each generated fragment with the parameter `trigger class low bits`. It is a hexadecimal number (32 bits). If not present, the value is 0.

TRIGGER_CLASS_HIGH

Synopsis `TRIGGER_CLASS_HIGH <trigger class high bits>`

Description Fill the “trigger class high” field (bits 32-49 of the trigger class information) in the CDH for each generated fragment with the parameter `trigger class high bits`. It is a hexadecimal number (18 bits). If not present, the value is 0.

ROI_LOW

Synopsis `ROI_LOW <ROI low bits>`

Description Fill the “ROI low” field (bits 0-3 of the region of interest information) in the CDH for each generated fragment with the parameter `ROI low bits`. It is a hexadecimal number (4 bits). If not present, the value is 0.

ROI_HIGH

Synopsis `ROI_HIGH <ROI high bits>`

Description Fill the “ROI high” field (bits 4-35 of the region of interest information) in the CDH for each generated fragment with the parameter `ROI high bits`. It is a hexadecimal number (32 bits). If not present, the value is 0.

20.4.3.4 Example of a DDG configuration file

Listing 20.4 shows a DDG configuration file for used to generate fragments on channel 0 on a dual channel D-RORC with minor device number 1 (line 46). The ***physmem*** memory is utilized between its base address and 10 MB (lines 37-38). The generated fragments have an incremental data pattern of 1500 words of random length (lines 47-49). A fixed length can be easily achieved, e.g. by removing the

commenting semicolon (line 49). The CDH is part of the generated fragments (line 40) with a starting bunch crossing number of 1 (line 41) and a fixed increment of 2^10-1 for the bunch crossing and hence orbit number (line 42). The “mini-event ID” field is also set (line 51), but not the “block length” field (line 50). There is no limit on the number of fragments to be generated (line 43). The DDG program starts sending data only after receiving the RDYRD command (line 39).

Listing 20.4 Example of a DDG configuration file

```
35:  # DDG configuration file
36:
37:  PHYSMEM_OFFEST 0
38:  PHYSMEM_LENGTH 10
39:  DDL_COMMANDS
40:  HEADER
41:  BUCH_CROSSING_START 1
42:  BUNCH_CROSSING_INCREMENT 10 10
43:  MAX_EVENT 0
44:
45:  # DDG pcddl01ldc 405 channel 0
46:  RORC_CHANNEL 1 0
47:  DATA_PATTERN d
48:  DATA_LENGTH 1500
49:  ;NORANDOM
50:  NO_BLOCK_LENGTH
51:  MINI_EVENT_ID
```

20.4.4 Syntax of the DDG data files

Any number of empty lines is allowed. Lines starting with a ‘#’, ‘*’ or ‘;’ character are considered as comment. After the character ‘;’ or ‘//’ the remaining part of any line is considered as in-line comment.

The structure of the data files is as follows:

1. Put the maximum fragment size in words (4 bytes) in a separate line. It is a decimal number, which must be greater than 0 and less than 2^24-1 = 1677215.
2. Put the fragment size in words (4 bytes) of the first fragment in a separate line. It is a hexadecimal number, which must be greater or equal than 0 and less than 2^{24} -1 = 0xffffff.
3. Put the data words of the first fragment. They can be separated by space(s), tabulator(s) or new line character(s).
4. Continue with the following fragments as described in point 2. and 3.

20.5 Stand-alone installation

The DDL and RORC library and test programs are installed together with the DATE kit. For a stand-alone installation, follow the procedure below:

- The header, source, object and executable files of the RORC and DDL test programs and library are in the common AFS area:

  `/afs/cern.ch/alice/daq/ddl/rorc/`

This directory contains the different versions of the software as separate sub-directories. It also contains the different versions in compressed formats.

- The compressed file names show the version number and the time of archiving. Always use the latest date of a given version. The latest distributed version can be found on the following Web page:
  http://cern.ch/ddl/rorc_support.html
- Copy the compressed file on a local directory and uncompress it. Use the following command for extracting the files:
  `gtar -xvzf  rorc_vers.x.y.z_year.month.day.tgz  rorc/`
- All test programs use ***physmem***, which needs be previously installed on the machine (see Chapter 15).
- To do the compilation, type the following commands:
  ```
  cd rorc
  make -f Makefile clean
  make -f Makefile
  ```
- To compile the driver type
  `make -f Makefile driver`
- To create the device files, and prepare the driver to be loaded at boot time type as ***root*** the following commands
  `make -f Makefile dev`
- To load the ***rorc_driver*** kernel module without booting type as ***root***:
  `make -f Makefile load`
- In case an older version of the ***rorc_driver*** is already loaded, then type as ***root***:
  `make -f Makefile reload`
- To check if RORC card is plugged and the driver is loaded type
  `./check_driver.`
  This script shows if the RORC card is plugged (calling `/sbin/lspci`), if the driver is loaded (calling `/sbin/lsmod`) and the driver messages during load time (calling `dmesg`).

21

RORC Application Library

This chapter describes the API library that allows to develop programs using the RORC device in a stand-alone manner.

21.1 Introduction

The DATE kit provides the readout software to perform long-term high-volume data taking with several RORC devices in a LDC (see Chapter 6 and Chapter 7). Some software is also provided to use the RORC device in a stand-alone manner (see Chapter 20), useful to facilitate the installation procedure, to help debugging and to use the features of the RORC as a test device for DATE. However, if someone wants to develop his own special program to exploit the capabilities of the RORC device then an application library (written in C) provided with the DATE kit can be used. This chapter provides a description of the most important routines of this library.

21.2 Header files

Before calling any of the following routines, the user must include a header file:

```
#include "rorc_ddl.h"
```

This file contains the necessary definitions for the use of the DDL. It has a reference to another header file, which contains the definitions of the RORC cards:

```
#include "rorc_lib.h"
```

The header files contain the type definition of the structures referred to in further descriptions. In addition, they contain the definition of the macros described later on.

21.3 The rorc_driver

The programs and routines described in this documents work under the LINUX operating system. Currently we have a RORC driver for CERN Scientific LINUX version 4 (SLC4, kernel version 2.6.9) and SLC5 (kernel version 2.6.18). Before using the described routines and programs, the RORC driver must be loaded (see Section 21.5 for details).

21.4 Description of the routines and functions

rorcFindAll

Synopsis

```
#include <rorc_lib.h>

int rorcFindAll(rorcHwSerial_t    *hw,
                rorcHwSerial_t    *diu_hw,
                rorcChannelId_t   *channel,
                int               *rorc_revision,
                int               *diu_vers,
                int               max_dev)
```

Description

Find all RORC channels not in use. The ***rorcFindAll()*** routine returns the version, serial, revision, minor and channel numbers of all RORC cards plugged in the PC together with the same information regarding the plugged in or embedded DIUs. The routine tries to open all RORC devices and reads the hardware version and serial numbers from their configuration EPROM. It also sends a DDL command to the DIU to find out the DIU version and serial number.

The routine needs to open the RORC channel to read the above data. If the RORC channel is used by some other program then it cannot be opened and it will be not included into the list of “RORC channels found”. To get the list of all RORC devices, independently of its occupancy, use ***rorcQuickFind()***.

Parameters

hw pointer to an array of ***rorcHwSerial_t*** type structures. The routine loads into these structures the version and serial numbers of RORC cards found. ***rorcHwSerial_t*** is defined in the header file ***rorc_lib.h***. Besides the major and minor version and the serial numbers it contains the full string found in the configuration EPROM of the RORC card. If there is no information in the EPROM about the hardware version and serial numbers, then the routine puts –1 into the structure as version and serial numbers.

diu_hw pointer to an array of ***rorcHwSerial_t*** type structures. The routine loads into these structures the version and serial numbers of the DIU card found. If no DIU is plugged or there is no information in the DIU’s EPROM about the hardware version and serial numbers, then the routine puts –1 into the structure as version and serial numbers.

channel pointer to an array of ***rorcChannel_t*** structures. Here the routine supplies the corresponding minor device numbers of the RORC cards and the channel numbers of the DIUs.

rorc_revision pointer to an array of integers. Here the routine supplies the corresponding device revision numbers (1 for pRORC, 2 for D-RORC with connector for DIU. 3 for D-RORC with

embedded DIUs, 4 for version 2 D-RORC and 5 for PCI express RORC) of the RORC cards.

diu_vers pointer to an array of integers. Here the routine supplies the corresponding DIU version number (0 if no DIU, 1 if prototype DIU, 2 if final DIU plugged in version, and 3 if embedded DIU) of the DIU on the corresponding DDL channel.

max_dev the size of *hw*, *diu_hw*, *channel*, *rorc_revision* and *diu_vers* arrays.

Return value number of DDL channels (not in use) found or 0.

See also ***rorcFind()*, *rorcQuickFind()*, *rorcSerial()*, *rorcOpenChannel()***

rorcQuickFind

Synopsis

```
#include <rorc_lib.h>
int rorcQuickFind (int            *rorc_minor,
                   int            *rorc_revision,
                   unsigned long  *com_stat,
                   int            *pci_speed,
                   int            *rorc_serial,
                   int            *rorc_fw_maj,
                   int            *rorc_fw_min,
                   int            *max_chan,
                   int            *ch_pid0,
                   int            *ch_pid1,
                   int            max_dev)
```

Description Find all RORC cards. The ***rorcQuickFind()*** routine returns the minor number, revision number, PCI command / status information, PCI speed, hardware serial number, firmware version major and minor numbers and the IDs of processes using RORC channels for all RORC cards plugged in the PC. The routine gets this information from the `/proc/rorc_map` process file. The file is filled by the RORC driver using (except the process IDs) the data established at the boot time.

Parameters

rorc_minor pointer to an integer array containing the corresponding minor device numbers of the RORC cards.

rorc_revision pointer to an array of integers. Here the routine supplies the corresponding device revision number (1 for pRORC, 2 for D-RORC with connector for DIU, 3 for D-RORC with embedded DIUs, 4 for version 2

D-RORC and 5 for PCI express RORC) of the RORC cards.

com_stat — pointer to an array of unsigned long integers. Here the routine copies the value found in the device's command / status register. This value reflects the PCI settings of the given slot. The settings are correct if the value is
`0x04300107` for revision 2 or 3 cards,
`0x04100107` for revision 4 cards, and
`0x00100007` for revision 5 cards.

pci_speed — pointer to an array of integers. Here the routine supplies the speed settings of the cards. It could be 33, 66, 100 and 133 Hz. Note, the PCI-X type RORC card cannot work on 133 Hz.

rorc_serial — pointer to an array of integers. Here the routine loads the hardware serial number of RORC cards found.

rorc_fw_maj — pointer to an array of integers. The routine interprets the device firmware version number and loads here the major part. E.g. for firmware ID 2.12 this value is 2.

rorc_fw_min — pointer to an array of integers. The routine interprets the device firmware version number and loads here the minor part. E.g. for firmware ID 2.12 this value is 12.

max_chan — pointer to an array of integers. Here the routine supplies the number of channels (DIUs) of the given RORC cards.

ch_pid0 — pointer to an array of integers. Here the routine supplies the ID of the process currently using channel 0 of the given RORC cards or 0 if the channel is not in use.

ch_pid1 — pointer to an array of integers. Here the routine supplies the ID of the process currently using channel 1 of the given RORC cards or 0 if the channel is not in use.

max_dev — the size of `rorc_minor`, `rorc_revision`, `com_stat`, `pci_speed`, `rorc_serial`, `rorc_fw_maj`, `rorc_fw_min`, `max_chan`, `ch_pid0`, and `ch_pid1` arrays.

Return value number of DDL cards found on PCI bus or
`-1` if /proc/rorc_map cannot be open.

See also ***rorcFind(), rorcFindAll(), rorcSerial(), rorcOpenChannel()***

rorcFind

Synopsis

```
#include <rorc_lib.h>
```

```
int rorcFind(int revision, int serial, int *minor)
```

Description Find the specified RORC card. The ***rorcFind()*** routine returns the minor number of a RORC card with the specified revision and serial numbers. The minor number is necessary to open a DDL channel with the ***rorcMapChannel()*** or ***rorcOpenChannel()*** routines. At boot time the ***rorc_driver*** module matches the revision and serial numbers with the minor numbers and puts this info into the `/proc/rorc_map` process file. The ***rorcFind()*** routine reads this file and finds the given minor number. If the `/proc/rorc_map` file does not exist (it can happen if an older rorc_driver module is loaded, which does not create this file) then ***rorcFind()*** tries to open all the RORC devices plugged in the PC and reads the revision number from their PCI configuration space and the hardware serial number from their configuration EPROM.

If several cards have the same specified revision and serial numbers, then the routine returns the first one.

Parameters

revision	device revision number (1 for pRORC, 2 for D-RORC with connector for DIU, 3 for D-RORC with embedded DIUs, 4 for version 2 D-RORC and 5 for PCI express RORC cards) of the RORC card to be found.
serial	the serial number (a 5 digit decimal number) of the RORC card to be found.
minor	pointer to an integer where the minor number of the specified card has to be returned.

Return value

`RORC_STATUS_OK = 0`	the specified RORC was found. ***minor*** points to the minor number of the specified card.
`RORC_STATUS_ERROR = -1`	the specified RORC was not found, such a card is not plugged.

See also ***rorcFindAll(), rorcSerial(), rorcMapChannel(), rorcOpenChannel()***

rorcOpenChannel

Synopsis

```
#include <rorc_lib.h>

int rorcOpenChannel (rorcHandle_t   handle,
                     int           rorc_minor,
                     int           rorc_channel)
```

Description Arm and reset the DDL channel. The ***rorcOpenChannel()*** routine should be called for every DDL channel at the start of a run. The routine checks the existence of the RORC channel. If it finds the channel, it opens it and fills a descriptor. The descriptor address can be used as a handle for every further use of the given channel. The ***rorcOpenChannel()*** routine resets the RORC device and sends a command the DIU to find out whether there is any DIU plugged and if so, what is the given DIU version (prototype or final). The above information is written into

the handle structure. If one does not want the RORC to be reset, use the ***rorcMapChannel()*** routine instead.

Parameters

handle	address of a RORC descriptor structure. The rorcHandle_t type is a pointer to a rorcDescriptor_t structure, which contains all information about the PCI-based RORC. The structure type is defined in the ***rorc_lib.h*** file. The caller, before calling the ***rorcOpenChannel()*** routine, has to allocate a descriptor and supply its address to the routine. The routine fills the structure with data necessary for further calls.
rorc_minor	device file minor number of the RORC card. Multiple RORC cards can be supported (with device file names "/dev/prorcN", where N is the minor number). The minor numbers start from 0.
rorc_channel	RORC channel number (0 or 1). For pRORCs or D-RORCs without embedded DIUs only channel 0 can be used.

Return value

RORC_STATUS_OK = 0	no error, channel initialized and *handle* points to a valid RORC descriptor.
RORC_STATUS_ERROR = -1	the RORC channel couldn't be opened. Either no card was found or another process uses it or its PCI memory cannot be mapped.

See also ***rorcMapChannel(), rorcClose()***

rorcMapChannel

Synopsis

```
#include <rorc_lib.h
int rorcMapChannel (rorcHandle_t    handle,
                    int             rorc_minor,
                    int             rorc_channel)
```

Description Arm the RORC card. The ***rorcMapChannel()*** routine can be called instead of ***rorcOpenChannel()*** routine when one does not want to reset the open device. It should be called for every DDL channel at the start of a run. The routine checks the existence of the RORC channel. If it finds the channel, it opens it and fills a descriptor. The descriptor address can be used as a handle for further use of the given channel. The routine does not initialize the RORC card and does not send any command via the DDL channel.

To initialize the DDL components (reset RORC, DIU, SIU and establish the DDL link) use the routines ***rorcReset()*** or ***rorcArmDDL()***.

Parameters

handle	address of a RORC descriptor structure. The rorcHandle_t type is a pointer to a

	rorcDescriptor_t structure, which contains all information about the PCI-based RORC. The structure type is defined in the rorc_lib.h header file. The caller, before calling the ***rorcMapChannel()*** routine, has to allocate a descriptor and supply its address to the routine. The routine fills the structure with data necessary for further calls.
rorc _minor	device file minor number of the RORC card. Multiple RORC cards can be supported (with device file names "/dev/prorcN", where N is the minor number). The minor numbers start from 0.
rorc_channel	RORC channel number (0 or 1). For pRORCs or D-RORCs without embedded DIUs only channel 0 can be used.

Return value

RORC_STATUS_OK = 0	no error, channel initialized and *handle* points to a valid RORC descriptor.
RORC_STATUS_ERROR = -1	the RORC channel couldn't be opened. Either no card was found or another process uses it or its PCI memory cannot be mapped.

See also ***rorcOpenChannel(), rorcReset(), rorcArmDDL(), rorcClose()***

rorcClose

Synopsis

```
#include <rorc_lib.h>
int rorcClose(rorcHandle_t handle)
```

Description Close the RORC channel. The ***rorcClose()*** routine should be called for every DDL channel at the end of a run. The routine closes all resources set up by a previous call of the routines ***rorcOpenChannel()*** or ***rorcMapChannel()***.

Parameters

handle	address of the RORC descriptor. When the routine returns the handle, it will point to an invalid descriptor.

Return value

RORC_STATUS_OK = 0	no error, channel closed
RORC_STATUS_ERROR = -1	the RORC channel couldn't be closed properly.

See also ***rorcOpenChannel(), rorcMapChannel()***

rorcReset

Synopsis

```
#include <rorc_lib.h>
void rorcReset(rorcHandle_t   handle,
```

```
                    int             option)
```

Description Reset the RORC channel. The ***rorcReset()*** routine initializes the RORC card and / or a DDL channel. According to the user request the routine resets the `Free FIFO`, the other parts of the RORC, the DIU or the SIU. Resetting the RORC channel means to empty all its FIFOs, including the `Free FIFO` and error bits, and then putting all programmable features to their reset values. Resetting the DIU or the SIU means cutting the DDL link; afterwards the DDL link rebuilds itself.

Parameters

handle	address of the RORC descriptor.	
option	the following values can be used:	
	`RORC_RESET_FF`	clear Rx and Tx Free FIFOs
	`RORC_RESET_FIFOS`	clear RORC's other FIFOs
	`RORC_RESET_ERROR`	clear RORC's error bits
	`RORC_RESET_COUNTERS`	clear RORC's byte counters
	`RORC_RESET_RORC`	reset RORC
	`RORC_RESET_DIU`	reset DIU
	`RORC_RESET_SIU`	reset SIU
	`0`	reset RORC

See also ***rorcOpenChannel(), rorcMapChannel(), rorc_reset***

rorcEmptyDataFifos

Synopsis

```
#include <rorc_lib.h>
void rorcEmptyDataFifos(rorcHandle_t        handle,
                        int                 timeout)
```

Description Try to empty all data FIFOs of the RORC channel.The ***rorcEmptyDataFifos()*** routine tries to empty the RORC card's receive (Rx) and transmit (Tx) data FIFOs by continuously sending the 'clear RORC FIFOs' command and checking Rx FIFO status. It is not enough to send the command only once, because new data can arrive from the - not empty - FIFOs of the FEE, SIU or DIU. The routine returns if the RORC's Rx FIFO is empty or the time-out has expired.

Parameters

handle	address of the RORC descriptor.
timeout	time-out value in μsecs.

Return value

`RORC_STATUS_OK = 0`	no error, data FIFOs emptied.
`RORC_TIMEOUT TIMEOUT = -64`	there are still data in RORC's Rx FIFO after timeout.

See also ***rorcReset(), rorcArmDDL()***

rorcArmDataGenerator

Synopsis

```
#include <rorc_lib.h>

int rorcArmDataGenerator(rorcHandle_t   handle,
                         __u32          initEventNumber,
                         __u32          initDataWord,
                         int            dataPattern,
                         int            eventLen,
                         int            seed,
                         int            *rounded_length)
```

Description

Initialize RORC's Data generator. The ***rorcArmDataGenerator()*** routine should be called for every DDL channel where the RORC card will be used as data generator. The routine can be called after the call of ***rorcOpenChannel()*** and before the call of ***rorcStartDataGenerator()*** routines. It defines all the parameters needed for data generation. If ***rorcStartDataGenerator()*** is called without calling ***rorcArmDataGenerator()***, then the data generator will use unpredictable values.

Parameters

handle	address of the RORC descriptor.
initEventNumber	each event starts with the serial number of the given event (event count). This parameter defines the starting value of it.
initDataWord	the first data word of the event (after the event count). It is used only for some of the test patterns. Note: for D-RORC, if the seed is not RORC_DG_NO_RANDOM_LEN, the first data word of each event is 0.
dataPattern	an integer between 1 and 7:
RORC_DG_CONST:	all data words are *initDataWord*.
RORC_DG_ALTER:	alternating pattern, starting from *initDataWord*
RORC_DG_FLY0:	flying 0 starting from 0xfffffffe
RORC_DG_FLY1:	flying 1 starting from 0x00000001
RORC_DG_INCR:	incrementing data starting from *initDataWord*
RORC_DG_DECR:	decrement data starting form *initDataWord*
RORC_DG_RANDOM:	random data
eventLen	length (from 1 to 2^19-1) of the generated events in 32 bit words, including the event count. Important: because of the special features of the random number generation, if random length is used (seed is not equal to RORC_DG_NO_RANDOM_LEN), the minimum generated event length is 1, and the maximum value of the length will be *eventLen* rounded down to the nearest integer of power of 2.
seed	defines the seed value for random data length. If given, the event lengths will vary between 1 and *eventLen*.

	Using the value `RORC_DG_NO_RANDOM_LEN` no random length will be generated.
rounded_length	it is an output parameter: in case of random length generation the maximum event length is rounded to the nearest integer of power of two. The routine transfers this value to RORC as the maximum length and returns it to the user in this variable.

Return value

`RORC_STATUS_OK = 0`	no error, data generator initialized
`RORC_INVALID_PARAM = -2`	error: some of the parameters out of range.

See also ***rorcOpenChannel()*, *rorcStartDataGenerator()***

rorcArmDDL

Synopsis

```
#include <rorc_ddl.h>
int rorcArmDDL(rorcHandle_t     handle,
               int              options)
```

Description Arm the DDL. The ***rorcArmDDL()*** routine should be called for every DDL channel when the RORC card is not used as data generator but data come from the Front-end Electronics (FEE). The purpose of the routine is to establish or check the connection between the DIU and SIU, to reset all components and to clear all data, which could remain in the channel from previous use of the link. According to the user request the routine resets the `Free FIFO`, the other parts of the RORC, the DIU or the SIU units. If several reset requests are “OR-d”, the program first resets the SIU, then establishes the link, then resets the DIU and at last resets the RORC. Resetting the RORC card means emptying all its FIFOs, including the `Free FIFO`, and then put all programmable features to their reset values. Resetting the DIU or SIU means cutting the DDL link (if it was on before the call of the routine). In the case of prototype version of the DDL cards, after the link cut, the link has to be re-established by calling ***rorcArmDDL()*** with RORC_LINK_UP parameter. In case of the final DDL cards, the link is set up automatically.

Parameters

handle		address of the RORC descriptor.
options		the following values can be “OR-d”:
	`RORC_RESET_FF`	reset Free FIFO
	`RORC_RESET_RORC`	reset RORC
	`RORC_RESET_DIU`	reset DIU
	`RORC_RESET_SIU`	reset SIU
	`RORC_LINK_UP`	establish the DDL link

Return value

`RORC_STATUS_OK = 0`	no error, requested task done.
`RORC_LINK_NOT_ON = -4`	link initialization did not succeed
`RORC_CMD_NOT_ALLOWED =-8`	routine called with not permitted option

`RORC_NOT_ACCEPTED = -16` unsuccessful SIU reset

See also ***rorcArmDataGenerator(), rorcReset(), rorcEmptyDataFifos()***

rorcPushFreeFifo

Synopsis

```
#include <rorc_lib.h>
void rorcPushFreeFifo(rorcHandle_t     handle,
                      rorcMemAddres_t  blockAddress,
                      __u32            blockLength,
                      int              readyFifoIndex)
```

Description Push one entry into RORC's Free FIFO. The ***rorcPushFreeFifo()*** is an in-line function what should be called when the user has a free data page and wants to load its parameters into the Free FIFO. It loads the parameters directly into the RORC registers.

The function does not check the range of the parameters: it masks them for the given range. It neither checks the `Free FIFO` status. If the `Free FIFO` is overflowed then the new parameters will not be loaded. The caller can check this situation using ***rorcCheckFreeFifo()***.

Parameters

handle	address of the RORC descriptor.
blockAddress	physical address of the next free page in physmem memory.
blockLength	length of the next free page in byte (24 bit).
readyFifoIndex	index of the ready FIFO, where the "data arrived" flag has to be put to (8 bit).

See also ***rorcCheckFreeFifo()***

rorcCheckFreeFifo

Synopsis

```
#include <rorc_lib.h>
int rorcCheckFreeFifo(rorcHandle_t handle)
```

Description Return the status of the RORC's `Free FIFO`. The ***rorcCheckFreeFifo()*** should be called when the caller wants to know how many FIFO entries are in the `Free FIFO`. Using pRORC device, it returns the number of entries in "8 entry" units (i.e. 0 means 1 to 8 entries, 1 means 9 to 16 entries, etc.). FIFO full and FIFO empty statuses are signaled. In the case of D-RORC (RORC revision number > 1), the routine only signals if the `Free FIFO` is not empty (returns not 0).

Parameters

handle	address of the RORC descriptor.

Return value In case of pRORC:

a value between 1 and 15 specifying number of not empty Free FIFO entries in the following way:

0	Between 1 and 8 words
1	Between 9 and 16 words
2	Between 17 and 24 words
3	Between 25 and 32words
…….	…….
13	Between 105 and 112 words
14	Between 113 and120 words
15	Between 121and 128 words

RORC_STATUS_OK = 0	Free FIFO is not empty (and not full).
RORC_FF_EMPTY = -256	Free FIFO is empty.
RORC_FF_FULL = -128	error: Free FIFO full.

In case of D-RORC:

0:	Free FIFO is empty,
any other value:	Free FIFO is not empty.

See also ***rorcPushFreeFifo()***

Setting RORC parameters on/off

The following 6 routines can be used to set RORC internal control parameters on or off.

rorcLoopBackOn

Synopsis

```
#include <rorc_lib.h>
int rorcLoopBackOn(rorcHandle_t handle)
```

Description The ***rorcLoopBackOn()*** routine should to be called when the user wants to set the operational control parameter “Internal Loop-back” bit. If this control bit is on, the data generated by the RORC’s Data Generator will be sent back to the RORC as if it had arrived from the link.

This conditions can be reset by the routine ***rorcLoopBackOff()***.

Parameters

handle	address of the RORC descriptor.

Return value `RORC_STATUS_OK = 0` no error

rorcLoopBackOff

Synopsis

```
#include <rorc_lib.h>
int rorcLoopBackOff(rorcHandle_t handle)
```

Description The ***rorcLoopBackOff()*** routine should to be called when the user wants to reset the operational control parameter “Internal Loop-back” bit

This condition is automatically set after RORC reset.

Parameters *handle* address of the RORC descriptor.

Return value *RORC_STATUS_OK = 0* no error

rorcHltSplitOn

Synopsis

```
#include <rorc_lib.h>
int rorcHltSplitOn(rorcHandle_t handle)
```

Description The D-RORC card with 2 integrated DIU can be used in “split mode”. It means that the data arriving on one channel can be transferred to the other channel. The ***rorcHltSplitOn()*** routine should to be called when the user wants the given channel to be used as output channel.

This conditions can be reset by the routine r***orcHltSplitOff()***.

Parameters *handle* address of the RORC descriptor.

Return value

`RORC_STATUS_OK = 0`	no error
`RORC_CMD_NOT_ALLOWED = -8`	the routine cannot be called for pRORCs or D-RORCs without integrated DIU.

rorcHltSplitOff

Synopsis

```
#include <rorc_lib.h>
int rorcHltSplitOff(rorcHandle_t handle)
```

Description The ***rorcHltSplitOff()*** routine should to be called when the user wants to switch off the data sending for the given channel.

This condition is automatically set after a RORC reset.

Parameters *handle* address of the RORC descriptor.

Return value

RORC_STATUS_OK = 0	no error
RORC_CMD_NOT_ALLOWED = -8	the routine cannot be called for pRORCs or D-RORCs without integrated DIU.

rorcHltFlctlOn

Synopsis

```
#include <rorc_lib.h>
int rorcHltFlctlOn(rorcHandle_t handle)
```

Description The D-RORC card with 2 integrated DIU can be used in “split mode”. It means that the data arriving on one channel can be transferred to the other channel. The ***rorcHltFlctlOn()*** routine should to be called when the given channel is used as the output channel (***rorcHltSplitOn()*** is called or will be called) and the user wants the flow control from the receiver side (probably the HLT farm) be taken into account.

This conditions can be reset by the routine ***rorcHltFlctlOff()***.

Parameters *handle* address of the RORC descriptor.

Return value

RORC_STATUS_OK = 0	no error
RORC_CMD_NOT_ALLOWED = -8	the routine cannot be called for pRORCs or D-RORCs without integrated DIU.

rorcHltFlctlOff

Synopsis

```
#include <rorc_lib.h>
int rorcHltFlctlOff(rorcHandle_t handle)
```

Description The ***rorcHltFlctlOff()*** routine should to be called when the given channel is used as the output channel (***rorcHltSplitOn()*** is called or will be called) and the user wants the flow control from the receiver side (probably the HLT farm) NOT be taken into account.

This condition is automatically set after RORC reset.

Parameters *handle* address of the RORC descriptor.

