ALEPH PUB 2001-003

9 June 2001

Experience with the ALEPH Online system

Olivier Callot

Laboratoire de l'Accélérateur Linéaire, Orsay

Marco Cattaneo, Markus Frank, John Harvey,
Beat Jost, Pere Mato, Wolfgang Tejessy

CERN, Geneva

TABLE OF CONTENTS

TABLE OF FIGURES

1 - INTRODUCTION

1.1 - The context of the Aleph experiment

ALEPH was one of the four experiments at LEP. This section describes shortly the context for the reader unfamiliar with LEP and the Aleph experiment. The detector is described in reference [1].

The LEP machine usually operated with four bunches of particles, with 22.5 μs between crossings. The machine configuration changed to 8 bunches in the so called pretzel scheme for a few years, then bunch trains were invented, where each bunch was a succession of up to 4 bunches with 320 ns separation. In the last years, this bunch train mode was used when taking Z0 calibration data, while the simple 4-bunch mode was used at high energy. The machine energy was constant in a fill up to 2000, when mini-ramp allowed changing the energy during a fill. The fill duration also decreased with time, due to the increasing energy. Twelve hours was usual in 1994, the last year at the Z0, while in 2000 the duration was more typically 2 hours.

Aleph was a classical collider detector, with several concentric layers of detectors, plus two end-caps and low angle luminosity monitors. A two level trigger system, one able to decide before the next crossing and the second one requiring 100 μs, produced an accepted rate between 4 and 10 Hz. A third level was implemented in software. The tape writing rate was around 5 Hz, with an average event size of 40 Kbytes. Hadronic events have an average size of 120 Kbytes, the maximum acceptable event size was 1.8 Mbytes.

The readout was performed by about 60 Fastbus-based processors, which were eventually replaced by VME processors having a Fastbus interface. Event building, which involved assembly of the event fragments produced by the various readout processors, was performed in up to 3 steps using ~ 10 processors. The complete event was finally assembled in the Main Event Builder, and was then transferred by optical link to the host computer. One very important system concept was **partitioning**, which implied that one can read only part of the system, and even have concurrent readout activities as long as they don't interfere. The trigger distribution supports this paradigm, with a programmable Fan-In-Out (FIO) tree, enabling the system to be configured such that each processor can be connected to the main trigger system or to a local trigger. The readout was also 'partition-aware'. A mechanism for managing resources i.e. booking and sharing had to be implemented in order for the partitioning concept to work in practice. Heavy use was made of databases to allow for the dynamic re-configuration of the readout system. Every DAQ task inherited some context information from its creator, such as the partition it belongs to, and accessesd configuration databases to know what information to use. Removing a detector from a readout activity just involved editing the database, no recompilation of software was required. Adding a new detector implied also creating the proper configuration tasks and database entries.

1.2 - The ALEPH Online system

Work on the ALEPH online system began in ~1984 when a small team of people started to assemble and work on the project. Already at that time the overall project was divided into a number of subprojects, each assumed to be largely independent of the other and often under the responsibility of different teams working largely independently. The main sub projects were the ALEPH Event Builder (Marchioro), the Mac64 system for detector controls (Maugain), the General Surveillance System (GSS), and the DAQ and control software (ALEPH software team). As the size of the group increased other projects were started, such as the Level 3 trigger and Fastbus/Vax interface (in 1986), and the FIO for trigger distribution.

By 1984 some of the main technology choices had also been made, e.g. Fastbus for the data acquisition bus and VAX/VMS for the online computers. Software was to be written in FORTRAN and full exploitation of VMS system services was taken for granted; one of the first packages to be written was UPI and this relied heavily on the VMS/SMG library. Very little consideration was given to use of third party commercial software, ORACLE and Multinet being a notable exception. In 1986, the decision was taken to use OS9 and the C programming language for programming the embedded processors. At the time commitment to these technologies was absolute and little consideration was given to the need for changing to new technologies at some future date. In retrospect, this approach might have had quite costly consequences.