Return value

RORC_STATUS_OK = 0	no error
RORC_CMD_NOT_ALLOWED = -8	the routine cannot be called for pRORCs or D-RORCs without integrated DIU.

ddlSendCommandAndWaitReply

Synopsis

```
#include <rorc_ddl.h>

int ddlSendCommandAndWaitReply(rorcHandle_t   handle,
                               __u32          feeCommand,
                               __u32          feeAddress,
                               long long      timeout,
                               stword_t       *stw,
                               int            expected,
                               int            *n_reply)
```

Description

Send a command and wait for the reply. The ***ddlSendCommandAndWaitReply()*** routine should to be called when the user wants to send a command to the FEE via the DDL channel. The routine returns the received replies.

Parameters

handle	address of the RORC descriptor.
feeCommand	a maximum 4-bit long value which will be sent to the FEE as a part the command. The following FEE commands are allowed:
RDYRX = 1	Ready to Receive
EOBTR = 11	End of Block Transfer
STBWR = 13	Start of Block Write
STBRD = 5	Start of Block Read
FECTRL = 12	Front-end control
FESTRD =4	Front-end status readout
feeAddress	a maximum 19-bit long value which will be sent to the FEE as a part of the command.
timeout	the number of waiting cycles for receiving the SIU reply. If you want to specify the timeout value in microseconds, then use the value (<timeout in ms> * handle->loop_per_usec).
stw	pointer to an array of status word structures where the routine returns the received statuses.
expected	number of expected reply words.
n_reply	pointer to a variable where the routine returns the number of received statuses.

Return value

RORC_STATUS_OK = 0	no error, the command sent, the expected number of reply words received
RORC_LINK_NOT_ON = -4	error: the link is down
RORC_TIMEOUT = -64	error: command can not be sent in time specified by timeout

`RORC_TOO_MANY_REPLY = -512`	error: too many replies arrived or before sending the command, the RORC's received FIFO contained already some words from a previous command
`RORC_NOT_ENOUGH_REPLY = -1024`	error: less reply arrived then expected in time specified by `timeout`

See also ***rorc_send_command***

rorcStartDataGenerator

Synopsis

```
#include <rorc_lib.h>

int rorcStartDataGenerator(rorcHandle_t    handle,
                           __u32           maxLoop)
```

Description Set RORC to start sending generated data. The ***rorcStartDataGenerator()*** routine should to be called when the user wants to receive generated data. Normally the Data Generator sends the data to the DDL link. If the user wants the simulated data to arrive in the PC, then the RORC has to be set to loop-back mode before starting the Generator. This can be done by the routine ***rorcLoopBackOn()***.

Data will arrive only when data receiver is started by calling the ***rorcStartDataReceiver()*** routine, and the RORC's `Free FIFO` is not empty. Features of the generated data (data pattern, event length, event frequency) can be defined by a previous call to ***rorcArmDataGenerator()***. To stop the data generator (in the case of infinite number of events) call the routine ***rorcStopDataGenerator()***.

Parameters

handle	address of the RORC descriptor.
maxLoop	number of events to be generated. Possible values are from 1 to 2^32-1, or RORC_DG_INFINIT_EVENT (infinite number of events).

Return value

`RORC_STATUS_OK = 0`	no error, data generator started

See also ***rorcArmDataGenerator(), rorcLoopBackOn(), rorcStartDataReceiver(), rorcStopDataGenerator()***

rorcStopDataGenerator

Synopsis

```
#include <rorc_lib.h>

int rorcStopDataGenerator(rorcHandle_t handle)
```

Description Stop sending generated data. The ***rorcStopDataGenerator()*** routine should to be called when the user wants to stop receiving generated data. The data generator stops sending events when the number of events set in ***rorcStartDataGenerator()*** is reached. However ***rorcDataStopGenerator()*** has to be called to set the RORC card into normal state. If data sending is going on when this routine is called, then the current event will be finished and no more data will be sent. If the transfer is stuck, one has to reset the RORC card.

Parameters

handle	address of the RORC descriptor.

Return value

RORC_STATUS_OK = 0	no error, data generator stopped

See also ***rorcStartDataGenerator()***

rorcStartDataReceiver

Synopsis

```
#include <rorc_lib.h>
int rorcStartDataReceiver(rorcHandle_t     handle,
                          unsigned long    readyFifoBaseAddress)
```

Description Set the DDL channel to data collecting state. The ***rorcStartDataReceiver()*** routine should to be called when the user wants to receive data via the DDL channel.

Parameters

handle	address of the RORC descriptor.
readyFifoBaseAddress	the physical memory address of the Ready FIFO. It must be a multiple of 2K, i.e. the lower 11 bits of the Ready FIFO address must be 0.

Return value

RORC_STATUS_OK = 0	no error, data collection started.

See also ***rorcStopDataReceiver()***

rorcStopDataReceiver

Synopsis

```
#include <rorc_lib.h>
int rorcStopDataReceiver(rorcHandle_t handle)
```

Description Stop data collecting. The ***rorcStopDataReceiver()*** routine should to be called when the user wants to stop receiving data via the DDL channel.

Parameters

handle	address of the RORC descriptor.

Return value `RORC_STATUS_OK = 0` no error, data collection stopped.

See also ***rorcStartDataReceiver()***

ddlReadDataBlock

Synopsis

```
#include <rorc_ddl.h>
int ddlReadDataBlock(rorcHandle_t    handle,
                     unsigned long   bufferPhysAddress,
                     unsigned long   returnPhysAddress,
                     rorcReadyFifo_t *returnAddr,
                     __u32           feeAddress,
                     long long       timeout,
                     stword_t        *stw,
                     int             *n_reply,
                     int             *step)
```

Description Read a data block from the FEE. The ***ddlReadDataBlock()*** routine should to be called when the user wants to read a data block from the FEE via the DDL channel. The routine fulfils the following 3 steps:

1. Sends a *Start Block Read* (STBRD) command to the FEE, specifying the front-end address where the data is.

2. Receives the data block.

3. Sends an *End Of Block Transfer* (EOBTR) command to the SIU.

If an error occurs in any of the above steps, the routine returns an error code, the step number and the received reply from the FEE or SIU.

Parameters

handle	address of the RORC descriptor.
bufferPhysAddress	the physical memory address of the data.
returnPhysAddress	the physical memory address of a word where the number of transferred word and a status word will be put when the transfer had finished. When using D-RORC the address must be 2K aligned, i.e. its lower 11 bits must be 0. The routine writes –1 this address before sending the data and polls this address while the transfer is done.
returnAddress	a pointer to the virtual address of the above physical memory.
feeAddress	a maximum 19-bit long value which will be sent to the FEE in the STBRD command.
timeout	the number of waiting cycles for receiving the SIU reply. If you want to specify the timeout value in

	microseconds, then use the value (<timeout in ms> * handle->loop_per_usec)
stw	pointer to an array of status word structures where the routine returns the received status.
n_reply	pointer to a variable where the routine returns the number of received status.
step	pointer to a variable where the routine returns the step number at which the routine returned from.

Return value

RORC_STATUS_OK = 0	no error
RORC_LINK_NOT_ON = -4	error: the link is down
RORC_TIMEOUT = -64	error: command can not be sent in time *timeout*
RORC_TOO_MANY_REPLY = -512	error: too many replies arrived
RORC_NOT_ENOUGH_REPLY = -1024	error: less reply arrived then expected in time *timeout*

See also ***ddlWriteDataBlock()***

ddlWriteDataBlock

Synopsis

```
#include <rorc_ddl.h>
int ddlWriteDataBlock(rorcHandle_t           handle,
                      unsigned long          bufferPhysAddress,
                      unsigned long          bufferWordLength,
                      unsigned long          returnPhysAddress,
                      volatile unsigned long *returnAddr,
                      __u32                  feeAddress,
                      long long              timeout,
                      stword_t               *stw,
                      int                    *n_reply,
                      int                    *step)
```

Description Send a data block to the FEE. The ***ddlWriteDataBlock()*** routine should to be called when the user wants to send a data block to the FEE via the DDL channel. The routine fulfils the following 3 steps:

1. Sends a *Start Block Write* (STBWR) command to the FEE, specifying the front-end address where the data has to be written.

2. Sends the data block.

3. Sends an *End Of Block Transfer* (EOBTR) command to the SIU.

If an error occurs in any of the above steps, the routine returns an error code, step number and the received reply from the FEE or SIU.

Parameters

handle	address of the RORC descriptor.
bufferPhysAddress	the physical memory address of the data.
bufferWordLength	the length of the data block in 32 bit words. The maximum length is 512 K words – 1 word.
returnPhysAddress	the physical memory address of a word where the number of transferred word will be put when the transfer had finished. When using D-RORC the address must be 2K aligned, i.e. its lower 11 bits must be 0. The routine writes –1at this address before sending the data and polls this address while the transfer is done.
returnAddress	a pointer to the virtual address of the above physical memory.
feeAddress	a maximum 19-bit long value which will be sent to the FEE in the STBWR command.
timeout	the number of waiting cycles for receiving the SIU reply. If you want to specify the timeout value in microseconds, then use the value (<timeout in ms> * handle->loop_per_usec)
stw	pointer to an array of status word structures where the routine returns the received status.
n_reply	pointer to a variable where the routine returns the number of received status.
step	pointer to a variable where the routine returns the step number at which the routine returned from.

Return value

RORC_STATUS_OK = 0	no error
RORC_LINK_NOT_ON = -4	error: the link is down
RORC_TIMEOUT = -64	error: command can not be sent in time *timeout*
RORC_NOT_ABLE = -32	error: the previous download was not finished in time *timeout*
RORC_TOO_MANY_REPLY = -512	error: too many replies arrived
RORC_NOT_ENOUGH_REPLY = -1024	error: less reply arrived then expected in time *timeout*

See also ***ddlReadDataBlock()***

rorcStartTrigger

Synopsis

```
#include <rorc_ddl.h>
```

```
int rorcStartTrigger(rorcHandle_t         handle,
                     long long            timeout,
                     stword_t             stword)
```

Description The ***rorcStartTrigger()*** routine sends a RDYRX command to the FEE.

Parameters

handle	address of the RORC descriptor
timeout	the number of waiting cycles for receiving the FEE reply. If you want to specify the timeout value in microseconds, then use the value (<timeout in ms> * handle->loop_per_usec)
stword	the FEE reply: a DDL status word stword.stw contains the full reply. For the details of a status word, see the rorc_ddl.h.

Return value

RORC_STATUS_OK = 0	the RDYRX command was sent successfully.
RORC_STATUS_ERROR = -1	the RORC was not able to send the command.
RORC_LINK_NOT_ON = -4	the link is down; the RORC is not able to send the command.
RORC_NOT_ACCEPTED = -16	No reply arrived from SIU within the specified *timeout*.

See also ***rorcStopTrigger()***

rorcStopTrigger

Synopsis

```
#include <rorc_ddl.h>
int rorcStopTrigger(rorcHandle_t   handle,
                    long long      timeout
                    stword_t       stword)
```

Description The ***rorcStopTrigger()*** routine sends an EOBTR command to the FEE.

Parameters

handle	address of the RORC descriptor
timeout	the number of waiting cycles for receiving the FEE reply. If you want to specify the timeout value in microseconds, then use the value (<timeout in ms> * handle->loop_per_usec).
stword	the FEE reply: a DDL status word stword.stw contains the full reply. For the details of a status word, see the rorc_ddl.h.

Return value

RORC_STATUS_OK = 0	the EOBTR command was sent successfully.

`RORC_STATUS_ERROR = -1`	the RORC was not able to send the command.
`RORC_LINK_NOT_ON = -4`	the link is down (the RORC is not able to send the command).
`RORC_NOT_ACCEPTED = -16`	no reply arrived from SIU within the specified *`timeout`*.

See also ***rorcStartTrigger()***

rorcSerial

Synopsis

```
#include <rorc_lib.h>

rorcHwSerial_t rorcSerial(rorcHandle_t handle)
```

Description Reads RORC's version and serial numbers. The ***rorcSerial()*** routine reads from the card's configuration EPROM its hardware version and serial numbers. The routine ***rorcInterpretSerial()*** interprets the relevant fields and print them to standard output.

Parameters

`handle`	address of the RORC descriptor

Return value

`structure rorcHwSerial_t`	The routine loads into this structure the version and serial numbers of the RORC card. `rorcHwSerial_t` is defined in `rorc_lib.h`. Besides the major and minor version and the serial numbers it contains the full string retrieved from the configuration EPROM of the RORC card. If there is no information in the EPROM about the hardware version and serial numbers then the routine writes –1 into the structure.

See also ***rorcFind(), rorcFindAll(), rorcReadFw(), ddlSerial()***

rorcReadFw

Synopsis

```
#include <rorc_lib.h>

int rorcReadFw(rorcHandle_t     handle)
```

Description The ***rorcReadFw()*** function reads the RORC's firmware identification word. The routine ***rorcInterpretFw(fw)*** interprets the relevant fields and print them to standard output. The inline function ***rorcFFSize(fw)*** returns the number of `Free FIFO` entries of the card, while ***rorcFWVersMajor(fw)*** and ***rorcFWVersMinor(fw)*** return the major and minor version numbers of the card's firmware.

Parameters *handle* address of the RORC descriptor.

Return value The returned word contains the RORC's firmware identification in the following format:

`bits 0-4:` day

`bits 5-8:` month

`bits 9-12:` year form 2000

`bits 13-24` version number of the pRORC card's firmware

`bits 25-31` `Free FIFO` size of the card in 64 units.

See also ***rorcSerial()***

rorcReadRorcStatus

Synopsis

```
#include <rorc_lib.h>
int rorcReadRorcStatus(rorcHandle_t         handle,
                       rorcStatus_t         *status)
```

Description The ***rorcReadRorcStatus()*** function fills a structure (defined in `rorc_lib.h`) containing information about RORC status and errors, such as: the working mode of the RORC, `Free FIFO` status, link status, flow control status, etc. Before calling the ***rorcRorcReadStatus()***, the caller has to allocate a `rorcStatus_t` structure and supply its address to the routine. The routine fills this structure.

The `rorcStatus_t` structure contains three members:

`ccsr`: the copy of the RORC's Operation Control and Status Register,

`cerr`: the copy of the RORC's Error Register,

`cdgs`: the copy of the RORC's Data Generator Status Register.

The meaning of the status and error bits can be found in rorc_lib.h. The routines ***rorcInterpretStatus(ccsr)*** and ***rorcInterpretError(cerr)*** interpret the relevant register bits and print them to standard output.

Parameters

handle	address of the RORC descriptor.
status	address of a `rorcStatus_t` type structure. The routine fills into this structure the RORC status information.

Return value

`RORC_STATUS_OK = 0`	no error, RORC status structure filled

ddlSerial

Synopsis

```
#include <rorc_ddl.h>
rorcHwSerial_t ddlSerial(rorcHandle_t   handle,
                         int            destination,
                         long long      timeout)
```

Description Read the version and serial numbers of the DIU or SIU card. Send command to the DIU or SIU requesting the hardware version and serial numbers. The routine works only for plugged DIU and DDL cards of the final version. The routine ***rorcInterpreHwtSerial()*** interprets the relevant fields and prints them to standard output.

Parameters

handle	address of the RORC descriptor
destination	`DIU` or `SIU`
timeout	the number of waiting cycles for receiving the DDL card's reply. If you want to specify the timeout value in microseconds, then use (`<timeout in µs> * handle->loop_per_usec`).

Return value

`structure rorcHwSerial_t`	The routine loads into this structure the version and serial numbers of the DDL (DIU or SIU) card. `rorcHwSerial_t` is defined in `rorc_lib.h`. Besides the major and minor version and the serial numbers it contains the full string received from the card. If there is no information received, then the routine writes `-1` into the structure (this is the case for the prototype version DDL cards or integrated DIUs).

See also ***rorcSerial()***

rorcHasData

Synopsis

```
#include <rorc_lib.h>
int rorcHasData(rorcReadyFifo_t    readyFifoBaseAddr,
                int                readyFifoIndex)
```

Description Check the `Ready FIFO` for new data block. The calling program has to specify the `Ready FIFO` base address and index. It polls the `Ready FIFO` entry and signals if a data block arrived.

This routine is an in-line function. It does not return values from the Ready FIFO. The caller can read the block length and the status from the FIFO.The routine only

returns the information of block arrival. The memory address of the given block (and the other blocks of the same event) has to be known by the caller.

Parameters

readyFifoBaseAddr	base address of the `Ready FIFO`
readyFifoIndex	index of the `Ready FIFO` where the checking has to be done

Return value

`RORC_DATA_BLOCK_NOT_ARRIVED = 0`	no data block (data page) arrived (`Ready FIFO` status = -1)
`RORC_NOT_END_OF_EVENT_ARRIVED = 1`	data block (data page) arrived but not end-of-event block (`Ready FIFO` status = 0)
`RORC_LAST_BLOCK_OF_EVENT_ARRIVED= 2`	end-of-event data block arrived (`Ready FIFO` status = DTSTW). If the continuation bit (bit 8) is set, the event will continue.

rorcCheckLink

Synopsis

```
#include <rorc_lib.h>
int rorcCheckLink(rorcHandle_t handle)
```

Description Check a status word of the RORC card which reflects the link status.

Parameters

handle	address of the RORC descriptor.

Return value

`RORC_STATUS_OK = 0`	the DDL link is on
`RORC_LINK_NOT_ON = -4`	the DDL link is not on

See also ***rorcReadRorcStatus(), rorcArmDDL()***

21.5 Installation

The DDL-RORC Library and Test Programs are installed together with the DATE. For a stand-alone installation, follow the given procedure below:

- The header, source, object and executable files of RORC and DDL test programs and library are in a common afs area:`/afs/cern.ch/alice/daq/ddl/rorc/`
- This directory contains the different versions of the software as separate sub directories. These sub directories also contain the different versions in compressed formats.
- The compressed file names show the version number and the time of archiving. Use always the latest date of a given version. The latest distributed version can be found in the DDL home page as well: ***http://cern.ch/ddl/rorc_support.html***.
- Copy the compressed file onto your area, uncompress it and extract all directories and files from it. Use the following command for extracting files:

  ```
  > gtar –xvzf rorc_vers.<x.y.z>_<year.month.day>.tgz
  ```

- You will get a directory structure with the following subdirectories:

`rorc/`	source, header and make files
`rorc/Linux/`	executables and compiled API libraries
`rorc/examples/`	programs showing the usage of API libraries
`rorc/scripts/`	some functional test scripts

- Some test programs use the ***physmem*** memory manager module (see Chapter 15). If DATE is installed then ***physmem*** is installed as well. For a stand-alone installation, one can find the package in the DDL home page at ***http://cern.ch/ddl/rorc_support.html***.
- To compile it type the following commands:

  ```
  > cd rorc
  > make -f Makefile clean
  > make –f Makefile
  ```

- To compile the ***rorc_driver*** type:

  ```
  > make driver -f Makefile
  ```

- To register the driver and to create the device files type as ***root***:

  ```
  > make dev -f Makefile
  ```

  The driver will be automatically loaded at boot time.

- If you want to load the ***rorc_driver*** kernel module without rebooting the machine, type as root:

  ```
  > make load –f Makefile
  ```

- If an older version of the RORC driver is already loaded then run:

```
> make reload -f Makefile
```

Part IV

Detector Algorithms Framework

November 2010

ALICE DAQ Project

22 Detector Algorithms Framework

The online calibration tasks for the detectors are implemented using the Detector Algorithms (DA) Framework. The framework is available for download at:

http://cern.ch/alice-daq/DA-framework

This chapter describes the architecture and interfaces to implement Detector Algorithms.

22.1 Introduction

The ALICE sub-detectors require specific calibration tasks to be performed regularly in order to achieve the most accurate physics measurements. These systems are indeed sensitive to configuration settings, mechanical geometry, environmental conditions changes, components aging and sensors defects. The corresponding set of procedures to calibrate the sub-detectors involves events analysis in a wide range of experimental conditions. These calibration tasks may be done either in dedicated runs, or in parallel to physics data taking. Typical examples of calibrations include pedestal and gain computation, dead and noisy channels mapping, etc. Depending on the sub-detector and the calibration task, one has to define in particular:

- The trigger type, which can be a normal physics trigger, or some specific events related to dedicated hardware device (e.g. laser, LED, pulser).
- The number of events to collect, from a hundred events for pedestal runs to millions of events for dead channel mapping.
- The event formatting, zero-suppressed or not, which impacts on the required throughput. Event size ranges from sub-events of few kilobytes to 20 MB.
- The detector electronics settings, specific to the sub-detector operation mode.
- The calibration algorithm, i.e. the actual code to interpret the data and produce results.
- The type of run, standalone or global, depending if the task can be performed during normal data taking or requires a specific run. It has an impact on the operation mode and detector dead-time for physics.
- The frequency at which the calibration is required, from few times per day to once a year.

The calibration results produced may be needed to configure the detector electronics for data taking, for example to produce zero-suppressed data or to mask noisy channels, in order to reduce the data volume. Therefore, these results should be available right after the calibration data-taking procedure, in order to reconfigure the detector accordingly for the next physics run. In addition, the results are also used offline for the events reconstruction. Both usages of the results involve a drastic timing constraint on the way they are produced. It would be too heavy to make the full calibration analysis offline (a first pass over the data would be needed to produce calibration results), and sometimes too late (for calibrations required very frequently, or for which results are needed for the detector configuration). Only the most complex calibration data analysis should be done offline.

Therefore, a dedicated framework has been designed and implemented to achieve as much as possible the detector calibration directly online, and to address the heterogeneous requirements specific to each calibration task.

22.2 The Detector Algorithms (DAs)

The ALICE online calibration framework is used to implement and run a set of detector algorithms (DAs), which are calibration tasks running online. DAs are provided by the sub-detector teams, using the global framework to develop detector-specific calibration procedures.

Each DA grabs detector data and produces results online. These results can be reused directly online, e.g. to configure the detector, or shipped offline to be post-processed (if necessary) and used in event reconstruction.

To cover all the needs, we have defined two types of DAs, running either in exclusive mode (a dedicated run is required), or in the background (the task can be performed in a physics run):

- In the first case, called 'LDC DA', the data is recorded locally and in parallel on the LDCs, during a dedicated standalone run (single detector running), usually of short duration. At end of run, a DA process is launched on each LDC to analyze the data. In this mode, parallelization is optimal, and results are readily available for further export to FEE. Typical example is the pedestal run, with few hundreds of big events. The temporary data files stored on local disk are useful to re-play, tune or debug the DA behavior.
- In the second case, called 'MON DA', a single DA process of a given type is active during the run, on a dedicated monitoring machine. Data samples are picked up from the normal data flow in a non-intrusive way, and processed directly on the fly. The DA gets only what it can process, events may be dropped in case the DA is busy. The DA selects the type of events it needs (calibration, physics) and the source to monitor (typically, a detector or a set of detectors). At the end of run, the DA goes in a post-processing phase to finalize the results. A generic example is to populate an histogram event by event, and at the end of run compute a fit and extract some key values. Another example of MON DA usage is for the dead channel mapping, where millions of events may be needed to cover the full detector. Many runs may be needed to collect such statistics, in which case intermediate results are saved at end of run, and re-loaded at the next start of run.

22.3 DA framework architecture

The overall DA framework architecture, and in particular the interaction of the DA processes with the online components, can be seen in Figure 22.1. The DA process consists of detector code, written in C++ using the ***AliROOT*** framework, which is the ALICE offline code repository (therefore providing the same calibration algorithm implementation for both online and offline environments). This code uses the DAQ DA interface library in order to communicate with the other systems. The ECS is in charge of launching the DA where needed for the corresponding run type, depending on the experiment running mode selected by the operator. The run type is propagated to the other online systems and to the detector, in order to make

sure corresponding settings are applied. This information is also stored in the experiment logbook (see Chapter 24) for bookkeeping and further reference.

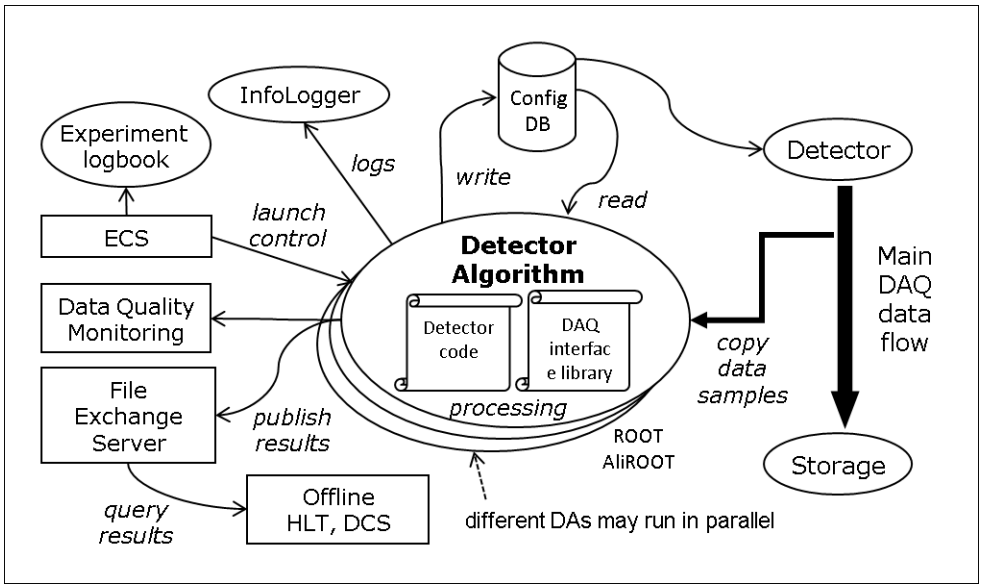

Figure 22.1 DA framework architecture.

Upon startup, the DA connects to the main DAQ data flow (being local files for LDC DA, or remote data sources for MON DA) in order to collect events.

The DA may use some configuration information stored centrally in the detector database (see Section 4.4.6), which allows the operator to define the DA operation settings or algorithm parameters.

At run time, all output messages from the DA process are exported to the central DAQ/ECS log system (described in Chapter 11) for online operator display and archival. Health of the process is also monitored constantly by the ECS, and return error code checked upon exit.

While running, the DA may publish its results to the experiment Data Quality Monitoring (DQM) framework (see Chapter 23) for feedback to the operator, graphical display, consistency and quality checks, or reuse for monitored data reduction in the DQM agents.

The DA is notified the end of the run, and then proceeds to final post-processing and results saving. The control system checks that the DA completes its tasks within the required time, and aborts the process if necessary. Allowed DA end-of-run duration is kept short for the global runs (typically less than one minute), to minimize data taking dead-time. Whenever possible, demanding computation tasks exceeding this threshold are performed offline where there is no such constraint.

The DA can save persistent files to a configuration database (local or central) in case results are to be reused in the DAQ (e.g. for a further DA run, for other online processes, for FEE configuration, etc). Calibration results are also exported to the File Exchange Server (see Section 17.4), which is the system used to publish data from DAQ to the other components. The DA result files may as well be reused in HLTor DCS. Most importantly, the results are collected offline by the Shuttle

framework (see Section 17.5), where the output data is post-processed and archived in the Offline Condition Database.

22.4 DA framework implementation

The framework relies on two main components:

- a programming interface for the DAs to interract with the outside world.
- a launch mechanism to start and control the DAs at runtime, so that they run when appropriate.

We describe below the DA Framework version 1.

22.4.1 DA interface API

The DAs are implemented in the ***AliROOT*** framework so that the calibration processing code is shared with the offline code and components reused. However, the DAQ provides the API for the detector code input and output mechanisms. The main loop (subscription to events and their processing) of each DA program is implemented in the detector code, and not provided by the framework library. The framework distribution provides some examples of DA skeletons with a main loop reading and processing events.

The API described in the file ***daqDA.h*** provides the means to read configuration files, store and export results, and get some runtime information. In particular, it provides:

- functions to read and write data from / to the DAQ detector database (Section 4.4.6): ***daqDA_DB_getFile()*** and ***daqDA_DB_storeFile()***.
- a function to export result files to the File Exchange Server (Section 17.4): ***daqDA_FES_storeFile()***.
- a function to check if the DA is requested to terminate ***daqDA_checkShutdown()***. If this is the case, the DA should exit within promptly (no more than 30-60 seconds, or may be killed).
- functions to retrieve the ECS loop parameters in case of a calibration requiring multiple iterations with different settings: ***daqDA_ECS_getCurrentIteration()*** and ***daqDA_ECS_getTotalIteration()***.
- a function to store some results locally (e.g. partial results, or output to be used in another DA running locally). This is useful to avoid storing large files in the database in the case they don't need to be used on other hosts. It can be done with ***daqDA_localDB_storeFile()***. Files stored there are then available at runtime in the directory ***$DAQ_DETDB_LOCAL***.
- a function to convert a trigger class name into a trigger class id, which then may be used in the monitoring API (Chapter 5) to request events on a specific trigger class. This function is named ***daqDA_getClassIdFromName()***.

Output messages should simply be written to ***stdout***: they are then redirected to the ***infologger*** log repository.

The DAs use the monitoring API described in Chapter 5 to retrieve events at runtime, either from a file (LDC DA) or from the online data stream (MON DA).

The DAs may export data to AMORE (DQM framework, see Chapter 23) for interactive display and results checking. It usually involves including the file ***AmoreDA.h*** provided by the AMORE distribution, and linking with the library ***libAmoreDA.a***.

For runtime stability, DAs are required to be provided by detector teams as static executables having no dependency. It allows DAs of different detectors relying on different AliROOT versions to coexist on the same machine.

Note that the DA build mechanism is provided by AliROOT and out of control from the DAQ. For reference, DAs may be build with the following ***make*** targets in ***$ALICE_ROOT***:

- `make daqDA-DETCODE-NAME`: builds a DA executable
- `make daqDA-DETCODE-rpm`: builds and package in RPM files all the DAs for a given detector detector.

The ***AMORE*** environment variable should be defined and pointing to the AMORE installation directory in order for AliROOT to link the DAs with AMORE support.

Some mandatory information should be provided in the RPM description tag. AliROOT takes care of packaging the DA in RPM. However the documentation fields must be completed in the DA source code (first comment of the file, `/* ... */`, with the syntax `KEYWORD: VALUE`).

They are automatically extracted from the source code and copied in the RPM description. This information is used to validate and check the packages before deployment at the experimental area.

The following fields should be filled in:

- ***Contact*** : E-mail of package responsibles (development and runtime)
- ***Link*** : External link to additional DA documentation, including some raw data test files and necessary input configuration files.
- ***Reference Run*** : The run number of a reference run made at the experimental area in the appropriate conditions for this DA, with recording to CASTOR enabled. This will be used to validate the DA. Such run should use a single LDC for a DA running on LDCs. It should contain a realistic number of events.
- ***Run type*** : The ECS run type(s) in which this DA should be running. (`PHYSICS` for global runs, otherwise the detector specific ECS run type in standalone operation).
- ***DA type*** : LDC or MON (for DAs running respectively on LDC at end of run or on Monitoring node during the run)
- ***Number of events needed*** : The number of events needed to produce adequate results.
- ***Input files*** : Names of files needed to run the DA. (these are the files stored

in the DAQ detector configuration database).

- ***Output files***: Names of files produced by this DA (including local files, FXS files, detDB files)
- ***Trigger types used***: Trigger type of events used by this DA.

These fields may be checked after the RPM is created with the command `rpm -qip daqDA-....rpm`

22.4.2 DA control mechanisms

The launching mechanism depends on the DA type, LDC or MON.

In both cases, the DA executable is called with command line arguments giving the monitoring data source(s) where to get the events (the name of an online monitoring data source for a MON DA, or a list of local data files for a LDC DA). No other command line arguments are allowed to be given to a DA executable. All configuration parameters should be read by the DA from the configuration database.

The starting of the DA processes is done through the ***LAUNCHER*** facility (identified as such in the ***infoLogger*** messages) implemented in the ***runControl*** package file named ***da.c***.

The DA rely on a set of runtime parameters that must always be defined, because used by some of the I/O functions.

22.4.2.1 Runtime parameters

The following runtime parameters, defined as environment variables, are necessary in order to use the DA I/O functions of the DA library. Most of them are provided by the DATE standard setup procedure and completed by calling launcher process for variable items.

- ***DATE_DETECTOR_CODE***, ***DATE_RUN_NUMBER***, ***DATE_ROLE_NAME***, ***DATE_FES_DB***, ***DATE_FES_PATH***: access parameters to the File Exchange Server (see Section 17.4).
- ***DATE_RUN_NUMBER***, ***DAQ_DB_LOGBOOK***: access to the logbook, e.g. to retrieve information on trigger classes to filter on them.
- ***DAQDALIB_PATH*** :path to the installation directory of the DAQ DA library, typically ***/opt/daqDA-lib*** . This is needed to use the I/O functions (database and File Exchange Server).
- ***AMORE_DB_MYSQL_...*** : definition of the access parameters to the AMORE database, in case some data shall be exported to the DQM. One may also need to define ***AMORE_DA_NAME*** used to identify the target AMORE output table (set to ***DATE_ROLE_NAME*** by default).
- ***DAQ_DETDB_LOCAL***: location of a local directory that may be used as a persistent location to store data. Usually, it is set to ***${DATE_SITE}/${DATE_ROLE_NAME}/db***.

At the moment, only the ***DATE_DETECTOR_CODE*** is not set automatically (because specific to the DA) and should be defined in a wrapper script.

It might also be necessary to define ***ROOTSYS*** to a dummy string, e.g. ***NULL*** or some ROOT routines called in some DA may crash if this variable is undefined.

When testing a DA executable manually, one may define the variable ***DAQDA_TEST_DIR*** in order to use a dummy configuration database and File Exchange Server. The directory pointed by this variable will be used to read configuration files from there, and to export result files (as to the File Exchange Server). The ***DATE_DETECTOR_CODE***, ***DATE_ROLE_NAME***, ***DATE_RUN_NUMBER*** shall still be defined in this situation.

Similarly, if a full AMORE setup is not available in the test environment, it is possible to define ***AMORE_NO_DB*** to ***true*** and an associated directory ***AMORE_NO_DB_DIR*** as a fake database.

At run time, the DAQ starts the DA in a directory named ***DATE_SITE_WORKING_DIR***, and typically located in ***${DATE_SITE}/${DATE_ROLE_NAME}/PARTITION-DETECTOR/work_DA-...***

This directory is meant to be a temporary working directory. Content may be cleared after the DA exits. Persistent files should be saved to the configuration database or to the ***DAQ_DETDB_LOCAL*** local directory.

22.4.2.2 LDC DA launching

LDC DAs are started by the ECS at the end of the run, after data taking is finished. The ECS looks in the DAQ database (***Files*** section in ***editDb***) for a script based on the corresponding hardcoded ***SYNCHRONOUS*** action name in the ECS SMI file defining the state machine of this detector. Such database files (***host*** field left empty) are named for example `/ECS/FMD/FMD_COMPUTE_GAIN` or `/ECS/TPC/TPC_PULSER_DA`.

The content of this file should be an executable (SHELL or other) script, defining necessary variables and launching the corresponding DA executable (passing to it the provided arguments), e.g.

```
export DATE_DETECTOR_CODE=TPC

/opt/daqDA-TPC-PULSER/TPCPULSERda.exe $@
```

The script is copied on the machine and executed at end of run, after the DAQ has completed the run and the local data files are available.

22.4.2.3 MON DA launching

MON DAs are started by the ***runControl*** following the configuration file named after the ***runControl***: ***/das/RCNAME.config*** where RCNAME is the ***runControl*** name (and ***host*** field left empty). Example names of such files are `/das/ALLPHYSICS_1.config` for the partition PHYSICS_1 or `/das/FMD.config` for a standalone FMD detector operation.

The file syntax is the following (as implemented in the ***runControl*** package): every line contains a sequence of fields: DA name, name of the ***MON*** machine (DATE role) where the DA shall be executed, name of the script to be executed (DATE database ***File*** entry in ***editDb***, to be named ***/das/scripts/...*** (and ***host*** field left empty), e.g. `/das/scripts/FMD-Base`), input parameter to be given to

the script (the name of a valid monitoring data source, e.g. `^FMD` to monitor FMD data, or `@*` to monitor full events from all GDCs), and a list of tags (i.e. run types) to activate the DA (typically, the name of the active ***runControl*** configuration, as saved from the ***runControlHI***, e.g. `DEFAULT` or `PEDESTAL`).

An example entry looks like:

```
DA-FMD-BASE mon-DA-FMD-0 FMD-Base ^FMD DEFAULT
```

In the case of a global run, it may be necessary to check (by accessing the logbook) in the DA script if the corresponding detector belongs to the run, because at the moment the ***runControl*** starts the MON DA scripts based on the configuration name. Despite some filtering is done on the monitoring data source, this might not be enough if the DA is not using monitoring by detector.

The content of the script itself is similar to the LDC DA script described above, and is a simple wrapper to the DA executable.

The name and host of the MON roles have to be defined in the DATE configuration database. A simple ***ROLE*** entry with ***TOPLEVEL*** set to `0`, with an associated ***IPC*** memory bank of size `-1` for a single ***control*** pattern is sufficient.

Note that the data source and monitoring policy should be chosen with care for the MON DAs, or they may not be able to receive at runtime the necessary data.

The MON DA is started at the start of the run, and is left active until the data taking is over. At that point, it receives a `QUIT` command from the ***runControl*** launcher, and the DA process should exit after a reasonnable time (time for post-processing and exporting result files should be kept under control).

Part V

Data Quality Monitoring

***November* 2010**

ALICE DAQ Project

AMORE

23

Automatic MOnitoRing Environment (AMORE)

The quality of the acquired data evolves over time depending on the status of the detectors, its components and the operating environment. To use the valuable bandwidth and the short data-taking period in an optimal way, the quality of the data being actually recorded must be continuously monitored. Data Quality Monitoring involves the online gathering of data, their analysis by user-defined algorithms and the storage and visualization of the produced monitoring information. This chapter describes the data quality monitoring framework AMORE which is based on the DATE monitoring library in conjunction with ROOT. It is a distributed and modular system, where each detector team develops one or several plug-ins on top of the framework. This chapter also describes the generic modules that leverage the development effort of the detectors teams, such as the Generic GUI and the Quality Assurance agent.

23.1 Architecture

AMORE (Automatic MOnitoRing Environment) is the Data Quality Monitoring (DQM) framework for ALICE. It is a flexible and modular software framework which is used to analyze data samples and produce and visualize monitoring results. The data samples, ie. events or subevents, are coming either from LDCs or from GDCs. Raw data files can also be used as data source. AMORE is founded on the widely-used data analysis framework ROOT and uses the DATE monitoring library (see Figure 23.1). In case the same analysis is needed online and offline, the use of the ALICE Off-line framework for simulation, reconstruction and analysis (AliRoot) is recommended at the level of the module.

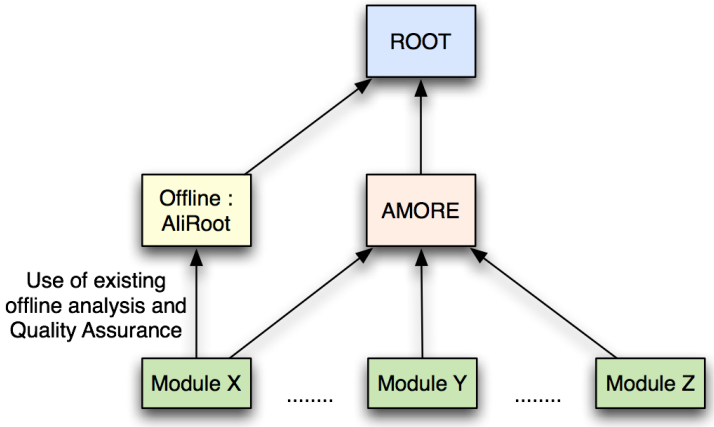

Figure 23.1 Schema of the main dependencies of AMORE.

23.1.1 Overview

AMORE is based on a publisher-subscriber paradigm (see Figure 23.2) where a large number of processes, called agents, execute detector-specific decoding and analysis on raw data samples and publish their results in a ***pool***. Clients can then connect to the pool and visualize the monitoring results through a dedicated user interface. The serialization of the published objects, which occurs on the publisher side before the actual storage in the database, is handled by the facilities provided by ROOT. The only direct communication between publishers and clients consists of notifications by means of DIM. The notifications coming from the outside world, especially from the Experiment Control System (ECS), use the same technology.

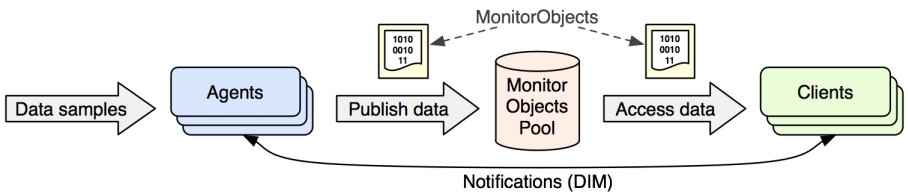

Figure 23.2 The publisher-subscriber paradigm in AMORE.

23.1.2 MonitorObjects

As illustrated in Figure 23.2, the monitoring results are encapsulated in so-called ***MonitorObjects*** that essentially contain additional metadata allowing a proper and coherent handling by the framework (see Section 23.5.1.1 for details).

23.1.3 AMORE taxonomy

AMORE uses a plug-in architecture to avoid any framework's dependency on users' code. The plug-in mechanism is implemented through the ROOT reflection feature. Users, usually detector teams, develop ***modules*** that are typically split into two main parts corresponding to the publishing and the subscribing sides of the framework (see Figure 23.3). The ***modules*** are built into dynamic libraries that are loaded at runtime by the framework if, and when, it is needed. There are typically 4 libraries produced (stacked boxes on the left), one for each package (the four boxes at the bottom of the figure): Common, Publisher, Subscriber and UI. A module's publisher can be instantiated as many times as needed, to collect more statistics for instance, each instance being called ***agent***. The same is true for the subscriber part of the module; we call these instances ***clients*** or ***GUI***. Note that a module can contain several publisher and subscriber classes (not shown in the Figure).

The shared libraries produced by the detector's code are stored in a special directory called ***amoreSite***. Its location is defined in the variable $AMORE_SITE. The directory also contains the file AMORE.params where the database credentials are stored.

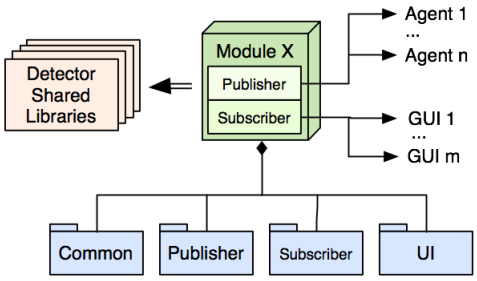

Figure 23.3 Description of a module.

23.1.4 Publishers

The publishers must extend the class PublisherModule. They are meant to analyze the raw data they receive and to publish results under the form of MonitorObjects. However, not all publishers directly do the work. The Quality Assurance (QA) module for example delegates the processing to the AliRoot QA framework. The High Level Trigger (HLT) module retrieves the objects from a private network and publish them in the pool. The module amoreDB simply publishes data retrieved from a database, including from the AMORE pool itself. One should avoid, if possible, to duplicate code but rather choose to delegate the processing to existing frameworks and libraries.

23.1.5 Clients

The clients must extend the class VisualModule. They mainly consist of a ROOT GUI in which some MonitorObjects are displayed. Clients are usually tied to the corresponding publisher.

There is no limit on what a client can do with the objects it retrieves, but it is in general not a good idea to deeply modify them. Indeed, these modifications will not be saved in the database and therefore will be lost to other clients. As a rule of thumb, only do “cosmetic” changes in the GUI. Even adding lines or boxes to an histogram shouldn’t be part of the client in most cases.

23.2 Database

23.2.1 Overview

The pool is implemented as a database. The open-source MySQL system was chosen as it proved to be reliable, performant and light-weight. Figure 23.4 shows a rough schema of the database and the detailed description of the tables follows.

The database is used not only to keep the data published by the agents, but also to store the configuration of AMORE as a system. This includes information about the agents such as the machine where they can run and to which detector they belong (***amoreconfig*** table) as well as the optional configuration files. When a new agent is created in the system, a row is added to amoreconfig table. The table where published data will be stored is created or recreated when the agent is started.

23.2.2 Archives

Former ***versions*** of the MonitorObjects can be kept. This is the case for the recent values of the objects as long as the data table of the agent doesn’t exceed a certain size. This size is specified in the table ***amoreconfig*** (see below the tables descriptions for details). To decide what objects must be removed, a First In First Out policy is applied.

Snapshots of the MonitorObjects can also be archived for a longer term on user’s request or automatically, at SOR, EOR and every hour during a run. These data are stored in a table that is pretty similar to the agent’s data table but whose name ends with “_archives”. The objects will stay there for a week before being deleted, although they could remain indefinitely if they are marked as ***permanent***. For details on the process which takes care of archiving, please refer to Section 23.4.3.

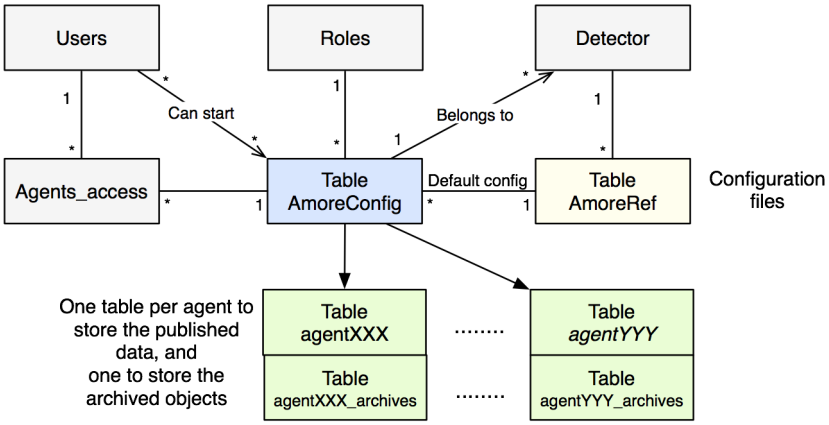

Figure 23.4 Schema of the database.

23.2.3 Tables descriptions

The main tables used by AMORE are described in this section.

amoreconfig

Table 23.1 ***amoreconfig*** List of the agents.

Field	Description
host	Machine, specified by its role name, where the agent is allowed to run
agentname	Name of the agent
detector	Detector to which the agent belongs (3 letter code)
source	Default data source (format: see Chapter 5)
dimnode	Dim server
poolnode	Database server
defaultmodule	Default module (in case the library contains several modules)
configfile	Default configuration file name
fifo_size	Size of the data table in Bytes
image_generation	Flag indicating whether objects' images must be produced
production	Flag indicating whether this agent must always be running
extra_flags	Any additional flag to pass to the agent at startup

amoreref

Table 23.2 ***amoreref*** Configuration files table.

Field	Description
detector	Detector to which the file belongs (3 letter code)
filename	Name of the file
data	The file itself
updatetime	Time of last update of the file

Agents tables

Each agent has a table where its ***MonitorObjects*** are stored (one version of each object per row). It is the "pool" where data transits between the publisher (agent) and the subscriber (client). The name of such a table is the name of the agent. Its size shouldn't exceed the size specified in amoreconfig->fifo_size. This is enforced within the framework. However, its minimal size will be the sum of the size of all the agent's ***MonitorObjects***. If fifo_size is big enough, several versions of each ***MonitorObject*** will be kept in the table, the oldest objects being removed first.

Table 23.3 Agents tables fields description

Field	Description
moname	Name of the MonitorObject
updatetime	Time when the object has been stored
data	The serialized MonitorObject
size	Size of the data
run	Run active when the object was last updated
image	Summary image

latest_values

Pointers to the latest version (i.e. last published) of each MonitorObject of each agent. By querying it, one gets the time of the last update of a given object. This table speeds up the numerous queries made by the clients by avoiding searching the large data tables and by being stored in RAM (in-memory table). It is kept up to date by triggers on the agent's data tables.

Table 23.4 ***latest_values*** table

Field	Description
agentname	The name of the agent publishing the object
moname	The name of the object
updatetime	Time when the object was published

Archives tables

Each agent has an archive table where temporary and permanent copies of ***`MonitorObjects`*** are stored. The naming convention for these tables is: <agent's name>_archives. The structure is very similar to the agent's data table.

Table 23.5 Archives tables

Field	Description
moname	Name of the MonitorObject
updatetime	Time when the object has been stored
data	The serialized MonitorObject
size	Size of the object
permanent	Flag to indicate if the object's archive is permanent or not
run	Run active when the object was last updated
image	Image of the object (if generated)
description	Description of the archived object

globals

Global variables used by AMORE. For example, the version of the current database schema is stored here.

Table 23.6 ***globals*** table

Field	Description
variable	Variable name
value	Value

Roles

List of roles, i.e. names given to machines where agents can run.

Table 23.7 ***roles*** table

Field	Description
name	Role's name
host	Hostname of the machine

Users

List the users allowed in the system.

Table 23.8 ***users*** table

Field	Description
user	User's name

Agents_access

Agents can only be manipulated (started or stopped) by certain users. This table contains, for each agent, the user(s) allowed to do so.

Table 23.9 ***agents_access*** table

Field	Description
agentname	Agent's name, foreign key to amore-config
user	Refers to a user in the table "Users"

Agents_details

The clients might need to know what an agent is publishing. They use DIM to get a list of the objects being published, but this is often not enough. For example, the Generic GUI needs to know the quality or the type of an object without actually loading it from the database. The agents_details table is used for this purpose. It contains, for each agent, a long string providing details on the objects it publishes. The format is:

<string> = [<object_name>#<type>#<quality>#<expert/shifter>:]*.

Table 23.10 ***agents_details*** table

Field	Description
agent	Agent's name
details	Details string

23.3 Application flow

The agents and the clients are implemented as finite state machines (FSM). The framework binaries, `amoreAgent` for the agents and `amore` for the clients, drive the FSM and call the user's module methods at certain steps.

The two FSMs are completely decoupled and the notion of monitor cycle is different on both sides. Thanks to this decoupling, a slow process doesn't affect the others.

23.3.1 Agents and clients Finite State Machines

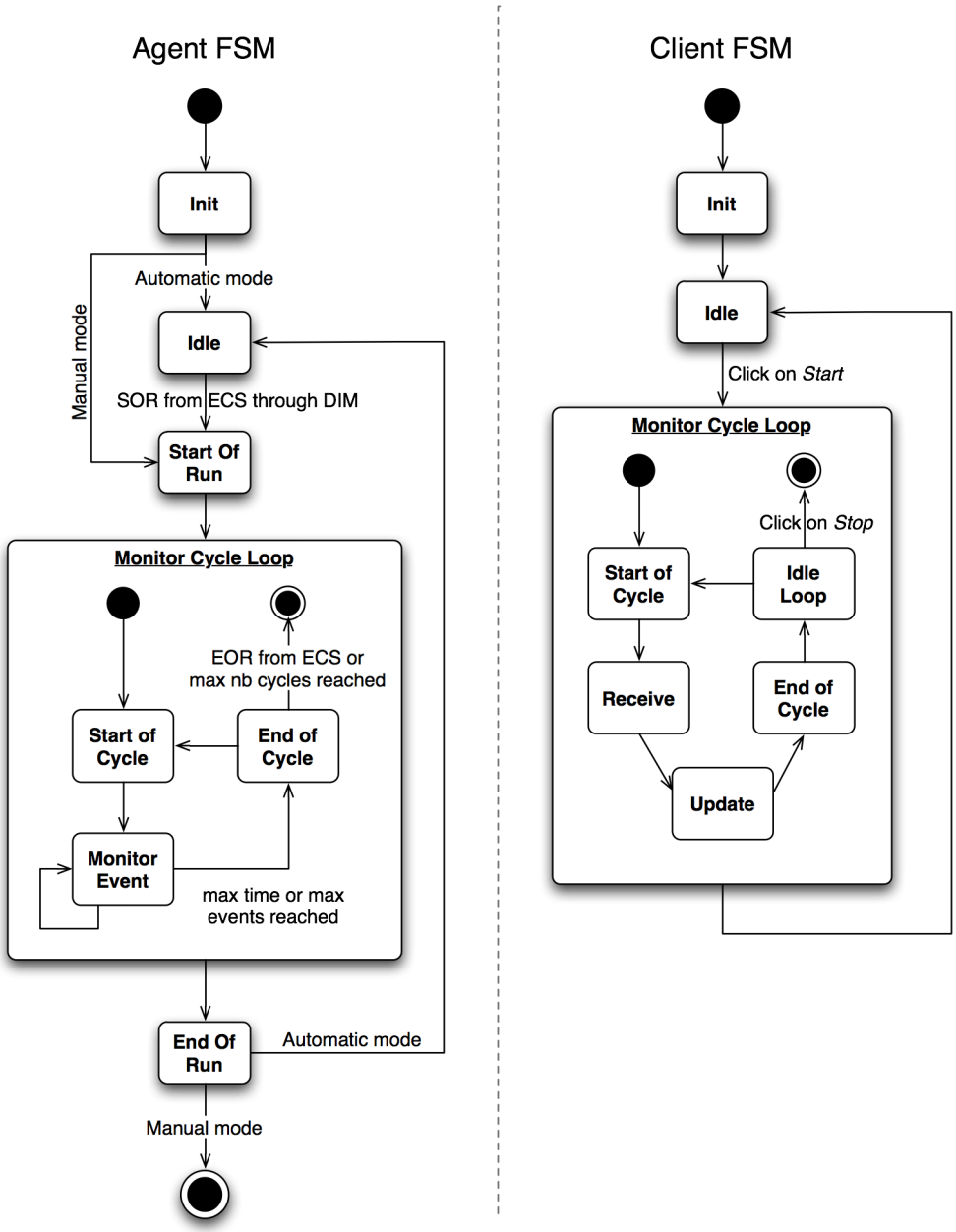

Figure 23.5 Left: the publisher Finite State Machine. Right: the client Finite State Machine.

23.3.2 Initialization

AMORE is a pluggable software where the detectors' libraries are loaded and the proper class is initialized at runtime. When starting an agent with the binary `amoreAgent`, three main steps occur:

1. Look up amoreconfig for the agent, check that it exists and runs on the correct machine. Retrieve information about the agent (detector, class name,...)
2. Load the detector library

3. Instantiate the class

No configuration table exists for the clients, such as ***amoreconfig*** for the agents. One must specify the detector name and the module name when starting the client. Therefore the startup is simpler and skips the first step described above.

23.3.3 Agents and clients inheritance and methods calls sequences

All the publisher modules must inherit from ***`PublisherModule`***. Reciprocally, the subscriber modules must inherit from ***`SubscriberModule`***. Their various methods will then be called by the framework depending on its current state. Figure 23.6 shows the detailed sequence of methods calls made by the framework for the agents and the clients.

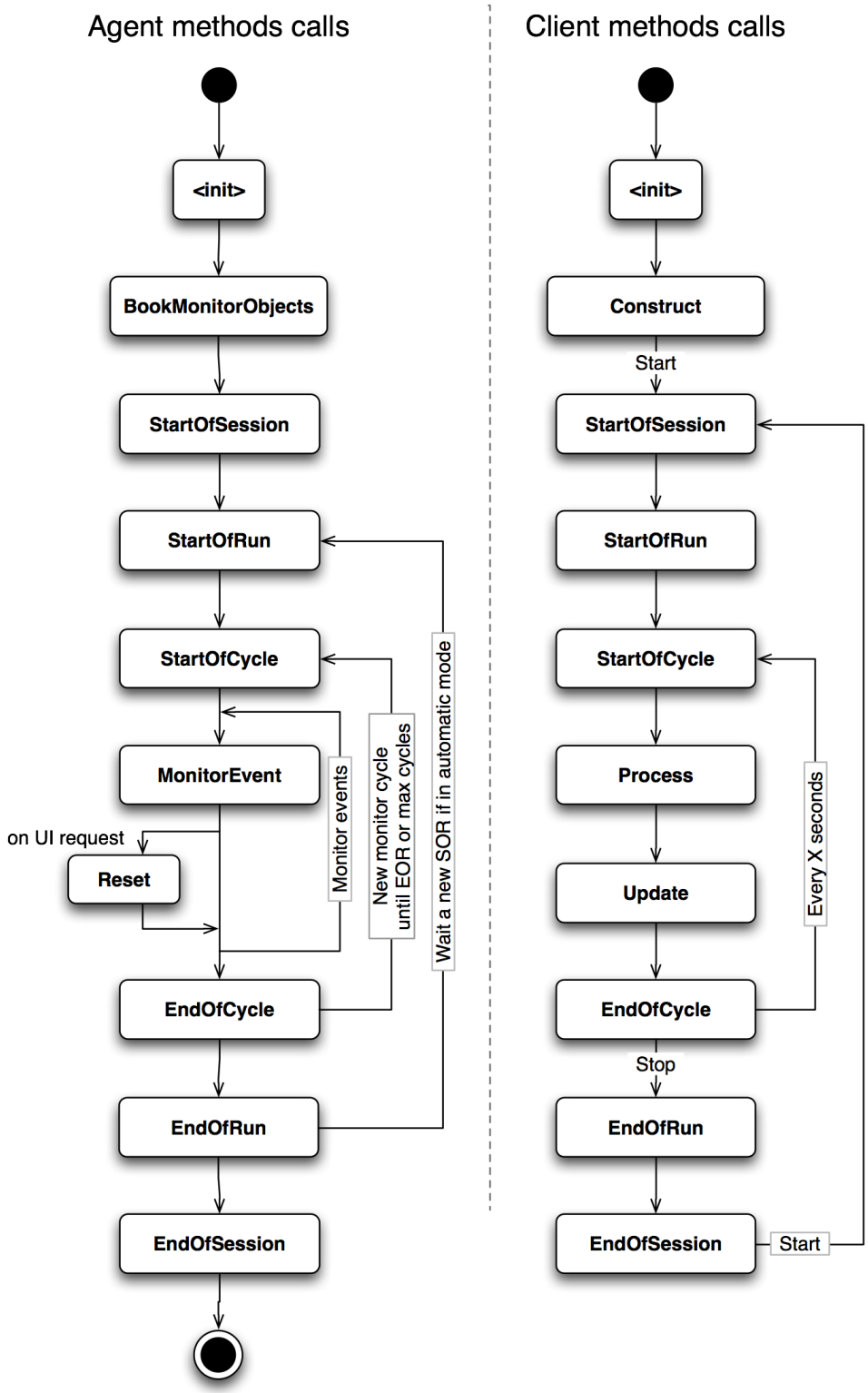

Figure 23.6 Sequence of methods calls on the agent and the client modules.