A great deal of effort was put into making a functional specification of the entire system. This defined the architecture, the main functional components and the dataflow protocols, as well as the mechanisms envisaged for detecting and handling errors. Some effort went into simulating the expected performance, although simulation tools were in their infancy at that time. In addition, a great deal of emphasis was placed

on system issues and some of these, in particular partitioning had especially far reaching consequences for the design of the whole system. The process for deriving this specification provoked a lot of discussion, some conflicting ideas and took a great deal of time and effort to complete. However the end-result was a document (‘the bible’) that was constantly referred to and that was also recognised for its worth by people from other experiments.

ALEPH started to take data in 1989 and finally stopped in November 2000. During this long period deficiencies in the original design were identified and corrected. Several software packages, such as UPI, were rewritten from scratch a number of times. In 1992, the G64 processors were replaced by commercial units (VM20s) running OS9 and in 1994, the Fastbus processors were replaced by commercial VME processors. The kernel of the Slow Control software was rewritten in C++[2]. Perhaps the most enduring feature of the online system was the functional specification and, despite the many changes to individual items, the overall architecture of the system in 2000 remains faithful to the one described in the original specification.

After completion of the experiment, it was recognised that a lot of operational experience had been gained during the 10 years or more of running ALEPH. The ALEPH online team decided to try to capture this experience on paper so that lessons could be learned from the mistakes that had been made such that they would not be consciously repeated in the LHC era. This is the purpose of this document.

2 - NETWORK

2.1 - Configuration

The Aleph online networking infrastructure was in the end based on an FDDI backbone. The main server machines and a few workstations were directly connected to FDDI, whereas most of the other equipment was connected to 10 Mb/s Ethernet, mostly through 10base2 ("cheapernet"). The bridging between FDDI and Ethernet was done through an FDDI-Ethernet bridge and subsequent fanning-out of the Ethernet signals to several cheapernet segments via a port switch. The entire network infrastructure was housed in 3 DECHub 900 chassis. Figure 1 shows the physical layout of the networking infrastructure in the Aleph online environment.