23.4 Features details

23.4.1 Quality

Each ***MonitorObject*** has a quality associated with it. This quality is stored within the object and can take different values :

- ***kNULLFLAG***: no quality. It can be used for objects such as error message sent to the client or intermediate objects needed to build another object.
- ***kINFO***: Good quality.
- ***kWARNING***: Object should be checked.
- ***kERROR***: Object is clearly out of the reference, there is an error.
- ***kFATAL***: Object is so incorrect that measures must be undertaken quickly.

To set the quality of an object, usually at End Of Cycle, use the method `SetQuality(flag)` of the ***MonitorObject*** class.

By default, if not explicitly set by the publisher, the quality of an object is ***kFatal***.

23.4.2 Expert/Shifter MonitorObjects

Each ***MonitorObject*** can be classified as shifter or expert, the former representing a subset of the latter.

In order to publish a ***MonitorObject*** as shifter use the following example:

Publish(fMO, "MOname", "MOtitle", MonitorObject::kSHIFTER)

and as expert:

Publish(fMO, "MOname", "MOtitle", MonitorObject::kEXPERT)

which is the default if nothing is specified.

There are two ways of exploiting such a functionality.

1. Start the agent with the option "**`-f S`**" . In this way, only the shifter objects will be published in the database and available in the GUI.
2. Start the agent with the usual options and filter the histograms in the client to show only the shifter ones (check for each ***MonitorObject*** its expert flag). In this way, all the objects will be published and available in the Logbook and in the database, but only the shifter objects will be displayed in the GUI.

23.4.3 Archiver and FIFO

23.4.3.1 Purpose

When an expert is called by an operator, he might want to check and study the objects even though the run has stopped. Therefore, snapshots of the

MonitorObjects must be saved, if not permanently at least for a week or longer. The archiver is meant to give a way to archive and recover interesting ***MonitorObjects*** for further study. The archiver must always be running and available to receive new requests. It also performs a clean up every night to erase temporary archives older than 7 days.

In addition to these mid and long-term archives, it might also be interesting to keep a very detailed short-term history to discover when a problem occurred or started. This is done through the so-called ***FIFO***, which is directly implemented within the database. It consists in keeping former versions of the objects in a First In First Out queue (see Figure 23.7).

This chapter describes the general design of both features. For information about how to operate the archiver, please refer to the ALICE DAQ WIKI.

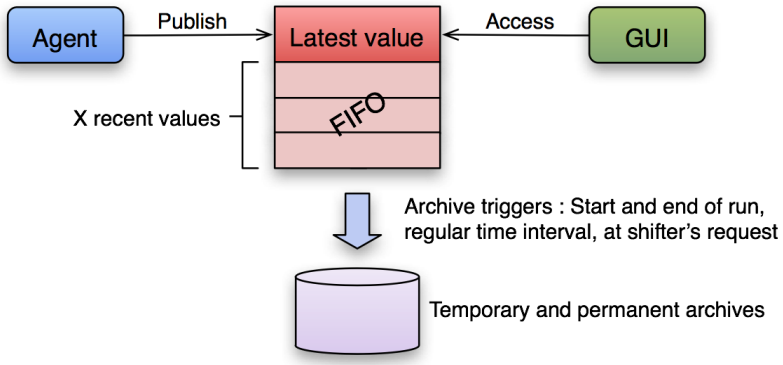

Figure 23.7 The archiving system in AMORE.

23.4.3.2 Implementation of the archiver

The archiver package in AMORE depends only on the core package. It produces a standalone binary that loads one or more ***ArchiverModule(s)***. It uses DIM to receive users' requests and SOR and EOR notifications; agents declare themselves automatically to the archiver at SOR and EOR. The tasks to be executed by the archiver are stored in an ordered queue. The complete class diagram of the package is shown on Figure 23.8. People interested in further details are encouraged to read the sources.

The archiver uses plug-ins, called ***ArchiverModules,*** to do the actual work of archiving and cleaning up. The configuration of the archiver is done by means of a configuration file (see Listing 23.1 for an example). The plug-in ***StdArchiverModule*** is currently used for both the cleaning and the archiving. It uses stored procedures to execute the archiving and to make permanent archives.

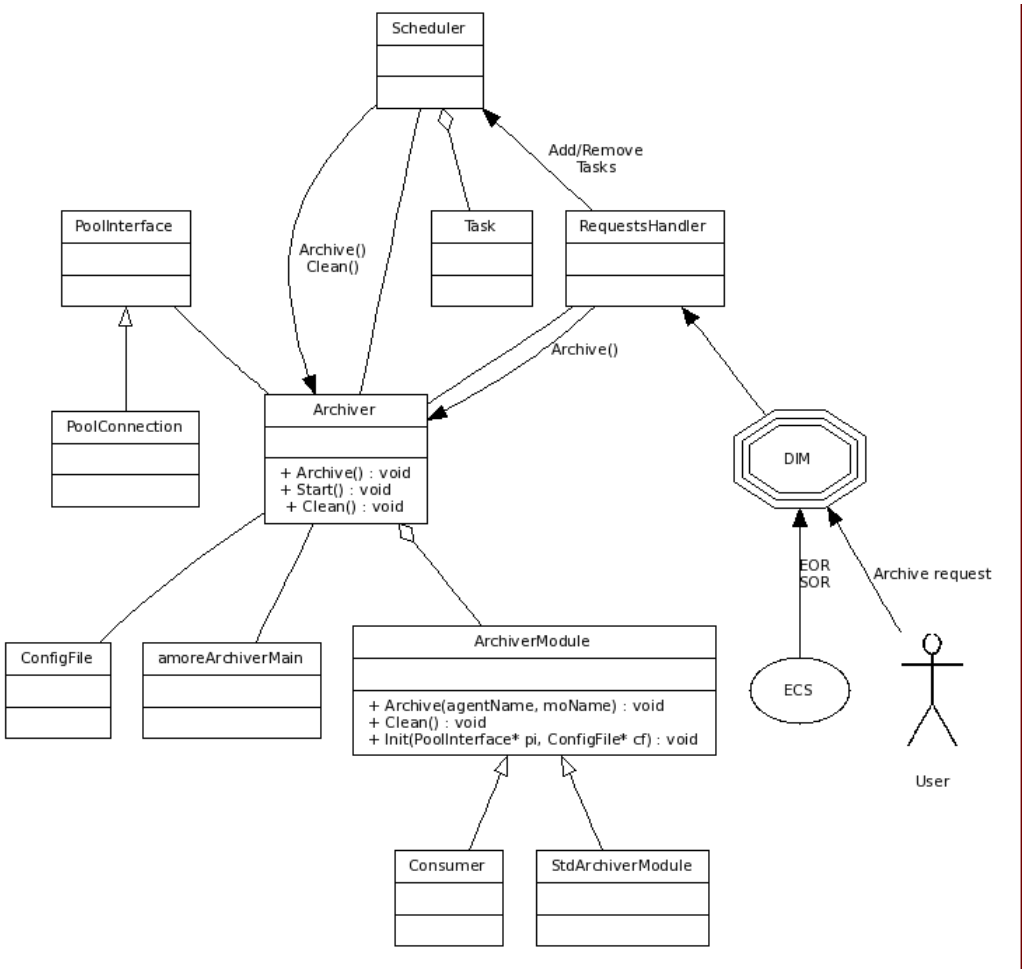

Figure 23.8 Class diagram (including some interaction information) of the package ***archiver.***

Listing 23.1 Example of a configuration file for the archiver

```
1:  # Archiver config file
2:  # Define the archiver module to use to archive
3:  archiver_module_archive amore::archiver::StdArchiverModule
4:  # Define the archiver module to use to clean up obsolete archives
5:  archiver_module_clean amore::archiver::StdArchiverModule
6:  # The number of days an archive is kept before it is cleared
7:  std_archiver_obsolescence 7
```

Table 23.11 DIM commands

DIM commands	Description
amore / archiver / archive	Trigger the archiving. Parameter is <name of agent>::[<name of object>] If no object specified, all objects are archived.
amore / archiver / makePer-manent	Make an archive permanent. The parameter is <name of agent>::<name of object>::<timestamp>

Table 23.11 DIM commands

DIM commands	Description
amore / archiver / agentSOR	The command that agents must use to declare themselves as alive at SOR or when they are started Parameter is <name of agent>::<run number>
amore / archiver / agentEOR	Define the command that agents must use to declare themselves as alive at EOR or when they stop Parameter is <name of agent>::<run number>
amore / archiver / printTasks	Force the archiver to print a list of the tasks currently in the system.

23.4.3.3 Implementation of the FIFO

The recent versions of the objects are kept in the data table (named after the agent's name). The size of the fifo, ie. the size of the table, is defined in the table ***amoreconfig*** for every agent. A table size smaller than the sum of the size of all the published objects results in a default behaviour where only the latest version of each object is kept.

The PoolConnection in the publisher takes care of determining if the maximum size is exceeded and, if so, to take the appropriate action, namely to delete the oldest objects

23.4.3.4 Access to the archives

The Logbook gives full access to the archives and the FIFO with possibilities of creating archives and making them permanent.

An archive request can also be sent from the Generic GUI.

23.4.4 Access Rights

One or more users from the table ***users*** are associated to every agent. They represent the users allowed to start, stop and restart the agent. If the user "all" is associated to the agent, all users present in the table are allowed to start, stop and restart the agent.

When creating a new agent, the operator must specify the allowed user. If the user is not present in the table or the field is left empty, the user "all" will be associated to the agent.

23.4.5 ECS-AMORE interaction

23.4.5.1 Motivation

Agents must be able to react to the runs' Start Of Run (SOR) and End Of Run (EOR) in order to re-initialize themselves accordingly and possibly to reset certain ***`MonitorObjects`***.

23.4.5.2 Implementation

The class ***RunControl*** (core) and its sub-class ***RunSequence*** (publisher) inherit from ***DimClient*** and subscribe to the SOR and EOR Dim info provided by the Logbook daemon. One command exists for each runControl, ie. for each detector and for each partition. When a standalone run is started for detector XXX, the corresponding command is received. In the case of a partition, the info for the partition is updated with the new run number, as well as the info of each included detector.

The ***RunSequence*** constructors takes as argument the name of the ***RunControl*** to which it must listen. It is therefore the responsibility of the publisher to identify the runControl. It does so by using the detector code attached to the agent and/or the partition given at startup (parameter **-p**).

23.4.6 Logbook usage

23.4.6.1 Motivation

The Logbook contains a large number of metadata about runs. It is thus a valuable source of information that AMORE needs to access. Moreover, AMORE takes advantage of the Logbook web interface to make statistics and objects available to the users worldwide.

23.4.6.2 Usages

At SOR, AMORE uses the Logbook to retrieve information about the run, such as its type (PHYSICS, CALIBRATION,...) or the detectors it includes.

At SOR, the framework also stores data in the Logbook in order to have it listed in the corresponding page of the web interface (see Table 23.12).

Table 23.12 Data passed to the Logbook at SOR

Field	Description
Run number	Which run the agent is running for.
Detector	The code of the detector for which the agent is running
Agent's name	Name of the agent
Version of the module	Version number of the module's libraries
Configuration	The configuration file specified at startup or by default (if any)

Every agent has one ***summary image*** that is generated in its method `GetSummaryImage()`. During a run, the agent regularly stores it in the Logbook.The update interval is currently set to 2 minutes and can be changed in the class ***ImagePublisher***.

Finally, at every end of cycle and at EOR, the agent inserts the following statistics in the Logbook:

- Number of objects
- Total number of objects published (all versions of all objects)
- Total number of bytes published
- Average CPU time per cycle
- Average real time per cycle

23.4.7 Multi thread image production

The image production is the functionality that makes the images of the ***MonitorObjects*** available in the Logbook. In order to enable it, the flag `image_generation` of the agent in the table ***amoreconfig*** must be set to 1.

The image generation can perform in single or multi-thread mode. In order to run it in multi-thread mode, the option "**`-i`**" must be specified in the launching parameters of the agent. It permits to split the data quality monitoring process in two independent threads, one for the analysis and one for the image production.

23.5 Application Programming Interface (API)

This section is dedicated to the API of AMORE. The classes and methods described below are grouped by package. Only the public interface is presented. For details on how to develop a new module please refer to the document "Modules' developer's guide" available on the AMORE website (http://ph-dep-aid.web.cern.ch/ph-dep-aid/amore/).

23.5.1 Core

23.5.1.1 MonitorObject

Any object published in AMORE is encapsulated in a ***MonitorObject*** data structure. To ensure type safety, a templated class hierarchy is used. AMORE provides classes derived from ***MonitorObject*** to handle scalars, histograms (1D and 2D) or ***TObjects***, for example. Several of them are templated to specify the type of histograms or scalars being encapsulated.

The ***MonitorObject*** abstract class contains a set of members that can be accessed (read and/or write). The accessors follow the naming convention (for a member ***fMyMember***: ***GetMyMember***)

Table 23.13 Members of the class MonitorObject

Member	Description
Name (read-only)	Name of the object, used as a unique id. Can contain only standard characters plus slashes (“/”), but no spaces.
Title	Title of the object.
Description	Description of the object.
UpdateTime (read-only)	Last time the object was published in the database.
Quality	Quality of the object. Variable of type ***QualityFlag_t***. It can take 1 of 5 values: ***kNULLFLAG*** (no quality), ***kINFO***, ***kWARNING***, ***kERROR*** and ***kFATAL***. See Chapter 23.4.1 for details
ExpertFlag	Specifies if the object is for shifter and/or expert. Variable of type ***ExpertFlag_t***. It can take 1 of 2 values: ***kEXPERT*** and ***kSHIFTER***.
DefaultDrawOption	Default draw option to use when drawing this object.
DisplayHint	Hints about how to display the object in the best way. This is highly type-dependent. At present, the options ‘logx’ and ‘logy’ are accepted for any type of histogram.

The following methods are available for each type of ***MonitorObject***. In addition, the interface of ***MonitorObjects*** subclasses contain type-specific methods, e.g. `Fill(...)` for ***MonitorObjectHisto***. Please have a look directly at the header file core/MonitorObject.h to know what methods exist for each type.

Reset

Synopsis

```
#include "MonitorObject.h"

void Reset()
```

Description The Reset method must be called to reset an object.

Draw

Synopsis

```
#include "MonitorObject.h"

void Draw(Option_t* option = "")
```

Description Draw the object on the current pad.

23.5.1.2 Run

A class representing a run. At start of run, the publisher code receives an object of this type.

RunType

Synopsis

```
#include "Run.h"

string RunType()
```

Description Return the run type.

RunNumber

Synopsis

```
#include "Run.h"

RunNumberType RunNumber()
```

Description Returns the run number.

RunDuration

Synopsis

```
#include "Run.h"

int RunDuration()
```

Description Returns the number of minutes elapsed since this objects has been created. This can be either the number of minutes since the start of the run if we received the SOR or the number of minutes since this object was created in case we started the agent after the SOR. The number of minutes is rounded down.

23.5.1.3 ConfigFile

This class represents a configuration file (stored in the database or in the file system) and provides methods to access its content. It tries to parse the file during its initialization. If the format is not recognized (pairs separated by spaces) the user can still use the object to retrieve its content and make its own parsing. The user gets a reference to a ***ConfigFile*** if one is specified at startup; he can also instantiates such an object at anytime. Please refer to the ***Modules' developer's guide*** for more details on how to use configuration files.

InitWithFile

Synopsis
```
#include "ConfigFile.h"

void InitWithFile (string filePath)
```

Description Initialize the object with the file specified by its path.

InitWithContent

Synopsis
```
#include "ConfigFile.h"

void InitWithContent (string content)
```

Description Initialize the object directly with the content of a file.

Exists

Synopsis
```
#include "ConfigFile.h"

void Exists ()
```

Description Call this method to know if this object has been initialized.

Returns True if this object has been initialized, false otherwise.

Get

Synopsis
```
#include "ConfigFile.h"

string Get (string key)
```

Description If the parsing of the file was successful, and if the *key* was specified, the method returns the value associated with the *key*.

Returns The value associated with the *key*.

Contains

Synopsis
```
#include "ConfigFile.h"

string Contains (string key)
```

Description If the parsing of the file was successful, tells if the *key* was specified in the file.

Returns True if the *key* was specified in the file, false otherwise.

GetContent

Synopsis
```
#include "ConfigFile.h"

string GetContent ()
```

Returns The content of the file.

GetMap

Synopsis
```
#include "ConfigFile.h"

map<string, string> GetMap ()
```

Description If the parsing of the file was successful, returns the map of pairs created during initialization. In case the parsing failed, returns an empty map.

Returns See description.

23.5.2 Publisher

23.5.2.1 PublisherModule

PublisherModule is an abstract class from which all the publisher classes must inherit. It contains a certain number of methods called by the publisher's Finite State Machine. All of them, apart ***Reset(), Version()*** and ***GetSummaryImage(),*** must be overwritten by the sub-classes. See Figure 23.6 to know when each method is called by the FSM.

Below is the description of the methods not shown in the figure.

GetSummaryImage

Synopsis
```
#include "PublisherModule.h"

string GetSummaryImage()
```

Description If implemented by the sub-classes, returns a so-called ***summary image***.

23.5.2.2 PublicationManager

The PublicationManager provides the interface to the publication methods. It also gives access to a certain number of utility methods, for example to know the run number or the agent's name. Each sub-class of PublisherModule (see above) has access to a global variable of type PublicationManager: ***gPublisher***. It is the only reference to the framework that the user's modules have.

Publish

Synopsis

```
#include "PublicationManager.h"

int Publish (TYPE object, const char* name, const char*
title,...)
```

Description This method exists in many flavours depending on the type of the object one wants to publish. It can be a MonitorObject or a TObject, the latter being encapsulated in a MonitorObjectTObject within the method. When publishing scalars or histograms, it is required to specify through templates the real type of the object (TH1F or TH1D for example). To be effective, a call to this method must occur within BookMonitorObjects or ***StartOfRun*** in the class PublisherModule. Publish() doesn't actually update the object in the data pool; it declares it as being part of the set which must be updated at every end of monitor cycle.

Returns 0 in case of success, 1 otherwise.

AgentName

Synopsis

```
#include "PublicationManager.h"

string AgentName ()
```

Description Returns the agent's name.

GetCurrentRun

Synopsis

```
#include "PublicationManager.h"

Run* GetCurrentRun ()
```

Description Returns the current run number.

Unpublish

Synopsis
```
#include "PublicationManager.h"

int Unpublish (MonitorObject*& mo)
```

Description Undo the Publish, ie. removes the object from the set of objects being updated at every end of cycle. To be effective, a call to this method must occur within the method PublisherModule::StartOfRun().

Returns 0 in case of success, 1 otherwise.

GetDbFileContent

Synopsis
```
#include "PublicationManager.h"

string GetDbFileContent (string detector, string filename)
```

Description Load the specified file from the table ***amoreref***.

Returns The content of the file if it exists, an empty string otherwise.

DownloadDbFile

Synopsis
```
#include "PublicationManager.h"

bool DownloadDbFile (string detector, string filename)
```

Description Load the specified file from the table ***amoreref*** and save its content in a file named *filename* in the current directory. If no file was found, an empty file is created.

Returns True if the file existed and was successfully downloaded, false otherwise.

Quit

Synopsis
```
#include "PublicationManager.h"

void Quit ()
```

Description Ask the framework to quit. It does so in a graceful way, executing first the end of run sequence.

GetMonitorObject

Synopsis
```
#include "PublicationManager.h"
```

```
MonitorObject* GetMonitorObject (string name)
```

Description Returns the ***MonitorObject*** published under the name *name*. If no object found, returns NULL.

23.5.3 Subscriber

23.5.3.1 SubscriptionManager

The ***SubscriptionManager*** provides the interface to the subscription methods. It also gives access to a certain number of utility methods. Each sub-class of ***VisualModule*** (see below) has access to a global variable of type ***SubscriptionManager***: ***gSubscriber***. It is the only reference to the framework that the user's visual modules have.

Subscribe

Synopsis

```
#include "SubscriptionManager.h"

int Subscribe (const char* name)
```

Description Subscribe to the object given by 'name' = "<agentName>/<objectName". In order to later use the ***At()***, described below, one must first subscribe to the object.

Returns A positive or null value in case of success.

A negative number in case of problem:

- -1: Max number of subscription reached.
- -2: Subscription error (dim error).
- -3: Memory allocation error.

Unsubscribe

Synopsis

```
#include "SubscriptionManager.h"

int Unsubscribe (const char* name)

int Unsubscribe ()
```

Description Unsubscribe from the object given by 'name' = "<agentName>/<objectName>". In the variant without parameters, it unsubscribes from all the objects.

Returns In the first variant, with parameters, the return codes are:

- 0: success.

- 1: already unsubscribed.
- 2: object never subscribed.
- -1: the proxy doesn't exist.

The variant without parameters returns the number of objects that were unsubscribed.

At

Synopsis

```
#include "SubscriptionManager.h"

template<typename MonitorObjectType> MonitorObjectType*
At(const char* key);
```

Description Returns the ***MonitorObject*** for the key = "<agentName>/<objectName>". The object must have been subscribed beforehand.

Last

Synopsis

```
#include "SubscriptionManager.h"

template<typename MonitorObjectType> MonitorObjectType*
Last(const char* agent, const char* object);
```

Description Returns the ***MonitorObject*** specified by its agent and object name, NULL if none was found. The returned object must be deleted by the caller.

Reset

Synopsis

```
#include "SubscriptionManager.h"

int Reset (string agentName)
```

Description Send a reset command to the agent named *agentName*

Returns

- 1:Ssuccess.
- 0: Reset could not be delivered (DIM issue).
- -1: Agent not found.

Stop

Synopsis

```
#include "SubscriptionManager.h"
```

```
int Stop (string agentname)
```

Description Send a command to the agent named *agentName* asking it to stop and exit.

Returns
- 1: Success.
- 0: Stop could not be delivered (DIM issue).
- -1: Agent not found.

Archive

Synopsis

```
#include "SubscriptionManager.h"

void Archive (const char* agentname, const char* moname, const
char* description)
```

Description Archive the object *moname* of agent *agentname* and put the description *description*.

ActiveAgents

Synopsis

```
#include "SubscriptionManager.h"

string ActiveAgents (const char* const det = 0)
```

Description Returns a string containing a list (colon separated) of the active agents in the system.

AllowedActiveAgents

Synopsis

```
#include "SubscriptionManager.h"

vector<string> AllowedActiveAgents (const char* name)
```

Description Returns a list of agents the subscriber is allowed to act on. Users can use this method to know whether they are allowed to start, stop or reset a given agent.

An agent is allowed to act (stop / start) on an agent if one of the following is true:

a. The user (login name) that started the agent is the same as the user who started the client

b. The detector code of the agent is the same as the detector code of the client

c. The user who started the client is equal to the detector code of the agent

AgentPublications

Synopsis

```
#include "SubscriptionManager.h"

string AgentPublications (const char* const agentname)
```

Description Return a list of all the objects published by the agent called *agentname*.

AgentPublicationsDetailsStop

Synopsis

```
#include "SubscriptionManager.h"

string AgentPublicationsDetails (const char* const agentname)
```

Description Return a list of all the objects published by the agent called *agentName* with details on their quality, their type and their visibility (expert / shifter).

Returns A string with the format (repeted for each agent):
agentName#type#quality#visibility:

GetDbFileContent

Synopsis

```
#include "SubscriptionManager.h"

string GetDbFileContent (string detector, string filename)
```

Description Load the specified file from the table amoreref.

Returns The content of the file if it exists, an empty string otherwise.

DownloadDbFile

Synopsis

```
#include "SubscriptionManager.h"

bool DownloadDbFile (string detector, string filename)
```

Description Load the specified file from the table amoreref and save its content in a file named *filename* in the current directory. If no file was found, an empty file is created.

Returns True if the file existed and was successfully downloaded, false otherwise.

StoreFile & StoreFileContent

Synopsis

```
#include "SubscriptionManager.h"

int StoreFile (string filename, string pathToFile, bool
overwrite)

int StoreFileContent (string filename, string content, bool
overwrite)
```

Description Store a file in the database table ***amoreref***. The first variant accepts a path name to a file, whereas the second directly takes the content of the file. The parameter *filename* is used to name the file in the database and can be different from the file on disk.

Returns

- 0: Successful insertion.
- 1: File overwritten.
- -1: File already exists, no overwrite.
- -2: Failure.

GetFilesList

Synopsis

```
#include "SubscriptionManager.h"

vector<string> GetFilesList (string pattern="")
```

Description Find the (configuration) files stored in the database for the user (login name) who started the client. If *pattern* is specified, only the files whose names contain the pattern will be returned.

Returns A vector of the files' names corresponding to the criteria

GetDetector

Synopsis

```
#include "SubscriptionManager.h"

string GetDetector (string agentname)
```

Description Returns the detector's code of the agent called *agentname*

GetRun

Synopsis

```
#include "SubscriptionManager.h"

int GetRun(const char* key)
```

Description Each ***MonitorObject*** is stored with the run number during which it was published. This method returns the run number for the object specified by *key* = "<agentName>/<objectName>".

23.5.4 User Interface (UI)

23.5.4.1 VisualModule

VisualModule is an abstract class from which all the UI classes must inherit. It extends ***SubscriberModule***. It contains a certain number of methods called by the subscriber's Finite State Machine. All of them, apart *Reset()*, must be overwritten by the sub-class even though most of them are usually left empty. See Figure 23.6 to know when each method is called by the FSM.

You can notice that the methods are almost the same between ***PublisherModule*** and ***VisualModule*** (inheriting from ***SubscriberModule***). This is made on purpose to have the same structure and the same FSM on both side of the framework. However, the drawback is that certain methods, especially ***StartOfRun*** and ***EndOfRun***, are not meaningful on the client side. Below a subset of methods is listed which are either especially important or need a bit of explanation.

---

fConfigFile

Synopsis
```
#include "VisualModule.h"
```

Description A pointer to the configuration file. It might be null if no configuration file was specified by the user at startup (using the flag **-g**).

Construct

Synopsis
```
#include "VisualModule.h"

void Construct()
```

Description Build the user interface within this method.

Update

Synopsis
```
#include "VisualModule.h"

void Update()
```

Description Get updates of the subscribed ***MonitorObjects*** in this method.

Process

Synopsis

```
#include "VisualModule.h"

void Process()
```

Description Process the ***MonitorObjects*** retrieved in *Update()* in this method.

InitAuto

Synopsis

```
#include "VisualModule.h"

void InitAuto()
```

Description The user interface calls the ***Construct*** at startup. However this behaviour can be changed by overwriting this method. Returns true if the module should initialize automatically, false otherwise. In the latter case, the user will have to press the button “Init”.

StartAuto

Synopsis

```
#include "VisualModule.h"

void StartAuto()
```

Description The user interface waits after it has been initialized. This behaviour can be changed in order to start updating the objects directly. Returns true if the module should initialize and start automatically, false otherwise. In the latter case, the user will have to press the button “Start”.

UpdatePeriod

Synopsis

```
#include "VisualModule.h"

int UpdatePeriod()
```

Description The user interface updates the objects at regular intervals. By default, the duration of the intervals is set to 30 seconds. This value can be changed within the module, by overwriting the method ***UpdatePeriod()*** in the subclass of ***VisualModule***. The end user can, of course, still change the duration. The method will be called right after *Construct()* has been called.

23.5.5 Detector Algorithms (DA) library

AmoreDA

Synopsis

```
#include "AmoreDA.h"

AmoreDA(EMode mode)
```

Description Constructor of the class ***AmoreDA***. *mode* should be ***kSender*** on the DA side. In an hypothetical client, it would be ***kReceiver***.

Send

Synopsis

```
#include "AmoreDA.h"

int Send(const char* objectName, const TObject* obj, const
bool asMonitorObject=false)
```

Description This function has to be used in the DA to send the object *obj* under the name *objectName* to the AMORE pool. It will be stored in a table named after the content of the environment variable ***$AMORE_DA_NAME***. If this variable is not defined, ***$DATE_ROLE_NAME*** is used instead. The last parameter, *asMonitorObject*, stipulates whether the ***TObject*** must be encapsulated within a ***MonitorObject*** or not.

Returns 0 on success, 1 otherwise.

23.5.6 Archiver

23.5.6.1 ArchiverModule

Archive

Synopsis

```
#include "ArchiverModule.h"

void Archive(string agentName, string moName, string
updatetime, string desc)
```

Description This method must implement the archiving of the object specified by its agent, its name and its updatetime.

Clean

Synopsis

```
#include "ArchiverModule.h"

void Clean()
```

Description This method must clean up too old, non-permanent archives.

Init

Synopsis

```
#include "ArchiverModule.h"

void Init(PoolInterface* pi, ConfigFile cf)
```

Description This method gives the opportunity to the module to set up its internal state, given a connection to the database (***PoolInterface***) and to the configuration file.

23.6 Tools

The AMORE package contains various tools in the form of binaries or scripts. Their purpose is to set up a machine, to configure or discover the environment, to launch agents and clients and finally to manage the infrastructure (agents or configuration files).

Here is a list of these tools sorted by category. For details on their usage please refer to the AMORE section in the ALICE DAQ WIKI.

Setup	newAmoreSite	Create and set up a new AMORE_SITE
	amoreMysqlSetup	Configures MySQL for AMORE (create the db, users and tables)
Utilities	amore-config	Get all configuration parameters for AMORE
	amoreSetup	Set up the environment according to AMORE.params
	amoreUpdateDB.tcl	Update the database scheme (after an update of AMORE)
Launchers	amore	Start a client
	amoreAgent	Start an agent
	amoreArchiver	Start the archiver
Management	newAmoreAgent	Create a new agent
	amoreConfigFilesBrowser	An interface to manage the configuration files
	amoreAgentsManager.tcl	An UI to start/stop the agents and the archiver
	amoreEditDb	An expert UI to browse and edit the database

Part VI

The ALICE electronic logbook

November 2010

ALICE DAQ Project

eLogBook

24 The ALICE Electronic Logbook

ALICE needs a bookkeeping facility to record not only the activities at the experimental area but also all the non-physics metadata associated with each performed run. As shifters come and go, a central information repository is needed to store reports of incidents, configuration changes, achievements or planned operations. Furthermore, a historical record of data-taking conditions and statistics is needed not only to allow the selection of good run candidates for prioritized offline processing, but also to detect trends and correlations, create aggregated reports and assist the run coordination in fulfilling the scientific goals.The ALICE Electronic Logbook (eLogbook) fulfills this requirement, providing a repository for both shifters / experts reports and run statistics / conditions. It also provides a modern Web-based Graphical Human Interface, allowing the members of the ALICE collaboration to access and filter this vast data record easily.

24.1 Architecture

24.1.1 Overview

The ALICE Electronic Logbook (eLogbook) is the Data Acquisition bookkeeping facility in ALICE. It is based on a LAMP (Linux, Apache, MySQL and PHP) software stack, with the relational database (see Section 24.2) serving as a data repository and the Web-based Graphical User Interface (see Section 24.6) providing interactive access to members of the ALICE collaboration.

An Application Programming Interface implemented in C (see Section 24.3) and several command-line tools (see Section 24.5) provide read / write access to the repository.

A daemon process (see Section 24.3) collects data published by the DCS and inserts it in the DB. Some of this data is gathered by a dedicated process that extracts information published by the LHC via the DIP protocol and republishes it in DCS (see Chapter 25).

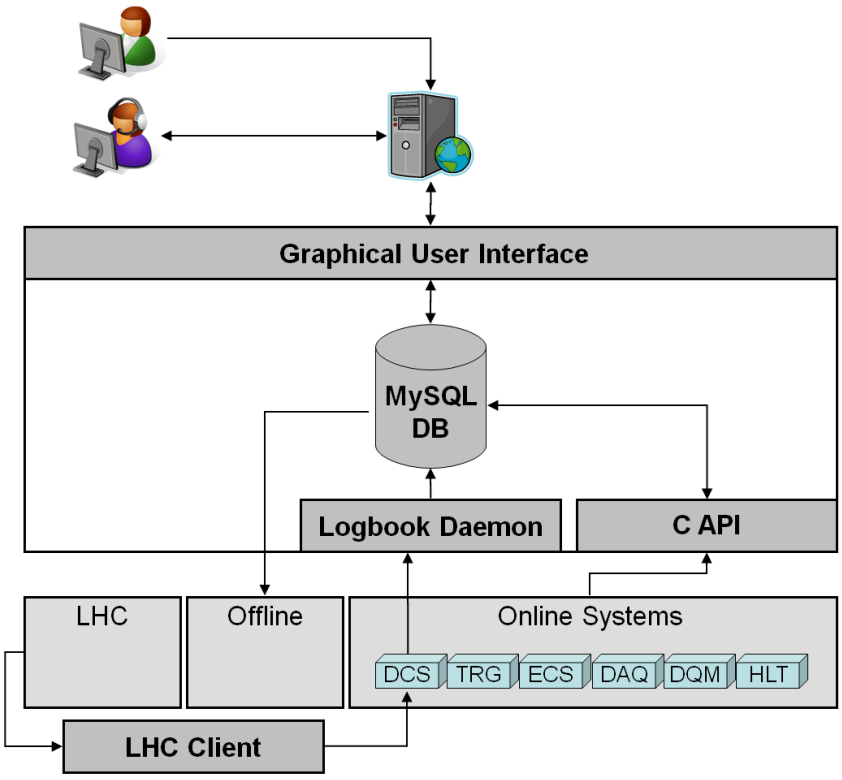

Figure 24.1 The architecture of the eLogbook and it's interfaces with the other ALICE systems and the LHC.

24.2 Database

24.2.1 Overview

The DB, running on a MySQL Server, is used to store heterogeneous data related with the experiment's activities. InnoDB is used as a storage engine for its support of both transactions and foreign keys constraints. As shown in Figure 22.2, the tables that compose this DB can be grouped into 4 different categories:

- RUN CENTRIC: related to a specific run.
- LOG ENTRY CENTRIC: related to a specific human or automatic text report with optional file attachment.
- USER CENTRIC: related to the GUI users.
- OTHER: tables that do not belong to the previous categories.

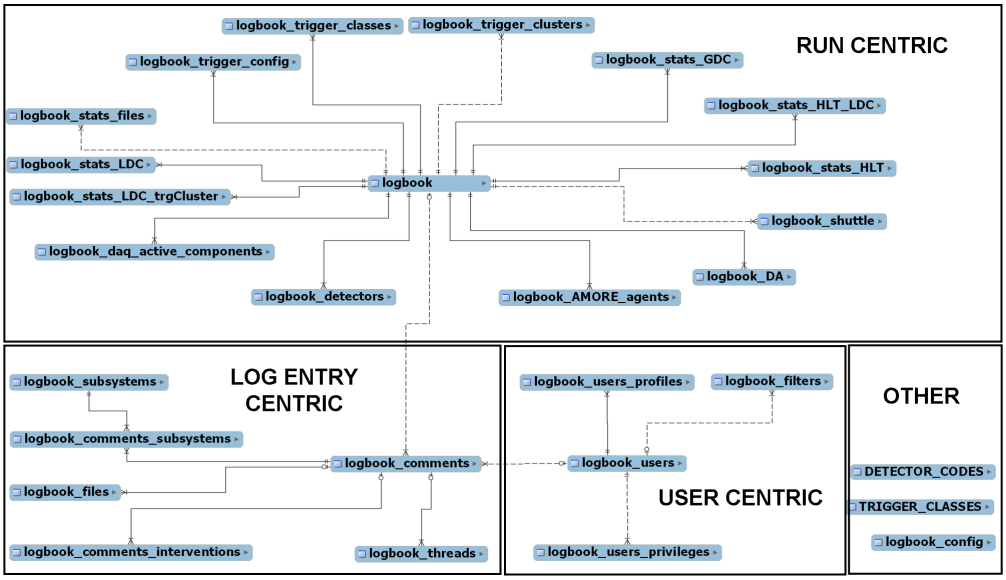

Figure 24.2 eLogbook's database schema

A stored procedure, called ***update_logbook_counters***, is executed every 60 seconds (and at the end of each run) to update the different global counters in the logbook table, whose value depends on partial counters spread throughout several tables.

Daily backups are performed to a RAID 6 disk array and the CERN Advanced STORage manager (CASTOR).

24.2.2 Table description

Below is a description of the eLogbook's tables.

24.2.2.1 *logbook* table

This table stores ***per run*** information. It is populated by:

- ***PCA/DCA***: run, time_created, time_completed, partition, detector, run_type, calibration, numberOfDetectors, detectorMask, ecs_success, daq_success, eor_reason, ecs_iteration_current and ecs_iteration_total fields.
- ***PCA Human Interface***: runQuality field.
- ***runControl***: HLTmode, DAQ_time_start, DAQ_time_end, runDuration, detectorMask, numberOfLDCs, numberOfGDCs, LDClocalRecording, GDClocalRecording, GDCmStreamRecording, eventBuilding and dataMigrated fields.
- ***logbookDaemon***: beamEnergy, beamType, LHCFillNumber, LHCTotalInteractingBunches, LHCTotalNonInteractingBunchesBeam1, LHCTotalNonInteractingBunchesBeam2, L3_magnetCurrent and Dipole_magnetCurrent fields.
- CTP software: L2a and ctpDuration fields.
- ***TDSM***: dataMigrated field.
- ***update_logbook_counters*** stored procedure:runDuration, totalSubEvents, totalDataReadout, totalEvents, totalDataEventBuilder, totalDataRecorded, averageDataRateReadout, averageDataRateEventBuilder, averageDataRateRecorded, averageSubEventsPerSecond and averageEventsPerSecond fields.

Table 24.1 ***logbook*** table (per run conditions and statistics)

Field	Description
run	Run number
time_created	Run number creation timestamp
DAQ_time_start	Start of data acquisition timestamp
DAQ_time_end	End of data acquisition timestamp
time_completed	End of run timestamp
time_update	Database row update date/time
runDuration	Duration of data acquisition
partition	ECS partition name (NULL if stand-alone run)
detector	Detector name (NULL if global run)
run_type	ECS run type
calibration	Flag indicating if run is a calibration run
ecs_success	Flag indicating if run finished successfully
daq_success	Flag indicating if run data acquisition finished successfully

Table 24.1 ***logbook*** table (per run conditions and statistics)

Field	Description
eor_reason	End of run reason as declared by the ECS
beamEnergy	Single beam energy in GeV
beamType	Type of collisions (‘p-p’, ‘Pb-Pb’, ‘p-Pb’, NULL if no collisions)
LHCFillNumber	LHC fill number
LHCTotalInteractingBunches	Total number of interacting bunches
LHCTotalNonInteractingBunchesBeam1	Total number of non-interacting bunches in beam 1
LHCTotalNonInteractingBunchesBeam2	Total number of non-interacting bunches in beam 2
numberOfDetectors	Number of detectors participating in the run
detectorMask	Bitmask of detectors participating in run (LSB = detector ID zero)
log	NOT USED
totalSubEvents	Total number of subevents collected by ***readout***
totalDataReadout	Total size of data collected by ***readout*** in bytes
averageSubEventsPerSecond	Average number of subevents per second
averageDataRateReadout	Average data rate collected by readout in bytes / second
totalEvents	Total number of events generated by ***eventBuilder***
totalDataEventBuilder	Total size of data generated by ***eventBuilder*** in bytes
averageEventsPerSecond	Average number of events per second
averageDataRateEventBuilder	Average data rate generated by ***eventBuilder*** in bytes / second
totalDataRecorded	Total size of data recorded by ***mStreamRecorder*** in bytes
averageDataRateRecorded	Average data rate recorded by ***mStreamRecorder*** in bytes / second
numberOfLDCs	Total number of LDCs participating in the run
numberOfGDCs	Total number of GDCs participating in the run

Table 24.1 ***logbook*** table (per run conditions and statistics)

Field	Description
numberOfStreams	NOT USED
LDClocalRecording	Flag indicating if local recording in the LDCs was enabled
GDClocalRecording	Flag indicating if local recording in the GDCs was enabled
GDCmStreamRecording	Flag indicating if mStreamRecording mode was enabled
eventBuilding	Flag indicating if Event Building was enabled
LHCperiod	LHC period ID (e.g. `LHC09c`)
HLT mode	High Level Trigger mode
dataMigrated	Status of the data migration to Tier 0
runQuality	Global run quality flag for the run as specified by the ECS shifter
L3_magnetCurrent	Current of the L3 magnet in Amperes (signed)
Dipole_magnetCurrent	Current of the Dipole magnet in Amperes (signed)
L2a	Total number of L2a trigger decisions generated
ctpDuration	Duration during which at least 1 trigger class time sharing group was active since SOR in seconds
ecs_iteration_current	Current ECS iteration number for detector calibration with several runs
ecs_iteration_total	Total ECS iterations expected for detector calibration with several runs

24.2.2.2 ***logbook_detectors*** table

This table stores ***per detector per run*** information. It is populated by:

- ***runControl***: run_number, detector and run_type fields.
- GUI: run_quality field.
- CTP software: L2a field.

Table 24.2 ***logbook_detectors*** table

Field	Description
run_number	Run number
detector	Detector name

Table 24.2 ***logbook_detectors*** table

Field	Description
run_type	ECS run type
run_quality	Run quality for the detector / run pair as indicated by the ECS shifter (‘`No report`’, ‘`Good run`’, ‘`Bad run`’)
L2a	Number of L2a trigger decisions generated for this detector

24.2.2.3 ***logbook_stats_LDC*** table

This table stores ***per LDC per run*** information. The HLT counters are populated by ***hltAgent*** and the other fields by ***readout***.

Table 24.3 ***logbook_stats_LDC*** table

Field	Description
run	Run number
LDC	LDC rolename
detectorId	Detector ID as specified by the ***id*** field of the ***DETECTOR_CODES*** table
eventCount	Number of subevents collected by ***readout***
eventCountPhysics	Number of PHYSICS subevents collected by ***readout***
eventCountCalibration	Number of CALIBRATION subevents collected by ***readout***
bytesInjected	Size of data collected by ***readout*** in bytes
bytesInjectedPhysics	Size of PHYSICS data collected by ***readout*** in bytes
bytesInjectedCalibration	Size of CALIBRATION data collected by ***readout*** in bytes
hltAccepts	Number of HLT accept decisions for this LDC
hltRejects	Number of HLT reject decisions for this LDC
hltBytesRejected	Size of data rejected by HLT decisions for this LDC in bytes
time_update	Database row update date / time

24.2.2.4 ***logbook_stats_LDC_trgCluster*** table

This table stores ***per trigger cluster per LDC per run*** information. It is populated by ***readout***.

Table 24.4 ***logbook_stats_LDC_trgCluster*** table

Field	Description
run	Run number
LDC	LDC rolename
cluster	Trigger cluster ID
eventCount	Number of subevents collected by **UHDGRXW**
bytesInjected	Size of data collected by ***readout*** in bytes
time_update	Database row update date / time

24.2.2.5 *logbook_stats_GDC* table

This table stores ***per GDC per run*** information. It is populated by ***eventBuilder***.

Table 24.5 ***logbook_stats_GDC*** table

Field	Description
run	Run number
GDC	GDC rolename
eventCount	Number of events generated by ***eventBuilder***
eventCountPhysics	Number of PHYSICS events generated by ***eventBuilder***
eventCountCalibration	Number of CALIBRATION events generated by ***eventBuilder***
bytesRecorded	Size of data generated by ***eventBuilder*** in bytes
bytesRecordedPhysics	Size of PHYSICS data generated by ***eventBuilder*** in bytes
bytesRecordedCalibration	Size of CALIBRATION data generated by ***eventBuilder*** in bytes
time_update	Database row update date / time

24.2.2.6 *logbook_stats_files* table

This table stores ***per data file per run*** information related with the full data chain, from the file creation up to the migration to CASTOR. It is mostly populated by ***mStreamRecorder***, with the status field (and corresponding timestamps) also updated by ***TDSM***.

Table 24.6 ***logbook_stats_files*** table

Field	Description
id	File ID
run	Run number

Table 24.6 ***logbook_stats_files*** table

Field	Description
fileName	Filename (without path)
location	Path to current file directory
local	Flag indicating if file is local
rolename	Rolename of DAQ node writing the file
hostname	Hostname of DAQ node writing the file
pid	Process ID writing the file
time_update	Database row update date / time
time_write_begin	File writing start date / time
time_write_end	File writing end date / time
time_migrate_request	File migration request date / time
time_migrate_begin	File migration start date / time
time_migrate_end	File migration end date / time
status	File status (‘`Writing`’, ‘`Closed`’, ‘`Waiting migration request`’, ‘`Migration requested`’, ‘`Migrating`’, ‘`Migrated`’)
eventCount	Number of events written to file
size	File size in bytes

24.2.2.7 ***logbook_daq_active_components*** table

This table stores ***per run*** information related with the active DAQ components (DDLs, LDCs, GDCs). It is populated by ***runControl***.

Table 24.7 ***logbook_daq_active_components*** table

Field	Description
run	Run number
LDC	Active LDCs IDs (LSB = ID zero)
GDC	Active GDCs IDs (LSB = ID zero)
DDL	Active DDLs IDs (LSB = ID zero)

24.2.2.8 ***logbook_shuttle*** table

This table stores ***per run*** information related with the Offline Shuttle processing. It is populated by ***ECS***, with the different processing status updated directly by the Shuttle software.

Table 24.8 ***logbook_shuttle*** table

Field	Description
run	Run number
shuttle_done	Flag indicating if Shuttle processing is complete
test_mode	Flag indicating if Shuttle should run in test mode
update_time	Database row update date / time
SPD	Shuttle processing status for detector SPD (‘UNPROCESSED’, ‘INACTIVE’, ‘FAILED’, ‘DONE’)
SDD	Shuttle processing status for detector SDD (‘UNPROCESSED’, ‘INACTIVE’, ‘FAILED’, ‘DONE’)
SSD	Shuttle processing status for detector SSD (‘UNPROCESSED’, ‘INACTIVE’, ‘FAILED’, ‘DONE’)
TPC	Shuttle processing status for detector TPC (‘UNPROCESSED’, ‘INACTIVE’, ‘FAILED’, ‘DONE’)
TRD	Shuttle processing status for detector TRD (‘UNPROCESSED’, ‘INACTIVE’, ‘FAILED’, ‘DONE’)
TOF	Shuttle processing status for detector TOF (‘UNPROCESSED’, ‘INACTIVE’, ‘FAILED’, ‘DONE’)
PHS	Shuttle processing status for detector PHOS (‘UNPROCESSED’, ‘INACTIVE’, ‘FAILED’, ‘DONE’)
CPV	Shuttle processing status for detector CPV (‘UNPROCESSED’, ‘INACTIVE’, ‘FAILED’, ‘DONE’)
HMP	Shuttle processing status for detector HMPID (‘UNPROCESSED’, ‘INACTIVE’, ‘FAILED’, ‘DONE’)
MCH	Shuttle processing status for detector MUON_TRK (‘UNPROCESSED’, ‘INACTIVE’, ‘FAILED’, ‘DONE’)
MTR	Shuttle processing status for detector MUON_TRG (‘UNPROCESSED’, ‘INACTIVE’, ‘FAILED’, ‘DONE’)
PMD	Shuttle processing status for detector PMD (‘UNPROCESSED’, ‘INACTIVE’, ‘FAILED’, ‘DONE’)
FMD	Shuttle processing status for detector FMD (‘UNPROCESSED’, ‘INACTIVE’, ‘FAILED’, ‘DONE’)
T00	Shuttle processing status for detector T0 (‘UNPROCESSED’, ‘INACTIVE’, ‘FAILED’, ‘DONE’)
V00	Shuttle processing status for detector V0 (‘UNPROCESSED’, ‘INACTIVE’, ‘FAILED’, ‘DONE’)
ZDC	Shuttle processing status for detector ZDC (‘UNPROCESSED’, ‘INACTIVE’, ‘FAILED’, ‘DONE’)
ACO	Shuttle processing status for detector ACORDE (‘UNPROCESSED’, ‘INACTIVE’, ‘FAILED’, ‘DONE’)

Table 24.8 ***logbook_shuttle*** table

Field	Description
EMC	Shuttle processing status for detector EMCal (‘UNPROCESSED’, ‘INACTIVE’, ‘FAILED’, ‘DONE’)
TST	Shuttle processing status for detector DAQ_TEST (‘UNPROCESSED’, ‘INACTIVE’, ‘FAILED’, ‘DONE’)
HLT	Shuttle processing status for HLT (‘UNPROCESSED’, ‘INACTIVE’, ‘FAILED’, ‘DONE’)
GRP	Shuttle processing status of global run parameters (‘UNPROCESSED’, ‘INACTIVE’, ‘FAILED’, ‘DONE’)
TRI	Shuttle processing status for of the Trigger parameters (‘UNPROCESSED’, ‘INACTIVE’, ‘FAILED’, ‘DONE’)

24.2.2.9 ***logbook_DA*** table

This table stores ***per run per Detector Algorithm*** information.

Table 24.9 ***logbook_DA*** table

Field	Description
run	Run number
detector	Detector name related to the DA
DAname	DA name
DAversion	DA version number (from RPM)
DAstdout	Output of the DA
role	Rolename of DAQ node associated with the DA
exitCode	Exit code of the DA
commandLine	Executed command
workingDir	Working directory

24.2.2.10 ***logbook_AMORE_agents*** table

This table stores ***per run per AMORE agent*** information. It is populated by the ***AMORE*** framework.

Table 24.10 ***logbook_AMORE_agents*** table

Field	Description
run	Run number
detector	Detector name related to the ***AMORE*** agent
agentName	***AMORE*** agent name

Table 24.10 ***logbook_AMORE_agents*** table

Field	Description
agentVersion	***AMORE*** agent version number (from RPM)
runtimeParameters	***AMORE*** agent runtime parameters
MOpublished	Number of published Monitoring Objects
MOversionsPublished	Number of published Monitoring Objects versions
bytesPublished	Total size of published Monitoring Objects in bytes
averageCPUtime	Average CPU time of a monitor cycle
averageRealTime	Average real time of a monitor cycle
QAsummaryImage	Quality Assurance summary image
QAsummaryImageSize	Size of Quality Assurance summary image in bytes
time_update	Database row update date / time

24.2.2.11 *logbook_trigger_clusters* table

This table stores ***per run per trigger cluster*** information. It is populated by the CTP software.

Table 24.11 ***logbook_trigger_clusters*** table

Field	Description
run	Run number
partition	ECS partition name
cluster	Trigger cluster ID (1-6)
detectorMask	24-bit detector IDs bitmask (LSB = detector ID zero) corresponding to the readout detectors of this cluster
inputDetectorMask	24-bit detector IDs bitmask (LSB = detector ID zero) corresponding to the trigger detectors of this cluster
triggerClassMask	50-bit trigger classes ID bitmask (LSB = trigger class ID zero) of the trigger classes triggering this cluster

24.2.2.12 *logbook_trigger_classes* table

This table stores ***per run per trigger class*** information. It is populated by:

- CTP software: run, classId, className, classGroupId, classGroupTime, L0b, L0a, L1b, L1a, L2b, L2a and ctpDuration fields.
- ***hltAgent***: hltAccepts, hltPartialAccepts, hltOnly, hltRejects and hltBytesRejected fields.

Table 24.12 ***logbook_trigger_classes*** table

Field	Description
run	Run number
classId	Trigger class ID (0-49)
className	Trigger class name
classGroupId	Trigger class time sharing group ID
classGroupTime	Trigger class time sharing group duration in seconds
L0b	Number of L0b trigger decisions generated for this trigger class
L0a	Number of L0a trigger decisions generated for this trigger class
L1b	Number of L1b trigger decisions generated for this trigger class
L1a	Number of L1a trigger decisions generated for this trigger class
L2b	Number of L2b trigger decisions generated for this trigger class
L2a	Number of L2a trigger decisions generated for this trigger class
ctpDuration	Duration during which this trigger class was active since SOR in seconds
hltAccepts	Number of HLT accept decisions for this trigger class
hltPartialAccepts	Number of HLT partial accept decisions for this trigger class
hltOnly	Number of HLT only decisions for this trigger class
hltRejects	Number of HLT reject decisions for this trigger class
hltBytesRejected	Size of data rejected by HLT decisions for this trigger class in bytes

24.2.2.13 ***logbook_trigger_inputs*** table

This table stores ***per run*** per trigger input information. It is populated by the CTP software.

Table 24.13 ***logbook_trigger_inputs*** table

Field	Description
run	Run number
inputLevel	Trigger input level (0-2)
inputId	Trigger input Id (1-24 for level 0, 1-24 for level 1, 1-12 for level 2)

Table 24.13 *logbook_trigger_inputs* table

Field	Description
inputName	Trigger input name
inputCount	Number of trigger signals for this trigger input

24.2.2.14 *logbook_trigger_config* table

This table stores ***per run*** information related with the CTP configuration. It is populated by the CTP software.

Table 24.14 *logbook_trigger_config* table

Field	Description
run	Run number
configFile	Trigger configuration file
alignmentFile	Trigger alignment file

24.2.2.15 *logbook_stats_HLT* table

This table stores ***per run*** information related with the HLT decisions. It is populated by the ***update_logbook_counters*** stored procedure.

Table 24.15 *logbook_stats_HLT* table

Field	Description
run	Run number
hltAccepts	Total number of HLT accept decisions for this run
hltPartialAccepts	Total number of HLT partial accept decisions for this run
hltOnly	Total number of HLT only decisions for this run
hltRejects	Total number of HLT reject decisions for this run
hltBytesRejected	Size of data rejected by HLT decisions for this run in bytes
time_update	Database row update date / time

24.2.2.16 *logbook_stats_HLT_LDC* table

This table stores ***per run per HLT LDC*** information. It is populated by ***hltAgent***.

Table 24.16 *logbook_stats_HLT_LDC* table

Field	Description
run	Run number

Table 24.16 ***logbook_stats_HLT_LDC*** table

Field	Description
LDC	HLT LDC rolename
hltAccepts	Number of HLT accept decisions taken by this HLT LDC
hltPartialAccepts	Number of HLT partial accept decisions taken by this HLT LDC
hltOnly	Number of HLT only decisions taken by this HLT LDC
hltRejects	Number of HLT reject decisions taken by this HLT LDC
time_update	Database row update date/time

24.2.2.17 ***logbook_comments*** table

This table stores the Log Entries. It is populated by the GUI (human generated Log Entries such as End-Of-Shift reports), by ***runControl*** (automatic "start/end of run" Log Entries) and by the ***PCA Human Interface*** (ECS operator "end of run" Log Entries).

Table 24.17 ***logbook_comments*** table

Field	Description
id	Log Entry ID
run	Run number associated with the Log Entry
userid	User ID of the Log Entry author as specified in the ***id*** field of the ***logbook_users*** table
title	Log Entry title
comment	Log Entry body
class	Log Entry class ('HUMAN', 'PROCESS')
type	Log Entry type ('GENERAL', 'HARDWARE', 'CAVERN', 'SOFTWARE', 'NETWORK', 'EOS', 'OTHER')
time_created	Log Entry creation date/time
deleted	Flag indicating if the Log Entry is deleted
parent	Parent Log Entry (for threads)
root_parent	Root parent Log Entry (for threads)
dashboard	Flag indicating if the Log Entry is an announcement
time_validity	Log Entry validity date/time

24.2.2.18 *logbook_comments_interventions* table

This table expands the ***logbook_comments*** table, adding additional information related to ***on call*** interventions. It is populated by the GUI.

Table 24.18 ***logbook_comments_interventions*** table

Field	Description
commentid	Log Entry ID
type	Intervention type ('REMOTE', 'ONSITE')

24.2.2.19 *logbook_files* table

This table stores the files attached to the Log Entries. It is populated by the GUI.

Table 24.19 ***logbook_files*** table

Field	Description
commentid	ID of the Log Entry to which the file is attached to
fileid	File ID
filename	Filename
size	File size in bytes
title	File title
data	File binary data
thumbnail_small	File small-sized thumbnail (100x100, only filled if file is an image)
thumbnail_medium	File medium-sized thumbnail (320x320, only filled if file is an image)
content_type	File ***Content-Type***
time_created	File creation date/time
deleted	Flag indicating if the file is deleted

24.2.2.20 *logbook_threads* table

This table expands the ***logbook_comments*** table, adding additional information related to threads. It is populated by the GUI.

Table 24.20 ***logbook_threads*** table

Field	Description
id	Log Entry ID (root parent Log Entry)
title	Thread title
ticket_status	Ticket status ('OPEN', 'CLOSED')

24.2.2.21 *logbook_subsystems* table

This table stores the definition of the Log Entries Subsystems.

Table 24.21 *logbook_subsystems* table

Field	Description
id	Subsystem ID
name	Subsystem name
text	Subsystem text (to be displayed in GUIs)
parent	Parent subsystem
email	Automatic email notification email address (single or multiple in CSV format)
notify_no_run_log_entries	Flag indicating if an automatic email notification should be sent for the Log Entries of this subsystem without run number
notify_run_log_entries	Flag indicating if an automatic email notification should be sent for the Log Entries of this subsystem with a run number
notify_quality_flags	Flag indicating if an automatic email notification should be sent for the Log Entries of this subsystem related with the change of the quality flags

24.2.2.22 *logbook_comments_subsystems* table

This table stores the relationship between the Log Entries and the Subsystems. It is populated by the GUI.

Table 24.22 *logbook_comments_subsystems* table

Field	Description
commentid	Log Entry ID
subsystemid	Subsystem ID

24.2.2.23 *logbook_users* table

This table stores the main information of the GUI's users. It is populated by the GUI at the moment of the users's first login.

Table 24.23 *logbook_users* table

Field	Description
id	User ID
first_name	User's first name
full_name	User's full name

Table 24.23 *logbook_users* table

Field	Description
email	User's email address
group_name	User's CERN group name

24.2.2.24 *logbook_users_privileges* table

This table stores the GUI's users privileges. It is populated by the GUI.

Table 24.24 *logbook_users_privileges* table

Field	Description
id	User privilege ID
userid	User ID
time_start	Privilege start date/time
time_end	Privilege end date/time
privilege	Privilege set (one or more of 'NONE', 'READ', 'WRITE', 'ADMIN', 'SUPER')
revoked	Flag indicating if the privilege is revoked

24.2.2.25 *logbook_users_profiles* table

This table stores additional information of the GUI's users. It is populated by the GUI.

Table 24.25 *logbook_users_profiles* table

Field	Description
userid	User ID
name	Profile entry name
value	Profile entry value

24.2.2.26 *logbook_filters* table

This table stores the definition of the search filters predefined values (see Section 24.6.3.6).

Table 24.26 *logbook_filters* table

Field	Description
id	User filter ID
userid	User ID
content_name	Name of the GUI's content to which the filter applies

Table 24.26 ***logbook_filters*** table

Field	Description
column_qs_var	Name of the GUI's URL query string variable to which the filter's value will be assigned
name	Filter's name
value	Filter's value or SQL code
print_order	Filter's order of appearance in the GUI
sql_flag	Flag indicating if this filter is SQL based
load_by_default	Flag indicating if this filter should be loaded by default
enabled	Flag indicating if this filter is enabled

24.2.2.27 ***DETECTOR_CODES*** table

This table stores the definition of the different ALICE detectors.