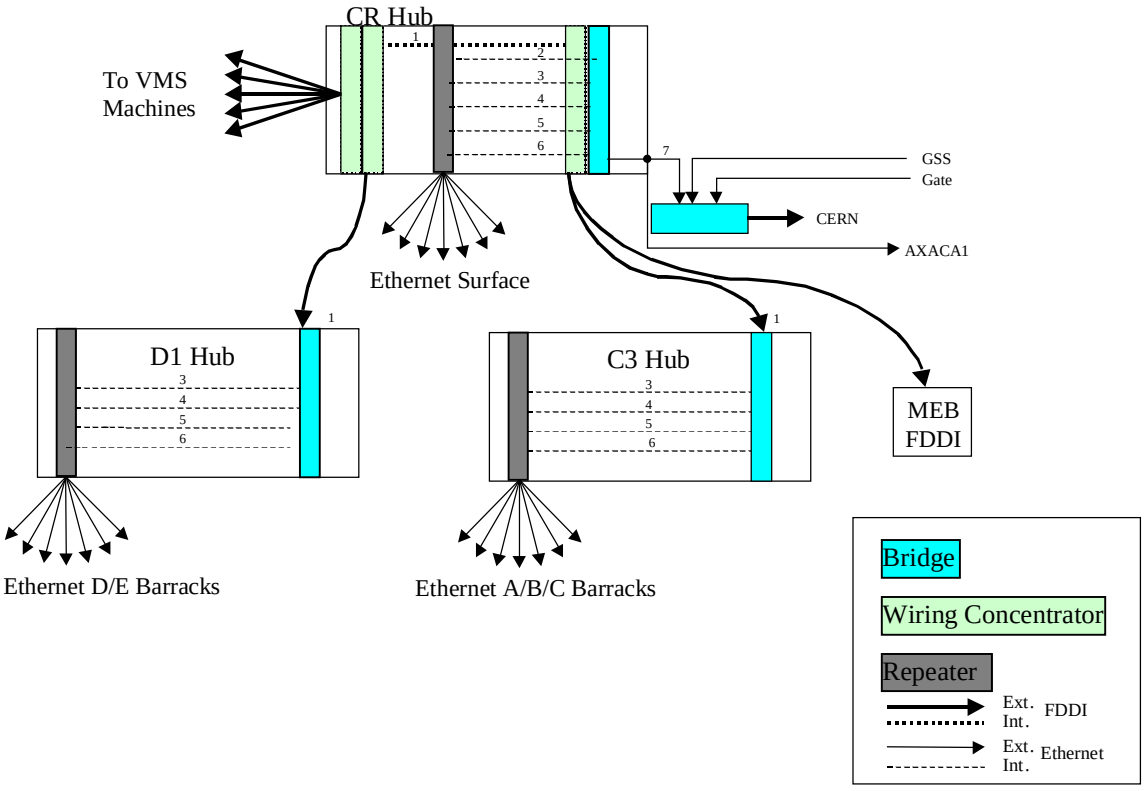


Figure 1: Physical view of the network configuration in the Aleph online environment.

The networking infrastructure as depicted in Figure 1 was the result of several upgrades, which followed to some extent the development of the networking technology. At the outset Aleph's networking topology was very simple. Basically the entire online network was one logical shared 10 Mb Ethernet segment implemented physically with very simple repeaters. Any network problem anywhere on this segment immediately brought down the entire network. To aggravate things, the equipment didn't allow any diagnostic, so that whenever there was a problem people from the networking groups of CERN had to be called with network analysers if the problem could not be isolated by disconnecting equipment.

In a first upgrade, around 1991 or 1992, the simple repeaters were replaced by more intelligent ones (Pirelli) that allowed for monitoring of the activity on a per-port basis and also to isolate certain ports via the front-panel. Still the topology was that of a single shared segment, which of course implied a serious performance penalty.

It was only in 1995, in the course of the upgrade of the computing infrastructure from VAX processors to Alpha-based machines, that the final topology with FDDI and separated Ethernet segments was installed. At first only the infrastructure on the surface (Computer room) was replaced and a year later also the infrastructure in the cavern was upgraded.

2.2 - Monitoring

Network monitoring and configuration is done through a commercial software package, "Clearvision", which allows configuring all components in the DEChub architecture remotely. There is a general problem with monitoring and trouble-shooting computer networks in that it is the most cost effective to use the network itself for control and monitoring. This approach works very well as long as there is no problem with the network, i.e. as long as the monitoring equipment (PC) can access the main network components, like the HUBs and the components in there. As soon as there is a general network problem that prevents general accessibility all this control and monitoring breaks down and manual intervention in form of disconnecting links etc. is necessary in order to isolate the problems. At least twice in the lifetime of Aleph serious network problems happened, related to a complete de-programming of the HUBs after violent power cuts.

Ideally there should be a separate very simple network (out-of-band network) that would allow monitoring and manage the vital pieces of the networking infrastructure, provided of course these pieces of equipment allow for out-of-band monitoring. In the future, an effort should be made to control and monitor critical pieces of equipment with an out-of-band network, to ensure reliable operation and efficient diagnostics and recovery.

2.3 - Performance

The performance of the network has never been measured systematically. However file transfers in excess of 800 kB/s have been measured using VMSCluster protocol. This doesn't sound exciting but no effort was invested to optimise the performance since we did not suffer from performance limitations for all practical purposes.[1]

[1] Part of this modest performance can be attributed to the fact that the maximum transfer unit (MTU) of the FDDI ring had to be set to the maximum Ethernet frame size of 1500 bytes, because of a limitation on the VME-to-FDDI interface.

3 - SOFTWARE COMPONENTS

3.1 - Operating Systems

The software of the Aleph online system is based on commercial operating systems, OS9 for the embedded processors and OpenVMS for the central online computers. An important feature of these operating systems that has been used extensively is the fact that they are both interrupt-driven, i.e. signals on OS9 and ASTs (Asynchronous System Traps) on VMS[2]. ASTs and signals help solve the problems of many parallel IO requests being pending and waiting for any of them to complete without the necessity of polling.

In the Aleph online environment there was a conscious decision taken early in the project to not hesitate to use special features of the chosen operating systems (VAX-VMS and OS9). While this approach eased the implementation of the system and increased its functionality, it clearly bears the risk to depend on the set of operating systems chosen. In retrospect the choices were such that there was not really a problem, in that Digital kept developing VMS for the lifetime of the experiment (and beyond) and also OS9 is still very much alive with different implementations for various CPU architectures. Aleph has never suffered from its decision to not attempt to write portable code.

It is evident that the situation would have been different if the system had to be used in many different environments and experiments, where the choice of hardware and software is not necessarily in the hands of the developers of the system. In these cases clearly portability is a serious issue. The Aleph system, however, was never intended to be used by a wider community.

3.2 - Third-Party Software

The Aleph online system relied basically on the Operating systems chosen and on software written within the Aleph collaboration. There was, however, a small fraction of the used software that was written outside the direct control of the online team. This incorporated software from other groups at CERN (Cernlib) but also software from commercial companies. Examples of the latter include "Multinet", an implementation of TCP/IP on VMS, and Oracle, the RDBMS. Usually these products worked without major problems. Difficulties only arose when an upgrade of VMS was necessary, for one reason or another. In these cases it had to be ensured that the appropriate version of Multinet and Oracle were available as well, otherwise the system stopped working.

In the future these issues will become more prominent and their solution will determine whether it will be possible to operate the systems efficiently. It is clearly not acceptable to wait for 6 or 12 or more months before an upgrade of the OS can be envisaged, just because the third-party products are upgraded. In the course of the evaluation of candidate products, it should be assured that the providing company has an agreement with the OS manufacturer to obtain field-test versions of new releases long beforehand, such that all products are available within the shortest time interval.

3.3 - Software organisation

The software is organised in packages, where each package implements a well defined functionality and exhibits a well-defined callable interface. Packages are only accessed through their interface and never should internals of a package be accessed by other packages or applications. Packages are made available to other packages and applications through the mechanism of shareable images (under VMS) or through trap libraries (under OS9). This mechanism was fundamental to the maintainability and extensibility of the online system, since it allowed packages to be maintained (bug-fixes) or extended without other packages or applications having to be re-compiled or re-linked. This was true as long as the interface didn't change. Under VMS, there exists also the concept of version matching (GSMatch option). This allows control of whether software linked with older versions of a shareable image can still run with newer versions. The use of shareable images and the fact that (under VMS) it is possible to define which shareable image is activated by defining a logical name also allowed new versions of packages to be tested with a subset of applications.

[2] The Thread concept did not exist or was not implemented at the time most of the Aleph Online software was developed. Nowadays most likely much more use would be made of threads.

The sources, object codes and executables of the individual packages were organised in subdirectories of a package's root directory. There were three areas defined for each package ("Development", "Production" and "old"). The production area was the default area. It was supposed to contain the latest officially released version of the package. Developments or bug fixes were made in the development area. Once the development was finished and tested a tool copied the files that were updated from the development area to the production area, after having saved the current production area's files to the 'old' area. The inverse tool, i.e. to copy the 'old' area to the production are also existed, which allowed to recover the last working version, in case the development version turned out to have problems, even after testing. This procedure was only very rarely necessary, because the shareable image approach allowed very extensive testing (even at large scale) before the software was moved to the production area.

With around one hundred packages, software management rules were established, enforced and used for the whole lifetime of the experiment. First the directories structure, with (at least) source, binaries and debug directories. Then the requirement to have release notes before any release. Last, an automatic procedure to save the current production area, move the source files from development to production area, a full rebuild from source, ensuring the software consistency, and last a notification of the release. This procedures were also used to manage sub-detector specific software. Having the same directory structure and naming allowed easy access to other people's code, a feature appreciated when debugging problems in a running experiment. The software manager was responsible for moving packages into the production area. He (or better his tools) also ensured that release notes were written and submitted at the time of the new release, even though there was no check on their contents. The software manager also ensured that disk space was used reasonably, inviting people to purge files if space became low.