Table 24.27 ***DETECTOR_CODES*** table

Field	Description
id	Detector ID
name	Detector name
code	Detector 3-letter code
isVirtual	Flag indicating if the detector is virtual
description	Detector description

24.2.2.28 ***TRIGGER_CLASSES*** table

This table stores the definition of the different ALICE trigger classes.

Table 24.28 ***TRIGGER_CLASSES*** table

Field	Description
className	Trigger class name
description	Trigger class description
obsolete	Flag indicating if the trigger class is obsolete

24.2.2.29 ***logbook_config*** table

This table stores internal eLogbook information (e.g. version number).

Table 24.29 ***logbook_config*** table

Field	Description
Name	Configuration parameter name
Value	Configuration parameter value
Description	Configuration parameter description

24.2.3 Stored Procedures

Below is a list of the eLogbook's stored procedures.

update_logbook_counters

Synopsis `CALL update_logbook_counters(run_number INT)`

Description Updates the ***runDuration***, ***totalSubEvents***, ***totalDataReadout***, ***totalEvents***, ***totalDataEventBuilder***, ***totalDataRecorded***, ***averageDataRateReadout***, ***averageDataRateEventBuilder***, ***averageDataRateRecorded***, ***averageSubEventsPerSecond*** and ***averageEventsPerSecond*** fields of the ***logbook*** table.

It also updates the statistics of the ***logbook_stats_HLT*** table.

Returns No value is returned.

DAQlogbookSP_updateListTriggerClasses

Synopsis `CALL DAQlogbookSP_updateListTriggerClasses()`

Description Updates the list of trigger classes stored in the ***TRIGGER_CLASSES*** table, based on the distinct values of the ***className*** field of the ***logbook_trigger_classes*** table.

Returns No value is returned.

24.2.4 Events

Below is a list of the eLogbook's events.

DAQlogbookEV_updateListTriggerClasses

Description Executes the ***DAQlogbookSP_updateListTriggerClasses*** stored procedure every day at 01:00 h.

24.3 Application Programming Interface

24.3.1 Overview

Read / write access is available via the ***DAQlogbook*** C API. A version for Tcl is also available as a shared library.

24.3.2 Environment variables

The following environment variables are available to configure the behavior of the ***DAQlogbook*** API:

- ***DAQ_DB_LOGBOOK:*** sets the credentials to access the DB. The format is ***"USERNAME:PASSWORD@HOSTNAME/DBNAME"***.
- ***WITH_INFOLOGGER:*** sets the logging mode. If set, logging uses the ***infoLogger*** system. If not set, log messages are sent to ***stdout***.
- ***DAQ_LOGBOOK_VERBOSE:*** sets the logging level. Possible values are:
  - 0: no messages
  - 1: error messages
  - 2: same as 1 + debug messages
  - > 2: same as 2 + all SQL queries

  If not set, the default value is 1.

24.3.3 Database connection functions

Below is a list of functions providing basic connection functionality to the eLogbook's database.

DAQlogbook_open

Synopsis

```
#include "DAQlogbook.h"

int DAQlogbook_open(const char *cx_params)
```

Description Open a MySQL connection. Credentials should be given via the *cx_params* parameter in the ***"USERNAME:PASSWORD@HOSTNAME/DBNAME"*** format. If an empty string is passed the credentials are taken from the ***DAQ_DB_LOGBOOK*** environment variable. If both are empty, eLogbook access via the API is disabled and further read / write function calls are ignored.

Returns Upon successful completion, this function will return a value of zero. Otherwise, the following value will be returned:

-1: error while connecting to the database.

1: *cx_params* and ***DAQ_DB_LOGBOOK*** are empty, eLogbook access disabled.

DAQlogbook_close

Synopsis

```
#include "DAQlogbook.h"

int DAQlogbook_close(void)
```

Description Close a MySQL connection and release previously used resources.

Returns This function always returns a value of zero.

24.3.4 Logging functions

Below is a list of functions allowing logging configuration.

DAQlogbook_verbose

Synopsis

```
#include "DAQlogbook.h"

void DAQlogbook_verbose(int v)
```

Description Set the logging level of the API. Possible values are the same as described for the ***DAQ_LOGBOOK_VERBOSE*** environment variable.

Returns No value is returned.

24.3.5 eLogbook READ access functions

Below is a list of functions providing READ access to the eLogbook database.

DAQlogbook_datafile_getIdFromName

Synopsis

```
#include "DAQlogbook.h"

int DAQlogbook_datafile_getIdFromName(const char *name,
unsigned int run, int *id)
```

Description Retrieve the `ID` of the entry in the ***logbook_stats_files*** table corresponding to the given filename and run number. The value is stored in the *id* parameter.

Returns Upon successful completion, this function returns a value of zero. Otherwise, an error code with a value equal to the line number where the error occurred will be returned.

DAQlogbook_get_AMORE_agent_summary_img

Synopsis

```
#include "DAQlogbook.h"

int DAQlogbook_get_AMORE_agent_summary_img(const char
*agentname, void **summary_img, unsigned long *n_bytes)
```

Description Retrieve the Quality Assurance summary image of a specific AMORE agent for the latest run where the agent was active. The image itself is stored in the *summary_img* parameter and it's size in bytes in the *n_bytes* parameter.

Returns Upon successful completion, this function will return a value of zero. Otherwise, an error code with a value equal to the line number where the error occurred will be returned.

DAQlogbook_query_getTriggerClusters

Synopsis

```
#include "DAQlogbook.h"

int DAQlogbook_query_getTriggerClusters(unsigned int run,
char *partition, unsigned int *clustermask)
```

Description Build a 6-bit bitmask (LSB = cluster #1) of the active trigger clusters for the given run. The value is then stored in the *clustermask* parameter.

Returns Upon successful completion, this function returns a value of zero. Otherwise, -1 will be returned.

DAQlogbook_query_getDetectorIdsFromTriggerClassMask

Synopsis

```
#include "DAQlogbook.h"

int DAQlogbook_query_getDetectorIdsFromTriggerClassMask
(unsigned int run, unsigned long long triggerClassMask,
unsigned int *detectorMask, unsigned int *clustermask)
```

Description Build a 24-bit detector IDs bitmask (LSB = detector ID zero, stored in the *detectorMask* parameter) corresponding to all the readout detectors of all the trigger clusters triggered by a given 50-bit trigger classes bitmask (*triggerClassMask* parameter). Optionally, if the *clustermask* parameter is not NULL, it will store at the end of the function execution a 6-bit bitmask of the triggered trigger clusters.

NOTE: This function is optimized to cache the full trigger configuration for a given run, so that the database is not queried again if the last call to this function used the same run number. Calling it with *run* parameter equal to zero clears the cache.

Returns Upon successful completion, this function returns a value of zero. Otherwise, the following value will be returned:

-1: no trigger cluster is triggered by the given trigger classes.

> 0: error code with a value equal to the line number where the error occurred.

DAQlogbook_query_getDetectorIdsFromTriggerCluster

Synopsis

```
#include "DAQlogbook.h"

int DAQlogbook_query_getDetectorIdsFromTriggerCluster
(unsigned int run, unsigned int cluster, unsigned int
*detectorMask)
```

Description Build a 24-bit detector IDs bitmask (LSB = detector ID zero, stored in the *detectorMask* parameter) corresponding to the readout detectors of the given trigger cluster.

NOTE: This function is optimized to cache the full trigger configuration for a given run, so that the database is not queried again if the last call to this function used the same run number. Calling it with *run* parameter equal to zero clears the cache.

Returns Upon successful completion, this function returns a value of zero. Otherwise, the following value will be returned:

-1: given trigger cluster not found.

> 0: error code with a value equal to the line number where the error occurred.

DAQlogbook_query_getDetectors

Synopsis

```
#include "DAQlogbook.h"

int DAQlogbook_query_getDetectors(unsigned int run, unsigned
int *detectormask)
```

Description Build a 24-bit detector IDs bitmask (LSB = detector ID zero, stored in the *detectormask* parameter) corresponding to the readout detectors participating in the given run.

Returns Upon successful completion, this function returns a value of zero. Otherwise, -1 will be returned.

DAQlogbook_query_getPartition

Synopsis

```
#include "DAQlogbook.h"

int DAQlogbook_query_getPartition(unsigned int run, char
**partition, int *standalone)
```

Description Retrieve the name of the ECS partition for the given run (stored in the *partition* parameter). In case of a standalone run, the value will be equal to the detector name.

Additionally, at the end of the execution of the function, the *standalone* parameter will indicate if it's a standalone or a global run.

Returns Upon successful completion, this function returns a value of zero. Otherwise, an error code with a value equal to the line number where the error occurred will be returned.

DAQlogbook_query_getDetectorsInTriggerClasses

Synopsis

```
#include "DAQlogbook.h"

int DAQlogbook_query_getDetectorsInTriggerClasses(unsigned
int run, char **table)
```

Description Build a 50x24 boolean table (*table* parameter) indicating, for each detector ID, which trigger classes will trigger it in the given run.

The table indexes are of the form `triggerClassId * 24 + detectorID`. As an example, if detector ID equal to 5 is triggered by trigger class equal to 20:

```
table[20 * 24 + 5] = 1
```

Returns Upon successful completion, this function returns a value of zero. Otherwise, an error code with a value equal to the line number where the error occurred will be returned.

DAQlogbook_query_getTriggerClassNames

Synopsis

```
#include "DAQlogbook.h"

int DAQlogbook_query_getTriggerClassNames(unsigned int run,
char ***table)
```

Description Build a 50-entries table (`table` parameter) indicating, for each possible trigger class ID, the corresponding trigger class name in the given run.

The table indexes are the trigger class IDs (from 0 to 49) and the values are the trigger class names. A value of `NULL` means that the corresponding trigger class ID is undefined for the given run.

Returns Upon successful completion, this function returns a value of zero. Otherwise, an error code with a value equal to the line number where the error occurred will be returned.

DAQlogbook_query_getTriggerClassIdFromName

Synopsis

```
#include "DAQlogbook.h"

int DAQlogbook_query_getTriggerClassIdFromName(unsigned int
run, const char *className, unsigned char *classId)
```

Description Retrieve, for a given trigger class name, the corresponding trigger class ID (stored in the *classId* parameter) in the given run.

If more than 1 match is found, the first one is retrieved.

Returns Upon successful completion, this function returns a value of zero. Otherwise, the following value will be returned:

-1: given trigger class name undefined for this run.

-10: more than 1 match found.

> 0: error code with a value equal to the line number where the error occurred.

DAQlogbook_query_getDetectorsInTriggerInput

Synopsis

```
#include "DAQlogbook.h"

int DAQlogbook_query_getDetectorsInTriggerInput(unsigned int
run, char **table)
```

Description Build a 24-entries boolean table (*table* parameter) indicating, for each detector ID, if the corresponding detector is a trigger detector in the run.

Returns Upon successful completion, this function returns a value of zero. Otherwise, an error code with a value equal to the line number where the error occurred will be returned.

DAQlogbook_query_getL2aPerTriggerClass

Synopsis

```
#include "DAQlogbook.h"

int DAQlogbook_query_getL2aPerTriggerClass(unsigned int run,
int **L2a)
```

Description Build a 50-entries table (*L2a* parameter) indicating, for each trigger class ID, the number of L2 accept decisions in the given run.

The table indexes are the trigger class IDs (from 0 to 49) and the values are the number of L2 accept decisions. A value of `-1` means that the corresponding trigger class ID is undefined for the run.

Returns Upon successful completion, this function returns a value of zero. Otherwise, an error code with a value equal to the line number where the error occurred will be returned.

DAQlogbook_query_getHLTDecisionsPerTriggerClass

Synopsis

```
#include "DAQlogbook.h"

int DAQlogbook_query_getHLTDecisionsPerTriggerClass(unsigned
int run, int **hltAccepts, int **hltPartialAccepts, int
**hltOnly, int **hltRejects)
```

Description Build four 50-entries tables (*hltAccepts*, *hltPartialAccepts*, *hltOnly* and *hltRejects* parameters) indicating, for each trigger class ID, the number of different HLT decisions in the given run.

The table indexes are the trigger class IDs (from 0 to 49) and the values are the number of HLT decisions. A value of −1 means that the corresponding trigger class ID is undefined for the run.

The tables will store, at the end of the execution of the function, the following information:

- *hltAccepts*: number of full event accept decisions per trigger class ID.
- *hltPartialAccepts*: number of partial readout decisions per trigger class ID.
- *hltOnly*: number of "hltOnly" decisions per trigger class ID.
- *hltRejects*: number of full event reject decisions per trigger class ID.

Returns Upon successful completion, this function returns a value of zero. Otherwise, an error code with a value equal to the line number where the error occurred will be returned.

DAQlogbook_get_DAQ_active_components

Synopsis

```
#include "DAQlogbook.h"

int DAQlogbook_get_DAQ_active_components(unsigned int
run,const char* type,void **mask_of_ids,int *n_bytes)
```

Description Retrieve the list of active DAQ components of a given type in the given run. Possible values for the *type* parameter are: DDL, LDC, GDC.

At the end of the execution of the function, the *mask_of_ids* parameter will store a bitmask of the active components'IDs as defined in the ***DATE_CONFIG*** database.

Returns Upon successful completion, this function returns a value of zero. Otherwise, an error code with a value equal to the line number where the error occurred will be returned.

DAQlogbook_query_runType

Synopsis

```
#include "DAQlogbook.h"

int DAQlogbook_query_runType(unsigned int run, char
**runType)
```

Description Retrieve, for a given run, the corresponding ECS run type (stored in the *runType* parameter).

Returns Upon successful completion, this function returns a value of zero. Otherwise, an error code with a value equal to the line number where the error occurred will be returned.

DAQlogbook_getLastRun

Synopsis

```
#include "DAQlogbook.h"

int DAQlogbook_getLastRun(const char *partition, unsigned int
*run, unsigned char *active)
```

Description Retrieve, for a given ECS partition or detector (*partition* parameter), the number of the last run started (stored in the *run* parameter). The *active* parameter will store, at the end of the execution of the function, if the retrieved run is still active or not.

Returns Upon successful completion, this function returns a value of zero. Otherwise, an error code with a value equal to the line number where the error occurred will be returned.

DAQlogbook_getActiveRuns

Synopsis

```
#include "DAQlogbook.h"

int DAQlogbook_getActiveRuns(unsigned int **runs, unsigned
int *size)
```

Description Build an array with the number of the runs currently active (stored in *runs* parameter). The *size* parameter will store the number of runs found.

Returns Upon successful completion, this function returns a value of zero. Otherwise, an error code with a value equal to the line number where the error occurred will be returned.

DAQlogbook_getAmoreAgents

Synopsis

```
#include "DAQlogbook.h"

int DAQlogbook_getAmoreAgents(unsigned int run, char
***agents, unsigned int *size)
```

Description Build a list of active ***AMORE*** agents (stored in *agents* parameter) for a given run. The *size* parameter will store the number of agents found.

Returns Upon successful completion, this function returns a value of zero. Otherwise, an error code with a value equal to the line number where the error occurred will be returned.

24.3.6 eLogbook WRITE functions

Below is a list of functions providing WRITE access to the eLogbook database.

DAQlogbook_update_newRun

Synopsis

```
#include "DAQlogbook.h"

int DAQlogbook_update_newRun(unsigned int run, const char
*partition, unsigned int ndetectors, const char **detectors,
const char *runtype, unsigned int calibration, int
checkOldRuns)
```

Description This function should be called when a new run is started. It creates a new entry in the ***logbook*** table and initializes several fields of this table: ***run***, ***time_created***, ***partition***, ***detector***, ***run_type***, ***calibration***, ***numberOfDetectors*** and ***detectorMask***. Additionally, it also creates one entry in the ***logbook_detectors*** table for each detector participating in the run and initializes one entry in the ***logbook_daq_active_components*** table with the run number.

If the *checkOldRuns* flag is set, the function will first close all active runs having as participating detector(s) any of the ones participating in this run.

Returns Upon successful completion, this function returns a value of zero. Otherwise, an error code with a value equal to the line number where the error occurred will be returned.

DAQlogbook_update_detectorMask

Synopsis

```
#include "DAQlogbook.h"

int DAQlogbook_update_detectorMask(unsigned int run)
```

Description Update the ***detectorMask*** and ***numberOfDetectors*** fields of the ***logbook*** table, based on the content of the ***logbook_detectors*** table.

It is called automatically by the ***DAQlogbook_update_newRun*** function.

Returns Upon successful completion, this function returns a value of zero. Otherwise, an error code with a value equal to the line number where the error occurred will be returned.

DAQlogbook_update_EndRun

Synopsis

```
#include "DAQlogbook.h"

int DAQlogbook_update_EndRun (unsigned int run, int
ecs_success, int  daq_success, const char * const  eor_reason)
```

Description This function should be called when a run finishes. It populates the ***ecs_success***, ***daq_success*** and ***eor_reason*** fields of the ***logbook*** table. It also:

- populates the ***logbook_shuttle*** table with the `UNPROCESSED` value for each detector participating in the run.
- changes all files (related to the run) entries from the ***logbook_stats_files*** table still in `Writing` to `Closed`.
- updates the ***dataMigration*** flag from the ***logbook*** table.
- updates the different counters by executing the ***update_logbook_counters*** stored procedure.

Returns Upon successful completion, this function returns a value of zero. Otherwise, an error code with a value equal to the line number where the error occurred will be returned.

DAQlogbook_update_startTime

Synopsis

```
#include "DAQlogbook.h"

int DAQlogbook_update_startTime(unsigned int run)
```

Description Update the ***DAQ_time_start*** field of the ***logbook*** table with the current timestamp. It also adds a Log Entry of class `PROCESS`, marking the start of data taking.

Returns Upon successful completion, this function returns a value of zero. Otherwise, an error code with a value equal to the line number where the error occurred will be returned.

DAQlogbook_update_stopTime

Synopsis

```
#include "DAQlogbook.h"

int DAQlogbook_update_stopTime(unsigned int run)
```

Description Update the ***DAQ_time_end*** and the ***runDuration*** fields of the logbook table. It also adds a Log Entry of class `PROCESS`, marking the end of data taking.

Returns Upon successful completion, this function returns a value of zero. Otherwise, an error code with a value equal to the line number where the error occurred will be returned.

DAQlogbook_update_DAQrunning

Synopsis

```
#include "DAQlogbook.h"

int DAQlogbook_update_DAQrunning(unsigned int run)
```

Description Update the ***time_update*** field of the ***logbook*** table with the current timestamp. It should be called periodically during data taking, serving as heartbeat of the run and allowing the detection of crashed or not properly terminated runs.

It will also update the different counters by executing the ***update_logbook_counters*** stored procedure.

Returns Upon successful completion, this function returns a value of zero. Otherwise, an error code with a value equal to the line number where the error occurred will be returned.

DAQlogbook_update_DAQnode_config

Synopsis

```
#include "DAQlogbook.h"

int DAQlogbook_update_DAQnode_config(unsigned int run,int
LDCs,int GDCs, int LDCrecordingMode, int GDCrecordingMode)
```

Description Update the ***logbook*** table with the DAQ nodes configuration, populating the following fields: ***numberOfLDCs***, ***numberOfGDCs***, ***LDClocalRecording***, ***GDClocalRecording***, ***GDCmStreamRecording*** and ***eventBuilding***.

It will also initialize the ***dataMigrated*** field based on the active recording configuration.

Returns Upon successful completion, this function returns a value of zero. Otherwise, an error code with a value equal to the line number where the error occurred will be returned.

DAQlogbook_update_DAQnode_statsGDC

Synopsis

```
#include "DAQlogbook.h"

int DAQlogbook_update_DAQnode_statsGDC(unsigned int run, char
*GDC, unsigned long long eventCount, unsigned long long
eventCountPhysics, unsigned long long eventCountCalibration,
unsigned long long bytesRecorded, unsigned long long
bytesRecordedPhysics, unsigned long long
bytesRecordedCalibration)
```

Description Update the counters in the ***logbook_stats_GDC*** table. When called for the first time for the given run and GDC pair, it will create a new entry in the table.

Returns Upon successful completion, this function returns a value of zero. Otherwise, an error code with a value equal to the line number where the error occurred will be returned.

DAQlogbook_insert_DAQnode_statsLDC

Synopsis

```
#include "DAQlogbook.h"

int DAQlogbook_insert_DAQnode_statsLDC(unsigned int run, char
*LDC, unsigned int detectorId, unsigned long long eventCount,
unsigned long long eventCountPhysics, unsigned long long
eventCountCalibration, unsigned long long bytesInjected,
unsigned long long bytesInjectedPhysics, unsigned long long
bytesInjectedCalibration)
```

Description Create a new entry in the ***logbook_stats_LDC*** table for the given run and LDC pair, initializing the different counters to the specified values. Subsequent changes to this entry should be done via the ***DAQlogbook_update_DAQnode_statsLDC*** function.

Returns Upon successful completion, this function returns a value of zero. Otherwise, an error code with a value equal to the line number where the error occurred will be returned.

DAQlogbook_update_DAQnode_statsLDC

Synopsis

```
#include "DAQlogbook.h"

int DAQlogbook_update_DAQnode_statsLDC(unsigned int run, char
*LDC, unsigned long long eventCount, unsigned long long
eventCountPhysics, unsigned long long eventCountCalibration,
unsigned long long bytesInjected, unsigned long long
bytesInjectedPhysics, unsigned long long
bytesInjectedCalibration)
```

Description Update the counters in the ***logbook_stats_LDC*** table for the given run and LDC pair previously created via the ***DAQlogbook_insert_DAQnode_statsLDC*** function.

Returns Upon successful completion, this function returns a value of zero. Otherwise, an error code with a value equal to the line number where the error occurred will be returned.

DAQlogbook_update_DAQnode_statsLDC_trgCluster

Synopsis

```
#include "DAQlogbook.h"

int DAQlogbook_update_DAQnode_statsLDC_trgCluster(unsigned
int run, char *LDC, unsigned char clusterId, unsigned long
long eventCount, unsigned long long bytesInjected)
```

Description Updates the counters in the ***logbook_stats_LDC_trgCluster*** table. When called the first time for a given run, LDC and trigger cluster ID tuple, it will create a new entry in the table.

Returns Upon successful completion, this function returns a value of zero. Otherwise, an error code with a value equal to the line number where the error occurred will be returned.

DAQlogbook_update_DAQ_active_components

Synopsis

```
#include "DAQlogbook.h"

int DAQlogbook_update_DAQ_active_components(unsigned int run,
const char* type, void *mask_of_ids, int n_bytes)
```

Description Update one of the fields of the ***logbook_daq_active_components*** table. The *type* parameter defines which field is updated and can be one of the following: `LDC`, `GDC`, `DDL`.

Returns Upon successful completion, this function returns a value of zero. Otherwise, an error code with a value equal to the line number where the error occurred will be returned.

DAQlogbook_update_ECS_iteration

Synopsis

```
#include "DAQlogbook.h"

int DAQlogbook_update_ECS_iteration(unsigned int run,
unsigned int currentIteration, unsigned int totalIterations)
```

Description Update the ***ecs_iteration_current*** and ***ecs_iteration_total*** fields of the ***logbook*** table.

Returns Upon successful completion, this function returns a value of zero. Otherwise, an error code with a value equal to the line number where the error occurred will be returned.

DAQlogbook_datafile_new

Synopsis

```
#include "DAQlogbook.h"

int DAQlogbook_datafile_new(unsigned int run, const char
*filePath, unsigned int local)
```

Description Create a new entry in the ***logbook_stats_files*** table for the given run, populating the ***run***, ***fileName***, ***location***, ***local***, ***rolename***, ***hostname***, ***pid*** and ***time_write_begin*** fields. It should be called when a new data file is created.

The returned value should be used as an ID for further function calls related with data files statistics or status.

Returns Upon successful completion, this function returns the ID of the created table entry. Otherwise, an error code with a value equal to the line number where the error occurred will be returned.

DAQlogbook_datafile_update_size

Synopsis

```
#include "DAQlogbook.h"

int DAQlogbook_datafile_update_size(int id, unsigned long
long size, unsigned long long events)
```

Description Update the ***size*** and ***eventCount*** fields of the ***logbook_stats_files*** table for a given data file ID (as returned by ***DAQlogbook_datafile_new***).

Returns Upon successful completion, this function returns a value of zero. Otherwise, an error code with a value equal to the line number where the error occurred will be returned.

DAQlogbook_datafile_update_location

Synopsis

```
#include "DAQlogbook.h"

int DAQlogbook_datafile_update_location(int id, const char
*new_location)
```

Description Update the ***location*** field of the ***logbook_stats_files*** table for a given data file ID (as returned by ***DAQlogbook_datafile_new***).

Returns Upon successful completion, this function returns a value of zero. Otherwise, an error code with a value equal to the line number where the error occurred will be returned.

DAQlogbook_datafile_setStatus_closed

Synopsis

```
#include "DAQlogbook.h"

int DAQlogbook_datafile_setStatus_closed(int id)
```

Description Update the following fields of the ***logbook_stats_files*** table:

- ***status***: `Closed`.
- ***time_write_end***: current timestamp.

It should be called when a data file with the given ID (as returned by ***DAQlogbook_datafile_new***) is closed.

Returns Upon successful completion, this function returns a value of zero. Otherwise, an error code with a value equal to the line number where the error occurred will be returned.

DAQlogbook_datafile_setStatus_waitingMigrationRequest

Synopsis

```
#include "DAQlogbook.h"

int DAQlogbook_datafile_setStatus_waitingMigrationRequest(int
id)
```

Description Update the following fields of the ***logbook_stats_files*** table:

- ***status***: `Waiting migration request`.
- ***time_write_end***: current timestamp.

It should be called when a data file with the given ID (as returned by ***DAQlogbook_datafile_new***) is ready for migration.

Returns Upon successful completion, this function returns a value of zero. Otherwise, an error code with a value equal to the line number where the error occurred will be returned.

DAQlogbook_datafile_setStatus_migrationRequested

Synopsis

```
#include "DAQlogbook.h"

int DAQlogbook_datafile_setStatus_migrationRequested(int id)
```

Description Update the following fields of the ***logbook_stats_files*** table:

- ***status***: Migration request.
- ***time_migrate_request***: current timestamp.

It should be called when a data file with the given ID (as returned by ***DAQlogbook_datafile_new***) is marked for migration.

Returns Upon successful completion, this function returns a value of zero. Otherwise, an error code with a value equal to the line number where the error occurred will be returned.

DAQlogbook_datafile_setStatus_migrating

Synopsis

```
#include "DAQlogbook.h"

int DAQlogbook_datafile_setStatus_migrating(int id)
```

Description Update the following fields of the ***logbook_stats_files*** table:

- ***status***: Migrating.
- ***time_migrate_begin***: current timestamp.

It should be called when a data file with the given ID (as returned by ***DAQlogbook_datafile_new***) starts to be migrated.

Returns Upon successful completion, this function returns a value of zero. Otherwise, an error code with a value equal to the line number where the error occurred will be returned.

DAQlogbook_datafile_setStatus_migrated

Synopsis

```
#include "DAQlogbook.h"

int DAQlogbook_datafile_setStatus_migrated(int id)
```

Description Update the following fields of the ***logbook_stats_files*** table:

- ***status***: Migrated.
- ***time_migrate_end***: current timestamp.

It should be called when a data file with the given ID (as returned by ***DAQlogbook_datafile_new***) finishes being migrated.

Returns Upon successful completion, this function returns a value of zero. Otherwise, an error code with a value equal to the line number where the error occurred will be returned.

DAQlogbook_datafile_updateRunStatus

Synopsis

```
#include "DAQlogbook.h"
```

```
int DAQlogbook_datafile_updateRunStatus(unsigned int run)
```

Description Update the ***dataMigrated*** field of the ***logbook*** table for the given run. The new value is based on the status of the run's data files stored in the ***logbook_stats_files*** table. Only non-local files are taken into account.

Returns Upon successful completion, this function returns a value of zero. Otherwise, an error code with a value equal to the line number where the error occurred will be returned.

DAQlogbook_update_runQuality

Synopsis

```
#include "DAQlogbook.h"

int DAQlogbook_update_runQuality(unsigned int run, const char
*runQuality)
```

Description Update the value of the ***runQuality*** field of the ***logbook*** table for the given run.

Returns Upon successful completion, this function returns a value of zero. Otherwise, an error code with a value equal to the line number where the error occurred will be returned.

DAQlogbook_update_cluster

Synopsis

```
#include "DAQlogbook.h"

int DAQlogbook_update_cluster (unsigned int run, unsigned int
cluster, unsigned int detectorMask, const char *partition,
unsigned int inputDetectorMask, unsigned long long
triggerClassMask)
```

Description Create a new entry in the ***logbook_trigger_clusters*** table, thus declaring a new trigger cluster for the given run. The *detectorMask* and *inputDetectorMask* parameters should be a 24-bit detector ID bitmask of the detectors participating in the given cluster as readout detectors and as trigger detectors, respectivly. The *triggerClassMask* parameter should be a 50-bit trigger classes ID bitmask of the trigger classes defined for the given cluster.

Returns Upon successful completion, this function returns a value of zero. Otherwise, -1 will be returned.

DAQlogbook_update_triggerConfig

Synopsis

```
#include "DAQlogbook.h"
```

```
int DAQlogbook_update_triggerConfig (unsigned int run, const
char * const configurationFile, const char * const
alignmentFile)
```

Description Create a new entry in the ***logbook_trigger_config*** table, thus registering the trigger configuration and the alignment settings for the given run.