3.4 - Fundamental Packages

The basic packages of the Aleph Online system are

- Wait Utility to formalise waiting for one or many asynchronous inputs and taking proper action upon arrival.
- Message passing system (AMS) for process-to-process communication
- Buffer management for accessing and sharing physics data
- Database management system(s) for configuration storage and retrieval and for book-keeping
- Scheduling system for activating tasks
- Resource management system for controlling access to non-shareable resources
- User interface package (UPI) for controlling applications
- Finite-State-Machine (FSM) package to synchronise activities of distributed programs and processes

3.5 - Global sections

Global sections are a VMS concept to share data between separate processes allowing the different processes to access the data by simple memory accesses. VMS only allows writeable global sections to be accessed by processes running on the same machine. A package extending this to processes running anywhere within the same cluster has been written, making extensive use of the VMS distributed lock manager. Initially these global sections were extensively used to implement configuration and other databases. In 1995 most of the databases were re-implemented by means of a home-made object-oriented database management system (see also section 3.9). This step reduced the use of global section considerably. Henceforward global sections were principally only used to publish status information to interested pieces of software.

3.6 - Waiting for Events

With Event handling we mean the treatment of several concurrent external stimuli to a program. Examples of these are controls (Run control) messages and physics data arrival interrupts. The problem is that a wait has to be performed on an OR of several stimuli the number of which is not a-priori determined. Upon each stimulus, a specific action, intimately related to the nature of the stimulus, has to be executed. There are many ways to solve this common problem, such as

- Leave all the management to the 'main' program
- Implement the stimulus handling with threads
- Implement a formalised way of waiting for stimuli allowing to implement independent and concurrent 'threads' of stimulus generation/handling, which can be independently developed

Clearly the first possibility is the least attractive, since it will put a lot of burden to the application programmer and does not allow independent developments. Hence the choice rested between the last two possibilities, out of which we chose the latter. The reasons for this were basically the absence of thread support in the OS9 and (initially) in the VMS operating system. There is another advantage (in hindsight) of not using threads, in that the use of threads imposes a lot of restrictions on the user code and the underlying packages he uses, namely that all code has to be thread-safe. To guarantee this is not at all obvious, since normally the user might not know what packages he indirectly uses. Aleph was somehow lucky to do what it has done. However, of course, the implementation chosen does not allow real concurrency. If this is wanted/needed the thread approach has to be implemented with all the caution necessary.

3.7 - Communication

All communications between processes in the system (process-to-process communication) is based on the Aleph Message System (AMS). AMS is based on the wait utility and the TCP/IP implementation of Multinet. It allows sending arbitrary sized messages from one process to another. Processes are identified by the node (processor) they are running on and by their name. On VMS the process name has significance in the system, whereas on OS9 processes do not have names. On OS9 there was an infrastructure built in order to allow processes being identified by a meaningful name. Since there is no notion of process name also in TCP/IP another infrastructure had to be built to support the use of names in connecting from one process to another (AMS Name server) without the knowledge of the TCP/IP port number of the target process.

The communication system does not implement "keep-alive message" to determine whether all the systems are still alive or have rebooted. Instead, "I-AM-DEAD messages" are sent to all connections from the image-rundown routine when a process dies. When a processor reboots, a feature of TCP/IP consists in not informing the all previously connected partners of this reboot. An error is reported only when data are sent to an "orphan" connection. This receive error is trapped and an "I-AM-DEAD" message is generated internally.

Probably future implementations should try to use "keep-alive" messages and automatically try to reconnect to dead partners. Threading the software would definitely help.

The fact that TCP/IP was implemented in a third-party product (Multinet) caused some problems when trying to upgrade VMS to a new version. Because the TCP implementation is, by nature, quite intricately related to the operating system we always had to make sure that a new version of Multinet was available for the new version of VMS before upgrading. This usually caused delays in the upgrade timing. Evidently these problems will multiply themselves when more third-party products are used. Upgrades of the operating system will be delayed for a very long time until the last provider of third-party software has his version available.