Returns Upon successful completion, this function returns a value of zero. Otherwise, an error code with a value equal to the line number where the error occurred will be returned.

DAQlogbook_update_triggerClassName

Synopsis

```
#include "DAQlogbook.h"

int DAQlogbook_update_triggerClassName(unsigned int run,
unsigned char classId, const char *className, unsigned int
classGroupId, float classGroupTime)
```

Description Create a new entry in the ***logbook_trigger_classes*** table, thus registering a new trigger class for the given run. The *classId* parameter should be the corresponding bit number of the trigger class (as defined in the 50-bit trigger classes bitmask) and the *className* parameter should be the full trigger class name as defined by the Trigger Coordination.

Returns Upon successful completion, this function returns a value of zero. Otherwise, -1 will be returned.

DAQlogbook_update_triggerClassCounter

Synopsis

```
#include "DAQlogbook.h"

int DAQlogbook_update_triggerClassCounter(unsigned int run,
unsigned char classId, unsigned long long L0bCount, unsigned
long long L0aCount, unsigned long L1bCount, unsigned long
L1aCount, unsigned long L2bCount, unsigned long L2aCount,
float ctpDuration)
```

Description Update the ***L0b***, ***L0a***, ***L1b***, ***L1a***, ***L2b***, ***L2a*** and ***ctpDuration*** fields of the ***logbook_trigger_classes*** table for the given run and trigger class ID pair. It should be called only after the corresponding trigger class ID has been registered using the ***DAQlogbook_update_triggerClassName*** function.

Returns Upon successful completion, this function returns a value of zero. Otherwise, -1 will be returned.

DAQlogbook_insert_triggerInput

Synopsis

```
#include "DAQlogbook.h"

int DAQlogbook_insert_triggerInput(unsigned int run, unsigned
int inputId, const char *inputName, unsigned int inputLevel)
```

Description Create a new entry in the ***logbook_trigger_inputs*** table, thus registering a new trigger input for the given run41.

Returns Upon successful completion, this function returns a value of zero. Otherwise, -1 will be returned.

DAQlogbook_update_triggerInputCounter

Synopsis

```
#include "DAQlogbook.h"

int DAQlogbook_update_triggerInputCounter(unsigned int run,
unsigned int inputId, unsigned int inputLevel, unsigned long
long inputCount)
```

Description Update the ***inputCount*** field of the ***logbook_trigger_inputs*** table for the given run and trigger input (represented by the *inputId* and *inputLevel* pair). It should be called only after the corresponding trigger input has been registered using the ***DAQlogbook_insert_triggerInput*** function.

Returns Upon successful completion, this function returns a value of zero. Otherwise, -1 will be returned.

DAQlogbook_update_triggerDetectorCounter

Synopsis

```
#include "DAQlogbook.h"

int DAQlogbook_update_triggerDetectorCounter(unsigned int
run, const char *detector, unsigned long L2aCount)
```

Description Update the ***L2a*** field of the ***logbook_detectors*** table for the given run and detector pair.

Returns Upon successful completion, this function returns a value of zero. Otherwise, -1 will be returned.

DAQlogbook_update_triggerGlobalCounter

Synopsis

```
#include "DAQlogbook.h"
```

```
int DAQlogbook_update_triggerGlobalCounter(unsigned int run,
unsigned long L2aCount, float ctpDuration)
```

Description Update the ***L2a*** and ***ctpDuration*** fields of the ***logbook*** table for the given run.

Returns Upon successful completion, this function returns a value of zero. Otherwise, -1 will be returned.

DAQlogbook_update_setCTPbitInDetectorMask

Synopsis

```
#include "DAQlogbook.h"

int DAQlogbook_update_setCTPbitInDetectorMask(unsigned int
run)
```

Description Set to 1, for the given run, the bit in the ***detectorMak*** field of the ***logbook*** table corresponding to the CTP detector ID (represented by the 3-letter code `TRI`).

Returns Upon successful completion, this function returns a value of zero. Otherwise, an error code with a value equal to the line number where the error occurred will be returned.

DAQlogbook_insert_AMORE_agent

Synopsis

```
#include "DAQlogbook.h"

int DAQlogbook_insert_AMORE_agent(unsigned int run, const
char *detector, const char *agentname, const char
*agentversion, const char *params)
```

Description Create a new entry in the ***logbook_AMORE_agents*** table, thus registering an ***AMORE*** agent as being active for the given run.

Returns Upon successful completion, this function returns a value of zero. Otherwise, an error code with a value equal to the line number where the error occurred will be returned.

DAQlogbook_update_AMORE_agent

Synopsis

```
#include "DAQlogbook.h"

int DAQlogbook_update_AMORE_agent(unsigned int run, const
char *agentname, unsigned int mo_published, unsigned int
mo_v_published, unsigned long long bytes_published, float
aver_cpu_time, float aver_real_time)
```

Description Update the statistics of the given ***AMORE*** agent for the given run.

Returns Upon successful completion, this function returns a value of zero. Otherwise, an error code with a value equal to the line number where the error occurred will be returned.

DAQlogbook_update_AMORE_agent_summary_img

Synopsis

```
#include "DAQlogbook.h"

int DAQlogbook_update_AMORE_agent_summary_img(unsigned int
run, const char *agentname, char *summary_img, unsigned long
n_bytes)
```

Description Update the Quality Assurance summary image of the given ***AMORE*** agent for the given run.

Returns Upon successful completion, this function returns a value of zero. Otherwise, an error code with a value equal to the line number where the error occurred will be returned.

DAQlogbook_update_HLTmode

Synopsis

```
#include "DAQlogbook.h"

int DAQlogbook_update_HLTmode(unsigned int run, const char
*HLTmode)
```

Description Update the ***HLTmode*** field of the ***logbook*** table for the given run. It also sets to 1 the bit in the ***detectorMak*** field of the ***logbook*** table corresponding to the HLT detector ID (represented by the 3-letter code `HLT`) if the given HLT mode is set to `B` or `C`.

Returns Upon successful completion, this function returns a value of zero. Otherwise, an error code with a value equal to the line number where the error occurred will be returned.

DAQlogbook_update_local_HLT_stats

Synopsis

```
#include "DAQlogbook.h"

int DAQlogbook_update_local_HLT_stats(unsigned int run, char
*LDC, unsigned long hltAccepts, unsigned long hltRejects,
unsigned long long hltBytesRejected, unsigned long long
*hltBytesRejectedPerTriggerClass)
```

Description Update the HLT statistics per detector LDC (stored in the ***logbook_stats_LDC*** table) and the number of bytes rejected - following an HLT decision - per trigger class (stored in the ***logbook_trigger_classes*** table) for the given run.

Returns Upon successful completion, this function returns a value of zero. Otherwise, an error code with a value equal to the line number where the error occurred will be returned.

DAQlogbook_insert_HLT_stats

Synopsis

```
#include "DAQlogbook.h"

int DAQlogbook_insert_HLT_stats(unsigned int run,char *LDC)
```

Description Create a new entry in the ***logbook_stats_HLT_LDC*** table, thus registering an HLT LDC for the given run.

Returns Upon successful completion, this function returns a value of zero. Otherwise, an error code with a value equal to the line number where the error occurred will be returned.

DAQlogbook_update_HLT_stats

Synopsis

```
#include "DAQlogbook.h"

int DAQlogbook_update_HLT_stats(unsigned int run, char *LDC,
unsigned long hltAccepts, unsigned long hltPartialAccepts,
unsigned long hltOnly, unsigned long hltRejects, unsigned
long *hltAcceptsPerTriggerClass, unsigned long
*hltPartialAcceptsPerTriggerClass, unsigned long
*hltOnlyPerTriggerClass, unsigned long
*hltRejectsPerTriggerClass)
```

Description Update the HLT decisions per HLT LDC (stored in the ***logbook_stats_HLT_LDC*** table) and per trigger class (stored in the ***logbook_trigger_classes*** table) for the given run.

Returns Upon successful completion, this function returns a value of zero. Otherwise, an error code with a value equal to the line number where the error occurred will be returned.

DAQlogbook_add_comment

Synopsis

```
#include "DAQlogbook.h"

int DAQlogbook_add_comment(unsigned int run, const char
*title, const char * const comment,...)
```

Description Insert a new Log Entry of class `PROCESS` on the ***logbook_comments*** table. The only inserted fields are ***run***, ***class***, ***title*** and ***comment***.

The *run* parameter is optional.

Returns Upon successful completion, this function returns a value of zero. Otherwise, an error code with a value equal to the line number where the error occurred will be returned.

24.4 Logbook Daemon

The ***logbookDaemon*** is a daemon process that extracts data concerning the ALICE magnets and the LHC configuration published by the DCS via DIM and inserts it in the DB at start of run. The subscribed DIM services and the corresponding DB fields are listed in Table 24.30.

Table 24.30 ***logbookDaemon*** DIM services and DB fields relationship.

DIM Service	DB Field
DCS_GRP_L3MAGNET_CURRENT	***L3_magnetCurrent*** field of the ***logbook*** table
DCS_GRP_DIPOLE_CURRENT	***Dipole_magnetCurrent*** field of the ***logbook*** table
DCS_GRP_L3MAGNET_POLARITY	No field, this value is used to complete the value provided by the DCS_GRP_L3MAGNET_CURRENT service
DCS_GRP_DIPOLE_POLARITY	No field, this value is used to complete the value provided by the DCS_GRP_DIPOLE_CURRENT service
ALICEDAQ_LHCMachineMode	***beamType*** field of the ***logbook*** table
ALICEDAQ_LHCBeamEnergy	***beamEnergy*** field of the ***logbook*** table
ALICEDAQ_LHCFillNumber	***LHCFillNumber*** field of the ***logbook*** table
ALICEDAQ_LHCTotBunchInteract	***LHCTotalInteractingBunches*** field of the ***logbook*** table
ALICEDAQ_LHCTotBunchNotInteractBeam1	***LHCTotalNonInteractingBunchesBeam1*** field of the ***logbook*** table
ALICEDAQ_LHCTotBunchNotInteractBeam2	***LHCTotalNonInteractingBunchesBeam2*** field of the ***logbook*** table

Additionally, it also provides a publish mechanism (via DIM) to notify Start of Run and End of Run events of partitions and detectors, thus avoiding the need from the

different online subsystems processes to constantly poll the DB. The available DIM services are listed in Table 24.31.

Table 24.31 ***logbookDaemon*** run events DIM services

DIM Service	Description
/LOGBOOK/SUBSCRIBE/ECS_SOR_${PART}	ECS Start of Run per partition (replacing ${PART} by the partition name)
/LOGBOOK/SUBSCRIBE/ECS_SOR_${DET}	ECS Start of Run per detector (replacing ${DET} by the 3-letter detector code)
/LOGBOOK/SUBSCRIBE/ECS_EOR_${PART}	ECS End of Run per partition (replacing ${PART} by the partition name)
/LOGBOOK/SUBSCRIBE/ECS_EOR_${DET}	ECS End of Run per detector (replacing ${DET} by the 3-letter detector code)
/LOGBOOK/SUBSCRIBE/DAQ_SOR_${PART}	DAQ Start of Run per partition (replacing ${PART} by the partition name)
/LOGBOOK/SUBSCRIBE/DAQ_SOR_${DET}	DAQ Start of Run per detector (replacing ${DET} by the 3-letter detector code)
/LOGBOOK/SUBSCRIBE/DAQ_EOR_${PART}	DAQ End of Run per partition (replacing ${PART} by the partition name)
/LOGBOOK/SUBSCRIBE/DAQ_EOR_${DET}	DAQ Start of Run per detector (replacing ${DET} by the 3-letter detector code)

24.5 Tools

Below is a list of the available command-line tools providing access to the eLogbook's repository.

insert_file

Synopsis `insert_file HOSTNAME USERNAME PASSWORD DATABASE COMMENTID FILEID FILENAME SIZE TITLE CONTENT_TYPE`

Description Insert a file in the ***logbook_files*** table attached to an already existing Log Entry.

Parameters:

- `HOSTNAME`: MySQL server hostname.
- `USERNAME`: MySQL username.
- `PASSWORD`: MySQL password.
- `DATABASE`: MySQL database name.
- `COMMENTID`: ID of the Log Entry to which the file should be attached to (corresponding to the ***`commentid`*** field of the ***`logbook_files`*** table.
- `FILEID`: ID of the file (corresponding to the ***`fileid`*** field of the ***`logbook_files`*** table.
- `FILENAME`: full filename (including path).
- `SIZE`: file size in bytes.
- `TITLE`: file title.
- `CONTENT_TYPE`: file Content Type.

Returns Upon successful completion, this command returns a value of zero. Otherwise, 1 will be returned.

logbookCloseRun

Synopsis `logbookCloseRun RUNNUMBER`

Description Close a run not properly terminated (“zombie” run), calling the ***`DAQlogbook_update_EndRun`*** function of the C API.

Parameters:

- `RUNNUMBER`: Run number.

Returns Upon successful completion, this command returns a value of zero. Otherwise, 1 will be returned.

logbookGetTriggerInfo

Synopsis `logbookGetTriggerInfo RUNNUMBER`

Description Fetche and print the trigger information related with the given run.

Parameters:

- `RUNNUMBER`: Run number.

Returns Upon successful completion, this command returns a value of zero. Otherwise, 1 will be returned.

logbookShellAPI

Synopsis `logbookGetTriggerInfo -c COMMAND [-r RUN] [-v] [-h]`

Description Fetche and print information related with a specific run.

Parameters:

- `-c COMMAND`: defines which information should be printed. `COMMAND` can be one of:
  - `getRuntype`: prints the ECS run type.
  - `getActiveGDCs`: prints the IDs of the active GDCs.
  - `getActiveLDCs`: prints the IDs of the active LDCs.
  - `getActiveDDLs`: prints the IDs of the active DDLs.
- `-r RUN`: Run number. If not provided, the run number is taken from the ***DATE_RUN_NUMBER*** environment variable.
- `-v`: execute in verbose mode.
- `-h`: print help.

Returns Upon successful completion, this command returns a value of zero. Otherwise, an error code with a value equal to the line number where the error occurred will be returned.

DAQlogbook_dim_gateway

Synopsis `DAQlogbook_dim_gateway -s DIM_SERVICE -m DIM_MODE [-o]`

Description Implement an interface between DIM and the eLogbook. Received messages are inserted as Log Entries.

Parameters:

- `-s DIM_SERVICE`: DIM service name.
- `-m DIM_MODE`: defines the operation mode. `DIM_MODE` can be one of:
  - `subscribe`: subscribe to external server service and insert a Log Entry at each service update.
  - `command`: create a DIM command service and insert a Log Entry at each remote DIM client command execution.
- `-o`: don't run as a daemon (by default, the command will run as a daemon).

Returns Upon successful completion, this command returns a value of zero. Otherwise, -1 will be returned.

24.6 Graphical User Interface

24.6.1 Overview

The eLogbook's Web-based GUI (available at https://cern.ch/alice-logbook) was developed using modern Web technologies, including PHP5, Javascript and Cascading Style Sheets (CSS). It is hosted on an Apache web server and can be accessed from the experimental area (inside the experiment's technical network), the CERN General Purpose Network (GPN) and the internet.

24.6.2 Authentication and Authorization

Authentication is implemented via the CERN Authentication central service, providing Single Sign On (SSO) and removing the effort of authenticating the users from the eLogbook software. This way, when a user tries to access the GUI, he is redirected to the CERN Login page where it has to provide his credentials. If successful, he is then redirected back to the GUI.

Authorization is implemented in the GUI with 5 different levels of privileges:

- NONE: no access the GUI
- READ: read-only access
- WRITE: read/write access (e.g. can write Log Entries)
- ADMIN: same as previous + can grant WRITE privileges
- SUPER: same as previous + can grant ADMIN privileges

At the first login, the user's details are stored in the ***logbook_users*** table. Additionally, READ privilege is given by default.

24.6.3 Features

Below is a brief description of the main features of the eLogbook's GUI.

24.6.3.1 Run Statistics

The Run Statistics section provides users access to both data-taking statistics and conditions, ranging from event and data rates to trigger configurations and LHC parameters. It is presented in a tabular view, with different subsections grouped in individual tabs.

Given the number of available fields, some tabs allow users to select which fields should be displayed.

Additionally, there's an Overview tab which allows users to aggregate some of the data-taking statistics by different criteria (such as number of detectors or ECS partition) or as a function of time.

24.6.3.2 Run Details

The Run Details section provides users access to all the available information concerning a specific run, including ***infoLogger*** messages and ***AMORE*** histograms.

24.6.3.3 Log Entries

The Log Entries section allows users to read or create reports related to the ALICE operations. The inserted Log Entries can have attached files, with thumbnails being created for image files. Several view modes are available, ranging from “1 line per Log Entry” compressed views to expanded and full views. These reports can belong to zero, one or several logical groups denominated Subsystems. Users can also reply to existing Log Entries, thus creating a thread.

To be able to insert new Log Entries, a user needs to have at least the WRITE privilege.

24.6.3.4 Announcements

Announcements are special Log Entries which should be used to broadcast short messages of general interest to the ALICE Collaboration. Although appearing as a normal Log Entry in the Log Entries section, an announcement is also displayed in the Big Screen View page and in the ALICE Live public website.

When inserting a new announcement, users must set a validity timestamp, thus defining until when should the messages be displayed.

As for normal Log Entries, a user needs to have at least the WRITE privilege to create new announcements.

24.6.3.5 Automatic Email Notification

The eLogbook allows an automatic email notification to be sent every time a new Log Entry is inserted. There are 2 possible configurations:

- Global: an email is sent for every inserted Log Entry.
- Per Subsystem: an email is sent for every inserted Log Entry belonging to a given Subsystem. The email address is defined in the ***email*** field of the ***logbook_subsystems*** table, where additional configuration parameters can also be defined.

24.6.3.6 Search Filters

One of the main goals of the eLogbook is to allow the members of the ALICE Collaboration to search for runs that match their criteria. To accomplish that, a filtering mechanism has been implemented in the Run Statistics section, allowing users to set a search filter for each available field. Filters corresponding to fields displayed on different tabs can be combined, although they can only be set or modified when in the corresponding tab.

Some filters have predefined values available (defined in the ***logbook_filters*** table), thus allowing easy access to common queries.

The search filters are also available in the Log Entries section, although they cannot be combined with the ones from the Run Statistics section.

24.6.3.7 Export Run Statistics

The eLogbook allows users to export the information displayed in the Run Statistics section in 3 different formats:

- TXT: text format, with 1 line corresponding to 1 run and values separated by semicolons.
- XML: XML format, with the ***root*** element depending on the exported tab and each `<RUN>` element corresponding to 1 run.
- EXCEL: spreadsheet format, with 1 row corresponding to 1 run.

Additionally, users can choose different export options, such as include / exclude headers (for easier parsing) or exporting only the run numbers.

LHC machine monitoring

25

This chapter describes the tool developed to read information about the beams delivered by the LHC machine and publishing them on the ALICE DIM (DISTRIBUTED INFORMATION MANAGEMENT) server to be stored into the electronic logbook at the start of each run. The LHC values are published by means of the DIP (DATA INTERCHANGE PROTOCOL) system, which allows relatively small amounts of soft real-time data to be exchanged between very loosely coupled heterogeneous systems. A ***Java*** application has been developed to perform an off-line cross-check between the values stored into the ALICE logbook by means of the DIP / DIM process and the ones stored into the ***LHC Logging Database***.

25.1 DATA INTERCHANGE PROTOCOL (DIP)

25.1.1 The DIP architecture

DIP is an information distribution service, and as such it may be profitably compared to subscribing to a newspaper or magazine (Figure 25.1).

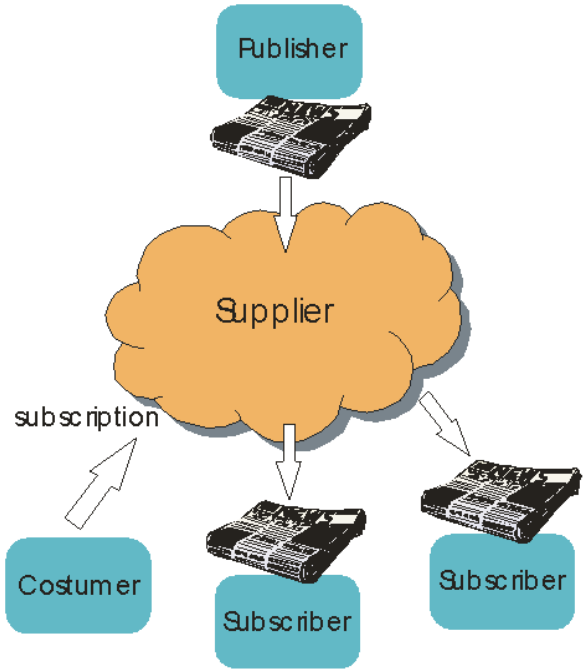

Figure 25.1 Information distribution service of a newspaper or magazine

If a person wants to receive a magazine on a regular basis, he may subscribe to it by giving its name to the supplier.
Whenever a new edition of that magazine is published, the person will receive a copy of it. If the person did not receive an edition when he was expecting one, he will contact the supplier who will give him one (ideally).
The person is of course not restricted to subscribing to one magazine only or just to magazines from a single publisher. The person needs only to provide the supplier the names of the publications he is interested in (he does not need to know who or where the publisher is) and he will receive new editions of the requested publication as they become available.
DIP is essentially playing the role of the Supplier in the above scenario. There is one notable difference between DIP and the magazine scenario: while a magazine is published on a periodic basis, this is not necessarily the case with DIP publications which may contain event based data which is updated as and when the event(s) occur.
Important components in the DIP architecture are publishers, subscribers and publications as shown in the diagram of Figure 25.2.

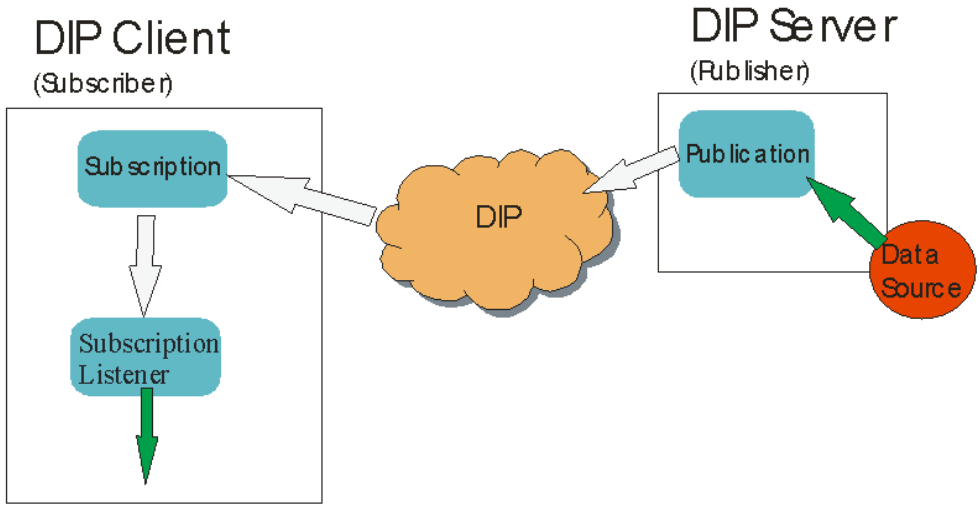

Figure 25.2 DIP architecture.

In the above diagram we see that DIP servers and clients act as the Publishers and Subscribers. The arrows show the flow of information between the producer (Data source) and Subscriber. The dark (green) arrows indicate that some action is required by the writer of the Publisher or Subscriber in order to make the information flowing. The light-coloured arrows are used where the flow of information between components is handled by DIP. A description of the components identified in the above diagram follows:

• ***Publisher:*** a producer of DIP data. A Publisher is responsible for the definition of the structure, the content and the provision of the DIP data to its Subscribers.
• **Subscriber**: a client of DIP data.
• **Data source**: this represents the source of the data that is to be sent via DIP. It may be internal or external to the DIP server. The DIP server is responsible for accessing the data source and writing it out to DIP through the Publication when it is needed (e.g. when the value obtained from the data source has changed).
• **Publication**: is a named object that represents an atomic piece of data published in DIP. The writer of the server must write the code that obtains the data from the Data Source and writes it into the publication object. A client subscribes to the publication by providing DIP with the publications name.
• ***DIP:*** provides the mechanism by which data is passed from the Publisher to the Subscriber (via the publication object), running on a DIP NAME SERVER (DNS). Moreover, the role of the DNS is to maintain the list of Publications available, and connect the Subscribers to the Publishers.
• **Subscription:** an object given to the client by DIP when that client subscribes to a publication. With this object a client may request the most recently published value of the publication or unsubscribe from a publication the client had previously subscribed to.
• **SubscriptionListener:** would be the equivalent of a magazine reader in our analogy. The SubscriptionListener acts as a wrapper, containing several callbacks. The most important of which handles data from those publications subscribed to when it arrives. The dark (green) arrow going out from the SubscriptionListener indicates that the implementer of the client must provide some code in the SubscriptionListener to do something with the data when it arrives at the Listener (i.e. display it on a console).

The DIP communication is based within LHC TECHNICAL NETWORK (TN), as shown in the Figure 25.3, which shows the DIP organization deployed in 2008, with a

central DNS in the TN and various Publishers and Subscribers communicating together across domains. This DNS is maintained by the IT/CO group.

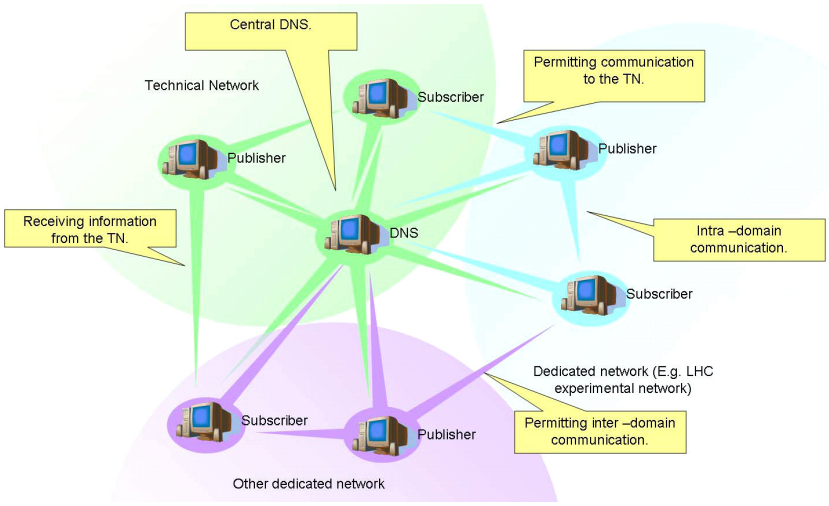

Figure 25.3 Cross–domain DIP communication on the TN.

There are some important characteristics about DIP one should be aware of:

• the Publisher and its Subscribers don't know each other explicitly. The DIP protocol, actually the DNS, connects the Subscribers with their Publishers of interest. A Subscriber knows at any time the status of its connections with the Publishers;
• a Publication is a structured container of atomic data. It is a consistent item and the Subscribers cannot subscribe only to a subset of its content;
• there is no filtering mechanism provided by DIP itself, i.e. once subscribed to a Publication, the Subscriber will receive all its updates;
• the Publisher has full control over data content, quality and timestamp;
• there is a notion of "data contract" between the Publishers and their Subscribers; a Publisher responsible for a set of Publications is not allowed to change online the structure of its Publications, once they have been made available for potential Subscribers;
• there are no particular security mechanisms implemented in the DIP protocol; for example, a Subscriber has no means to authenticate the source of the data it receives; similarly, the DIP protocol does not offer a Publisher the possibility to restrict access to a set of authenticated Subscribers; another example, the Publications' names are not nominative; in other words, when a Publisher stops, the Publications' names it was using are from then on freely available to other Publishers;
• there is no feedback to a Publisher that the data published actually reached its Subscribers (also known as "one-way" communication); hence there is no retransmission in case of a data delivery issue.

25.1.2 Setting up development environment

The distributions and a tutorial of DIP can be obtained from IT/CO's DIP support web page (***http://wikis.web.cern.ch/wikis/display/EN/DIP+and+DIM***). Two versions are available, DIP for ***Java*** and DIP for C++ (as zip files). Both these distributions run on the Windows XP/2000 and Linux platforms.

For C++ users:

- Under Linux
  - Be sure to modify your make file to include the following directories, ***<BASE>/dip/include/dip*** and ***<BASE>/dip/include/dim***, in your include search path.
  - Set the environment variable *LD_LIBRARY_PATH* to include the directory ***<BASE>/dip***

For Java users:

- Remember to include ***<BASE>\dip\dim.jar*** and ***<BASE>\dip\dip.jar*** in the class path when you compile and run your DIP applications. Additionally, ensure that the shared libraries (***libdim.so*** and ***libjdim.so*** for Linux) are reachable.

Before starting a DIP based application, be sure to set the environment variable *DIM_DNS_NODE* to **vodip01.cern.ch** and *DIM_DNS_PORT* to **2506**. This provides the application with the location of the name server that is required for DIP to run correctly.

25.1.2.1 DIP installation for C++ user under Linux

Download the released tar file for Linux 32 bit (***Linux tarball 32bits***) from the DIP Web site and save it in a temporary directory. Create the ***dip*** directory on your ***<BASE>*** account directory (eg. ***/home/dip_user***) and untar the file:

```
1:  <BASE> > mkdir dip
2:  <BASE> > cd dip
3:  <BASE>/dip > cp /tmp/DIP_tar_file.tar .
4:  <BASE>/dip > tar -xvf DIP_tar_file.tar
```

Create the bash script in Listing 25.1 (called ***dipEnv.sh***) in your ***<BASE>/dip*** directory, to set the environment variables need to develop, compile and run the DIP applications.