3.8 - Task scheduling

Task scheduling is different for processes scheduled under VMS from those to be run under OS9. Under VMS the VMS batch system is very extensively used. The reasons for this are many:

- Each batch job can have a log-file associated with it, which allows the software to print directly into the log file. Up to 10 version of log files were kept on disk such that also events reaching back in time could be investigated. There was however no tool available that would inspect all, or a set of, the log files in a given time window in order to correlate events in different tasks and to perform extensive trouble shooting.
- Each batch job executes from a command (script) file. This allows to define specific (private) environments for each batch job
- The VMS batch system allows a batch job to run several command files. This is used to setup a common environment for all batch jobs. A system (cluster) wide logical name was defined to specify the default preamble file. Up to 9 preamble files were in principle allowed. By defining appropriate logical names also in the private preambles, the special environments could be propagated down an

entire scheduling tree (from parent to children and children of children, etc). This proved very powerful for testing new software versions, where all processes had to run the same version for compatibility reasons (e.g. new AMS versions).

Under OS9 tasks were created via a server process that was specifically written and conveyed a certain environment to the newly created process. The environment there was however more rudimentary than under VMS. The logging of all tasks running under OS9 was centralised in a utility called ARPEL (Aleph Readout Processor Error Logger). This utility connected to a task on each OS9 processor that in turn collected all the logging data written to a special device (/tube). At the start-up of each task the standard output and error devices were reassigned to the /tube device.

The scheduling also kept track of whether a certain process with a given name already existed. In case the process was defined as single copy and already existed when an attempt was made to create it, an error was flagged, otherwise a new version number was appended to the process name. Under VMS the uniqueness of the process names was assured cluster-wide, where as under OS9 it was only processor-wide.

3.9 - Configuration databases

The management and configuration software relies heavily on the use of databases to hold descriptions and parameters of the detector, the readout, trigger, control and data monitoring systems. It is important that the contents of these databases are described in a complete and consistent way and without redundancy. Using these databases made our DAQ system completely data-driven. This gave us a good deal of flexibility since changes to the configuration or running conditions do not require any program to be re-compiled or re-linked, it only demands a few changes in the contents of some of the databases. The fact that the parameters are stored in a database ensures that all the programs use the same set of parameters, thus the coherence is guaranteed across the system. Finally, data-driven programs can be re-used in different environments and running conditions, thus reducing the software maintenance load.

We ended up having about 40 databases, each one specialised to a given domain of the on-line system. Examples of these databases included the Fastbus database, containing descriptions of the front-end electronics, the VME database, which describes the readout configuration, the trigger and slow control databases, as well as databases describing software components, such as histograms.

Based on our experience of running and maintaining the system for the first 5 years, we did identify some problems and limitations. To alleviate these problems, we undertook a gradual migration to a home-made database called "DBTool". Some of the problems that motivated this are as follows :

- Many domain-specific databases make the system more modular so in some way more maintainable. However it complicates the implementation of the relationships that exists between domains. The way we overcame this complication was by duplicating small portions of information or by using implicit links based often on strong naming conventions. This duplication complicated the maintenance of the integrity of the data.
- Since different people collaborated in the development, different application program interfaces (API) appeared and editors with different look-and-feel. Since similar code had been produced many times similar bugs appeared and were replicated many times.
- The physical implementation rendered difficult the evolution of the system since changes in the database schema were not always easy.

DBTool was an attempt to overcome these problems. With this tool we could implement new databases quite rapidly. Many databases where converted from the original implementation to use the tool. Examples were the Fastbus and VME databases, Slow Control database, Histogram database, ReadOut database etc.

DBTool supports database external links by the mechanism of importing remote classes. Using this feature we can build the global schema maintaining the idea of separated database domains. External and internal links are viewed the same way through the application program interface. Thus, for the developer it is very easy to navigate the entire database.

DBTool stores the database schema as metadata in the database itself, thus allowing the possibility of creating data-driven generic tools. In particular, a generic editor was developed that was used to fill and update all databases. This was the way we provided a uniform user interface for the people operating the DAQ system.