Listing 25.1 ***dipEnv.sh***: environmental setup

```
1:  #!/bin/bash
2:  export CLASSPATH=<BASE>/dip/linux/lib/dip.jar
3:  export
    LD_LIBRARY_PATH=<BASE>/dip/lib:/opt/dim/linux:$LD_LIBRARY_PATH
4:  export DIM_DNS_NODE=vodip01.cern.ch
5:  export DIM_DNS_PORT=2506
6:  export DIM_HOST_NODE=IPaddress_of_DIM_host_node
```

Check the connection with the DIP NAME SERVER (***vodip01.cern.ch***), which belongs to the TN. If it is not accessible, this means that your host does not belong to the TN. Request CERN-IT to include your host in the TN.

This check can be done launching, in a terminal, the ***Java*** DIP browser from your ***<BASE>*** directory with the bash script shown in Listing 25.2.

Listing 25.2 ***runBrowser.sh***: DIP browser launcher

```
1: #!/bin/bash
2: export CLASSPATH=<BASE>/dip/linux/lib/dip.jar
3: export
   LD_LIBRARY_PATH=<BASE>/dip/lib:/opt/dim/linux:$LD_LIBRARY_PATH
4: java -jar <BASE>/dip/linux/tools/dipBrowser.jar
```

If it does not start, check if Java real-time with a version greater than 1.4.2-16 is installed (***j2re.1.4.2-16***) in the directory ***/usr/java***.

25.2 LHC beam info: DIP client/DIM server

To retrieve the information on the beams a DIP client has been developed (***LHCClient.cpp***) that subscribes to the following publications:

- ***dip/acc/LHC/Beam/Energy***
- ***dip/acc/LHC/Beam/IntensityPerBunch/Beam1***
- ***dip/acc/LHC/Beam/IntensityPerBunch/Beam2***
- ***dip/acc/LHC/RunControl/BeamMode***
- ***dip/acc/LHC/RunControl/MachineMode***
- ***dip/acc/LHC/RunControl/FillNumber***
- ***dip/acc/LHC/RunControl/RunConfiguration***
- ***dip/acc/LHC/RunControl/CirculatingBunchConfig/Beam1***
- ***dip/acc/LHC/RunControl/CirculatingBunchConfig/Beam2***

This application has been developed in ***<BASE>/dip/linux/test***, in which you can find the makefile to compile the C++ code and produce the executable.

The information collected from the above DIP publication, after a processing step, is published on the ALICE DCS DIM server (***alidcsdimdns.cern.ch***), from which a logbook daemon takes it and stores it into the ALICE Logbook at start of each run. The DIM publications subscribed to by the logbook daemon are the following:

- ***ALICEDAQ_LHCBeamMode***: char ('C'), 'LHC Beam Mode (`STABLE BEAMS`, `INJECTION PROBE BEAM`, ...)'
- ***ALICEDAQ_LHCBeamType***: char ('C'), 'Type of collisions ('`p-p`','`Pb-Pb`')'
- ***ALICEDAQ_LHCBeamEnergy***: float ('F'), 'Energy of the beam in GeV'
- ***ALICEDAQ_LHCFillNumber***: int ('I'), 'LHC Fill Number'

- ***ALICEDAQ_LHCTotalInteractingBunches***: int ('I'), 'Number of Interacting Bunches'
- ***ALICEDAQ_LHCTotalNonInteractingBunchesBeam1***: int ('I'), 'Number of Non-Interacting Bunches in Beam 1'
- ***ALICEDAQ_LHCTotalNonInteractingBunchesBeam2***: int ('I'), 'Number of Non-Interacting Bunches in Beam 2'
- ***ALICEDAQ_LHCBetaStar***: char ('C'), 'LHC Beta* in meters'
- ***ALICEDAQ_LHCInstIntensityInteractingBeam1***: float ('F'), 'Instantaneous Intensity for Interacting Bunches in Beam 1 in num. of charged particles'
- ***ALICEDAQ_LHCInstIntensityInteractingBeam2***: float ('F'), 'Instantaneous Intensity for Interacting Bunches in Beam 2 in num. of charged particles'
- ***ALICEDAQ_LHCInstIntensityNonInteractingBeam1***: float ('F'), 'Instantaneous Intensity for Non-Interacting Bunches in Beam 1 in num. of charged particles'
- ***ALICEDAQ_LHCInstIntensityNonInteractingBeam2***: float ('F'), 'Instantaneous Intensity for Non-Interacting Bunches in Beam 2 in num. of charged particles'
- ***ALICEDAQ_LHCFillingSchemeName***: char ('C'), 'LHC Filling Scheme name ('`Single_12b_8_8_8`', '`150ns_152b_140_16_140_8bpi`',...)'

Other DIM publications are available for future applications:

- ***ALICEDAQ_LHCTotIntensityInteractingBeam1***: float ('F'), 'Total Intensity for Interacting Bunches in Beam 1 in num. of charged particles'
- ***ALICEDAQ_LHCTotIntensityInteractingBeam2***: float ('F'), 'Total Intensity for Interacting Bunches in Beam 2 in num. of charged particles'
- ***ALICEDAQ_LHCTotIntensityNonInteractingBeam1***: float ('F'), 'Total Intensity for Non-Interacting Bunches in Beam 1 in num. of charged particles'
- ***ALICEDAQ_LHCTotIntensityNonInteractingBeam2***: float ('F'), 'Total Intensity for Non-Interacting Bunches in Beam 2 in num. of charged particles'

All these values are sent to the infoBrowser every minute to be stored during the run.

The DIP client / DIM server process continuously runs and, if it stops for any reason, in less than one minute, is restarted automatically thanks to the execution of the bash script in Listing 25.3, (called ***restartLHCClient.sh*** and stored in ***<BASE>/dip*** directory) through the ***crontab*** service of Linux. The script executes every minute the check of the PID (PROCESS IDENTIFICATION) value of the process; if it is '0', it sets the environment and starts the process.

Listing 25.3 ***restartLHCClient.sh***: automatic check and restart of DIP client

```
 1: #!/bin/bash
 2: #daemon name:
 3: DIPDIMPROCESS="DIP_client_process_name"
 4: #pgrep command path:
 5: PGREP="/usr/bin/pgrep"
 6: #find the PID of daemon
 7: $PGREP ${DIPDIMPROCESS}
 8: #check if the daemon is active or not; if not, restart the daemon
 9: if [ $? -ne 0 ] then
10: # ================== Setup for DATE ========================
11: [ -d /dateSite ] && export DATE_SITE=/dateSite
12: if [ -f /date/setup.sh -a -d /dateSite ]; then
13: export DATE_ROOT=/date
14: . /date/setup.sh
15: fi
16: # ================== Setup for DIP/DIM ======================
17: export DIM_HOST_NODE=IPaddress_of_DIM_host_node
18:
19: . <BASE>/dip/dipEnv.sh
20: cd <BASE>/dip/linux/test
21: ./$DIPDIMPROCESS &
22: fi
```

25.3 LHC beam info: off-line cross-check

The non reliability of DIP publications has been proven so it is necessary to perform an off-line cross-check of values published by the DIP / DIM process and stored online by the logbook deamon. To do this, a Java application has been developed in order to perform an off-line cross-check between the values stored in the ALICE logbook (a ***MySQL*** database) and the ones stored in the ***LHC Logging Database***, the system developed to permanently store and manage the measured values of the most important parameters, configurations and working characteristics of the all accelerator parts (PS, SPS, LINAC, LHC, etc), and experiments.

The Java application uses the ***logging-data-extractor-client*** API, developed by the ***LHC Logging Database*** team. The application must be registered to have access to the data. To register a new application, the developing team of the ***LHC Logging Database*** has been contacted (***be-dep-co-dm@cern.ch***) and a meeting has been organized.

During this discussion both high-level views on the analysis / application objectives and also technical / implementation details have been addressed. As soon as the new application is registered and a sample method to access the data is sent, it is possible to start the development of the application to manage the data extracted by the ***LHC Logging Database***.

The following ***Java*** packages are needed to develop, compile and run the applications to extract the information from ***LHC Logging Database***: ***jdk1.6.0_20*** and ***jre1.6.0_20*** in ***/usr/java***, and, for the specific Java application developed for the off-line cross-check, also the ***Java MySQL connector*** package ***mysql-connector-java-5.1.13*** is needed, to retrieve and eventually update the information in the ALICE logbook.

By means of the ***Eclipse***'s software ('Eclipse IDE for Java Developers' package from ***http://www.eclipse.org/downloads/***) it is possible to create, modify and test the project of the application with the Java code and classes organized in the specified user workspace (***<BASE>/workspace/projectName***).

In ***<BASE>/workspace*** directory of the same host on which the DIP / DIM process runs, the Java project ***LHCLoggingProject*** has been developed to perform the daily off-line cross-check of the previous 24 hours of data taking. It needs to have the appropriate permissions to access and update the values in the ALICE logbook.

The Java application, named ***ALICEDataExtractionACR.java***, connects to and extracts the information from the ***LHC Logging Database*** stored in the last 24 hours of the day before, then it connects to and extracts the beam information stored in the ALICE logbook in the same time range, and for each run and for each variable it makes the comparison between the two values. If they are different, an update of ALICE logbook value is applied with the ***LHC Logging*** ones. Moreover the last value of each variable has been stored in specific files to know the starting point status for the next check.

Listing 25.4 ***ACRLHCLoggingCheck.sh***: automatic start of Java application for the off-line cross-check.

```
 1:  #!/bin/bash
 2:  export JAVA_HOME=/usr/java/jdk1.6.0_20
 3:  export PATH=$JAVA_HOME/bin:$PATH
 4:
 5:  JAVA_HOME_JRE=/usr/java/jre1.6.0_20
 6:  JAVA_PROJECT_DIR=/home/alicedaq/workspace/LHCLoggingProject
 7:  JAVA_LOG_DIR=/tmp/LogbookCrossCheck
 8:
 9:  cd $JAVA_PROJECT_DIR
10:  CLASSPATH=""
11:  for i in lib/*.jar; do
     CLASSPATH=${CLASSPATH}:/home/alicedaq/workspace/LHCLoggingProject
     /$i; done;
12:  export
     CLASSPATH=/home/alicedaq/workspace/LHCLoggingProject/build/bin:/u
     sr/java/jre1.6.0_20/lib/ext/mysql-connector-java-5.1.13-bin.jar${
     CLASSPATH}
13:  cp bin/* build/bin/
14:  cd src/java
15:  javac -Xlint:deprecation ALICEDataExtractionACR.java
16:  cp ALICEDataExtractionACR.class ../../bin/
17:  cp ALICEDataExtractionACR.class ../../build/bin/
18:
19:  DATETOCHECK=`date --date="now -1 day" +%F`
20:  echo ALICE cross-check for the date: $DATETOCHECK
21:
22:  JAVAEXEC="$JAVA_LOG_DIR/JavaExec/$DATETOCHECK.txt"
23:
24:  cd $JAVA_LOG_DIR/JavaExec/
25:  for i in $(ls -rt)
26:    do
27:    DATE=${i:0:10}
28:    echo "$i --> date: $DATE"
29:    LASTDATECHECKED=$DATE
30:  done
31:  MAKECHECK=1
32:  echo Last date checked: $LASTDATECHECKED
33:  if [ $DATETOCHECK = $LASTDATECHECKED ] && [ -s $JAVAEXEC ] ; then
34:      MAKECHECK=0;
35:  else
36:      NEXTDAY="$LASTDATECHECKED +1 day"
37:      NEXTDATETOCHECK=`date --date="$NEXTDAY" +%F`
38:  fi
39:
40:  echo Make check flag: $MAKECHECK
```

```
41:  cd $JAVA_PROJECT_DIR/src/java
42:
43:  if [ $MAKECHECK = 1 ] ; then
44:      echo "Make cross-check from $NEXTDATETOCHECK to $DATETOCHECK"
45:      ENDDATE=$DATETOCHECK
46:      STARTDATE=$LASTDATECHECKED
47:      DATE=$STARTDATE
48:      echo Start date: $NEXTDATETOCHECK
49:      echo End date: $ENDDATE
50:      DAY=1
51:
52:      while [  $DATE != $ENDDATE ]; do
53:          CURDATE="$STARTDATE +$DAY day"
54:          echo date command: $CURDATE
55:          DATETOCHECK=`date --date="$CURDATE" +%F`
56:          DATE=$DATETOCHECK
57:          echo The date to check is $DATETOCHECK
58:          let DAY=$DAY+1
59:
60:          JAVAEXEC="$JAVA_LOG_DIR/JavaExec/$DATETOCHECK.txt"
61:          echo Java executed file name: $JAVAEXEC
62:
63:          if [ ! -e "$JAVAEXEC" ] ; then
64:
65:              echo Make the check for day $DATETOCHECK
66:              cp /tmp/LogbookCrossCheck/lastHX:AMODE.txt
     /tmp/LogbookCrossCheck/NextLastBackup/nextlastHX:AMODE.txt
67:              cp /tmp/LogbookCrossCheck/lastHX:BMODE.txt
     /tmp/LogbookCrossCheck/NextLastBackup/nextlastHX:BMODE.txt
68:              cp /tmp/LogbookCrossCheck/lastHX:ENG.txt
     /tmp/LogbookCrossCheck/NextLastBackup/nextlastHX:ENG.txt
69:              cp /tmp/LogbookCrossCheck/lastHX:FILLN.txt
     /tmp/LogbookCrossCheck/NextLastBackup/nextlastHX:FILLN.txt
70:              cp
     /tmp/LogbookCrossCheck/lastLHC.BQM.B1:FILLED_BUCKETS.txt
     /tmp/LogbookCrossCheck/NextLastBackup/nextlastLHC.BQM.B1:FILLED_B
     UCKETS.txt
71:              cp
     /tmp/LogbookCrossCheck/lastLHC.BQM.B2:FILLED_BUCKETS.txt
     /tmp/LogbookCrossCheck/NextLastBackup/nextlastLHC.BQM.B2:FILLED_B
     UCKETS.txt
72:
73:              java ALICEDataExtractionACR $DATETOCHECK
74:
75:          else
76:
77:              if [ ! -s $JAVAEXEC ] ; then
78:
79:                  echo Make again the check
80:                  cp
     /tmp/LogbookCrossCheck/NextLastBackup/nextlast*.txt
     /tmp/LogbookCrossCheck/
81:                  java ALICEDataExtractionACR $DATETOCHECK
82:              fi
83:          fi
84:      done
85:  else
86:      echo "Date $LASTDATECHECKED already cross-checked!"
87:  fi
88:
89:  cd $JAVA_PROJECT_DIR
```

Part VII

The Transient Data Storage

November 2010

ALICE DAQ Project

TDS

26 The Transient Data Storage

This chapter describes the Transient Data Storage as it has been deployed at ALICE and its associated packages.

26.1 Introduction

ALICE data have to be recorded on the Permanent Data Storage (PDS). ALICE chose the CERN Adanced STORage manager (CASTOR) as support for the PDS. The decision was taken to implement in the ALICE counting room a Transient Data Storage (TDS) area where files would be stored at the output of the DAQ. The TDS would ensure low latencies and high reliability, enough to keep writing data whatever the status of the PDS is. A dedicated software package was developed to control the TDS: the Transient Data Storage Manager (TDSM). The target of the TDSM is to assign disks for writing to the GDCs, move the data files to the PDS and trigger their registration into AliEn. We will herein review the architecture and the features of both the TDS and the TDSM.

26.2 The Transient Data Storage architecture

The Transient Data Storage is organized in sets of hard disks, grouped in RAID6, realized as Disk Arrays (DAs). Several FibreChannel switches connect the DAs to the data writers (the GDCs) and the data readers (the TDSM Movers, hosts that handle the transfer of files from TDS to PDS).

Transfers in and out of the TDS disks work better when crossing fewer FibreChannel switches. For this reason, switches and hosts are organized in groups, a concept that helps optimizing the data traffic by keeping it (if possible) within the same group. In this model, a “group” is equivalent to one FibreChannel switch.

26.3 The TDSM

The TDS needs careful handling on order to:

1. Write data in ROOT format on the TDS with the smaller possible impact on the data acquisition procedure.
2. Synchronize the use of the disks belonging to the TDS to make them work at the best of their capabilities.
3. Efficiently migrate the data from the TDS to the PDS and register it in AliEn.

An example of the architecture of the TDSM is show in Figure 26.1. Here the data flows top to bottom, GDCs to TDS to CASTOR. Three group of disks have been highlighted within the TDS. Three volumes are used for writing (exclusive mode), while three other volumes are selected for reading in shared mode. Write volumes are critical as they must not slow down the DAQ: for this reason they are not shared. Read volumes, on the other hand, can experience latencies without effecting the migration procedure. It is mandatory that disks are not used for simultaneous write and read operations, which would make both operations extremely slow.

Outside the data flow we can see:

a. the TDSM Manager (left side): a node that coordinates all activities in the TDSM and keeps the liaison with the GDCs and with the CTP;

b. the TDSM configuration database which contains shared parameters, status variables, historic records and shared procedures;

c. the DAQ logbook, used to record the status of each file and to provide the necessary assistance to the TDSM operator;

d. the AliEn spooler, which runs on a dedicated node, talking to the DAQ network and to the CERN General Purpose Network (GPN), where the AliEn file catalogue gateway is hosted.

The small scheme on the top-right of the figure shows the state transition scheme that applies to the data disks.

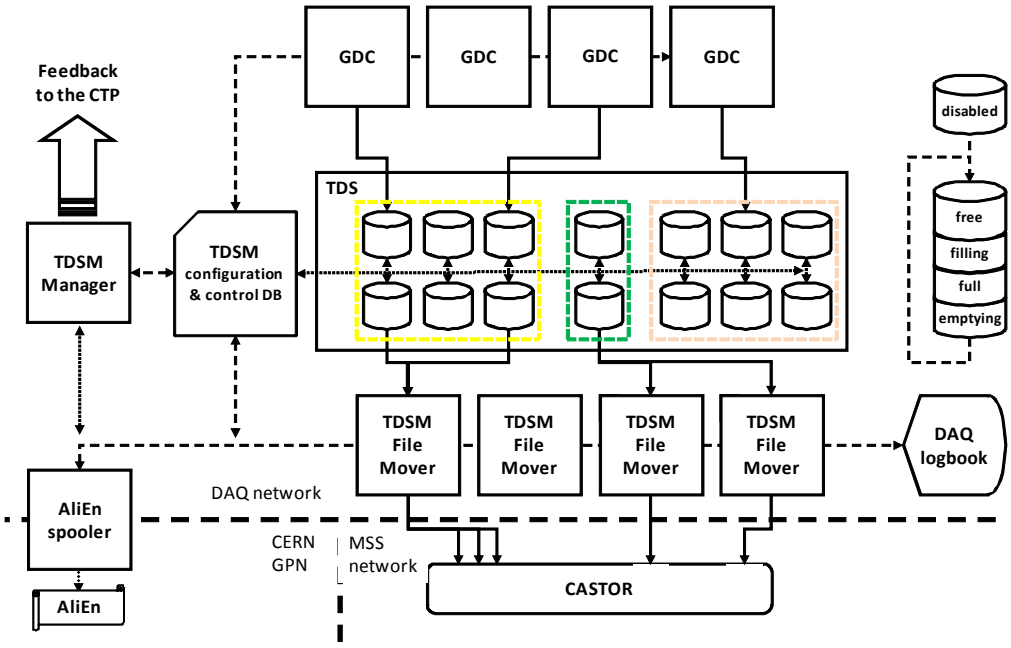

Figure 26.1 Architecture of the TDSM (example).

The central component of the TDSM is the configuration and control database. All the communications between actors are made through this database (dashed lines). Physically this database can run on any machine. It uses a separate set of tables in order to allow run independently from any other component (e.g. migrate data while the DAQ is under maintenance). The location of the TDSM database is defined in the DAQ/ECS configuration database.

The TDS and the TDSM can be monitored via dedicated TDSM statistics that reports the throughputs and timings grouped by several criterias (machines, groups, volumes). This allows spotting and solving problems such as faulty disk volumes or misbehaving TDSM File Movers.

The TDSM package provides a set of utilities for the operation of the TDS. These provide functions such as disabling a faulty volume, excluding an un-recoverable volume from the setup, off-loading a volume being rebuilt, disabling faulty File

Movers and so on. All these operations can be done at any time, regardless from the concurrent activities in ALICE, without stopping the TDSM and without changing the configuration database.

Other TDSM utilities allow more complex operations, such as re-organizing the hardware resources or assigning new nodes to the set. These operations usually require stopping and restarting the TDSM operation and therefore cannot be done when ALICE is in a running state.

In order to make good use of the hardware resources in ALICE, the TDSM monitors the occupancy of the TDS. Whenever this exceeds a pre-defined threshold, a dedicated script (named "***TDS Full***" script) is called. Actions taken by this script can be, for example, pausing the data acquisition process or changing the trigger profile in order to reduce the detectors data rates (e.g. allowing only rare trigger with lower throughput). A second script, named "***TDS empty***" script, is called whenever the TDS resources go below a second pre-define threshold, to revert whatever action was taken by the "***TDS Full***" script.

For detailed instructions on how to re-organize, re-configure, run and control the TDSM, refer to the ALICE DAQ WIKI.

26.3.1 The TDSM and the DAQ

The TDSM handles the attribution of resources to the DAQ as follows. At start of run, the ***mStreamRecorder*** process checks if the write volume declared for this particular run is available. If not, it requests a volume to the TDSM and waits for the directory to be assigned. When the TDSM completes the transaction, it creates a symbolic link which is detected by ***mStreamRecorder*** and used to create the output ROOT files.

During the run, the ***mStreamRecorder*** process checks at regular intervals the presence of the write directory. Whenever the write volume changes state (e.g. when it gets full), the TDSM removes the symbolic link and creates a new one pointing to a newly assigned volume. The ***mStreamRecorder*** process detects this change, closes the current output file and creates a new one, ensuring the transition to the new volume.

After a pre-defined period of inactivity, the TDSM changes the status of the write volumes and triggers their migration. This avoids stalling volumes for too long at the end of a run.

The operator can, at any time, trigger the migration of the data files written by a partition to the PDS. This is translated into a command for the TDSM that takes care of changing the status of the write volumes. Please note that the recostruction of a given run cannot be started as long as the run is not completed: therefore triggering the migration of a run before the run is closed will not speed up the reconstruction of the run itself (it will only make the data files available earlier to the reconstruction process).

26.3.2 Size of the output files

The size of the output files written on TDS is a very important factor. The PDS requests to handle big files, in order to optimize tape handling and packing. For

this reason it is better not to request the migration of a run to PDS as long as the run is not over. Smaller files increase the latency for their migration to PDS and their recall from tape, while making tape handling less efficient and slower. At ALICE, the size of the ROOT data files is specified in the configuration of ***`mStreamRecorder`***. Refer to the ALICE DAQ WIKI for more details on this subject.

26.3.3 Links within the TDS and TDSM components

At Point 2, all the TDS and TDSM components are inside the ALICE DAQ network (to allow communication between hosts and databases). However, two channels need to be open outside this network:

1. Data links to the PDS.
2. Communication links to AliEn.

Data links are allowed via firewalling commands on the Ethernet switches between the TDS and the PDS. Communications with AliEn are done using hosts equipped with two NICs, one connected to the DAQ network and one connected to whatever network hosts the AliEn server (currently on GPN).

26.3.4 The AliEn spooler

There are two separate implementations for the AliEn spooler:

1. one or more DAQ-hosted machines running dedicated Offline software for the registration;
2. one or more Offline-hosted machines handling the registration.

In the first architecture, the DAQ takes care of checking the health of the registration process, of the AliEn spooler daemon and of the forward progress of the procedure.

In the second architecture, currently in use in ALICE, the DAQ provides the needed information to the gateway. This information is then handled by the Offline software (which takes care of error handling, error recovery and forward progress checking). Hot-swap between multiple nodes is performed in case of failures: the operator is notified of the event, but no immediate corrective action is required as long as there is at least one active gateway available.

Part VIII

November 2010

References

References

27

1. K. Aamodt et al. (ALICE Collaboration), J Instrum., ***3***, S08002 (2008).
2. ALICE Collaboration, ALICE Technical Design Report on Trigger, Data Acquisition, High-Level Trigger, Control system, CERN / LHCC / 2003-062.
3. C. Gaspar, A distributed Information Management System for the DELPHI experiment at CERN, in ***Proc. of the IEEE Real Time Conference***, Vancouver, Canada, 1993.
4. J. Barlow ***et al.***, Run Control in Model: the State Manager, in ***Proc. of the 6th Conf. on Real Time Computer Applications in Nuclear, Particle and Plasma Physics***, Williamsburg, VA, USA, 1989.
5. B. Franek and C. Gaspar, ***SMI++, An object oriented framework for designing distributed control systems***, IEEE Trans. Nucl. Sci. 45 (1998) 1946-1950.
6. http: / / dev.mysql.com / doc / mysql.
7. http: / / httpd.apache.org.
8. Apache Software Foundation, Hypertext PreProcessor, http: / / www.php.net.
9. J.P. Baud ***et al.***, CASTOR status and evolution, ***Proc. Conf. on Computing in High Energy Physics***, La Jolla, CA, USA, 24-28 March 2003 (SLAC, Stanford).
10. R. Brun and F. Rademakers, ROOT An object oriented data analysis framework, Nucl. Instr. Meth. A389 (1997) 81-86.
11. ALICE Collaboration, ALICE Technical Design Report on Trigger, Data Acquisition, High-Level Trigger, Control system, CERN / LHCC / 2003-062, 41-108.
12. B. G. Taylor, LHC Machine Timing Distribution for the Experiments, in ***Proc. of the 6th Workshop on Electronics for LHC Experiments***, Cracow, Poland, 2000 (CERN 2000-010, Geneva, 2003).
13. Danny Cohen, ***On Holy Wars and a Plea for Peace***, IEEE Computer, Oct. 1981, 48-54.
14. ALICE Collaboration, ALICE Technical Proposal, CERN / LHCC / 1995-71.
15. R. Divià ***et al,*** Data Format over the ALICE DDL, Internal Note ALICE-INT-2002-010 V5.1.
16. ALICE Collaboration, ALICE Technical Design Report on Computing, CERN-LHCC-2005-018, 15-21.
17. ALICE Collaboration, ALICE Technical Design Report on Computing,

CERN-LHCC-2005-018, 27-30.

18. T. Oetiker, Round Robin Database tool, http://www.rrdtool.com, 2003.
19. http://cern.ch/alice-daq.
20. M. Boccioli, F. Carena and O. Pinazza, ALICE DCS Run Control Tool, DCS Internal Note, 2010.

List of Figures

List of Listings

List of Tables

List of Acronyms

A

ACR	ALICE Control Room
AliEn	Alice Environment
AliMDC	A class to represent raw data as a ROOT object (originally designed for alimdc program)
AliROOT	ALICE Off-line framework for simulation, reconstruction and analysis based on ROOT
AMORE	Automatic MOnitoring Environment
Apache	HTTP server

B

BC	Bunch Crossing
BE	Big-Endian

C

CASTOR	CERN Advanced STORage Manager
CDH	Common Data Header
COLE	Configurable LDC Emulator
CTP	Central Trigger Processor
ClT	Calibration Trigger flag

D

D-RORC	DAQ Read-Out Receiver Card
DAQ	Data Acquisition System
DATEMON	DATE system monitoring set
DATE	Data Acquisition and Test Environment
DB	Database
DCA	Detector Control Agent
DCS	Detector Control System
DDG	DDL Data Generator program
DDL	Detector Data Link
DIM	Distributed Information Manager package

DIU	Destination Interface Unit in RORC
DMA	Direct Memory Access
DST	Detector Software Trigger event type
DTSTW	Data Transmission Status Word in RORC

E

ECS	Experiment Control System
EDM	Event Distribution Manager
EOB	End of Burst
EOR	End of Run
EPS	Encapsulated Postscript file format

F

FEE	Front-End Electronics
FEIC	Front-End Emulator Interface Card
FERO	Front-End Read-Out
FIFO	First In First Out buffer type
FSM	Finite State Machine
FeC2	Front-end Control and Configuration program

G

GB	GigaByte
GDC	Global Data Collector
GUI	Graphical User Interface

H

HLT	High-Level Trigger
HTTP	Hypertext Transfer Protocol

I

IPC	Inter-Process Communication

K

KB	KiloByte

L

LDC	Local Data Concentrator
LSB	Least Significant Bit
LTU	Local Trigger Unit
L2a	Level-2 accept (trigger)

M

MB	MegaByte
MOOD	Monitor Of Online Data and Detector Debugger
MSB	Most Significant Bit
MSR	Multiple-Stream Recorder
ms	millisecond

N

ns	nanosecond

P

PCA	Partition Control Agent
PDS	Permanent Data Storage
PNG	Portable Network Graphics file format
pRORC	32-bit/33 MHz PCI bus RORC

R

RCS	Run-Control Server interface
ROI	Region Of Interest
ROOT	An object-oriented data analysis framework
RORC	Read-Out Receiver Card

S

SBC	Single Board Computer
SIU	Source Interface Unit (in DDL)
SMI	State Management Interface
SOB	Start of Burst
SOR	Start of Run
s	second

T

TPA	Trigger Partition Agent
TPC	Time Projection Chamber
TRG	Trigger
TTC	Timing, Trigger and Control system

Index

Symbols

A

B

D

E

F

I

L

M

P

R

S

T