DBTool provided program interfaces to Fortran/C/C++ and was running on VMS and OS9. The performance was improved by using disk caches and hashing algorithms for data retrieval and was comparable to the performance achieved using the global section implementation.

Since DBTool was not a full-blown DBMS, it missed some quite essential features:

- History of changes and rollback. Recording the changes done in the database (for diagnosing purposes) and the possibility to undo some of them was clearly missed on some occasions.
- Backup system. We had to build a backup system that regularly was saving the contents of all "known" healthy databases. This did allow us to recover from sporadic data corruption or inconsistencies on several occasions during the years of running.
- The binding to C++ was not really there. The application programming interface was basically C like.

3.10 - Resource Management and Partitioning

Resource management means synchronising access to resources. These resources can be non-shareable, in which case the resource management will act like a mutex, i.e. it will allow or restrict access to the resource, or it can be a resource with only a limited number of users. In this case the resource manager will act like a counted semaphore and block access once the maximum number of 'users' is exceeded. In Aleph the resources were identified through their name as ASCII strings. The names were composed with a naming convention that allowed implementation of a hierarchy e.g. booking 'TPC' reserved all resources belonging to the TPC detector, whereas booking TPC_SIDEA_EB just booked the Event Builder belonging to TPC Side A and so on.

Resource management and partitioning are quite closely related since resource management is a prerequisite for partitioning. Partitioning means the possibility to divide the DAQ system (hardware and software) into several independent data flows. This concept has far-reaching consequences both in the hardware architecture and in the software. For example the distribution of clocks and triggers has to follow the partitioning scheme in that the trigger decisions originating in one part of the readout should only go to the modules that are in the same partition and not to others. For the software, partitioning means that programs or software components must not make assumptions on the number of sub-components they are dealing with. For example the run-control program must know which microprocessors take part in the current configuration and must only configure those and ignore the others.

Originally the partition definition in Aleph followed to some extent the detector description in that a partition was described in terms of detectors, components and sub-components. This led to some confusion from time to time on which parts of the detector are currently read out. Later (1996) the partition definition was revised and a partition was subsequently defined by which non-shareable entities (Readout Controllers) were included in the readout. From this information and the configuration database all other readout components could be deduced and appropriately included. This helped significantly the understanding of the configuration that was in use.

3.11 - Error Logger

This is a centralised system, to which all tasks anywhere in the system can send formatted messages. The original purpose was to log these messages on a disk file, and to display the pending messages on one or several screens. New functions have been added, to access help information for the displayed errors, and to dispatch selected messages to interested clients.

According to our experience, this is an essential element for the operation of a big experiment. The following requirements should be kept in mind:

- Guaranteed delivery. When a task reports an error, the delivery should be guaranteed. This means that the error logger should acknowledge reception of the message, with a short latency so that a timeout can be implemented. The error logger should then have a fast interface for handling incoming messages, even if it puts them in a queue for later processing. We have in fact implemented two tasks, one that acknowledges immediately, and forwards messages to a second task without latency constraints.
- Severity levels. Only the sender can easily qualify if the message is informative, severe or fatal. This should be transported with the message. This severity was used to display the messages on dedicated screens, the ALARM screen information being handled with highest priority. We also had informative messages, which were not displayed on screens, but just logged so that experts could keep track of problems that could not be acted upon by the shift crew.
- Protection from message 'flooding'. It may occur that a task sends many errors. For example, a monitoring task detects and reports a fault for every event, several times a second. We have implemented a protection in the sending routine, such that the same message cannot be sent more than

once a minute. The list of recent messages is kept locally; any new request is compared to the recent messages and ignored if identical to a previous one. In addition, the error logger itself will ignore a message if it is identical to an existing one. This avoids filling the log file for the same message every second.

- Multiple messages. The problem is to handle catastrophes. An example is a power failure, where every crate trips. The simplest approach, where each fault generates an error report, will clearly fill the Error Logger screen, and may hide other important faults. We implemented a "conditional report" where an error can specify that it should not be displayed if an existing error has set a specified condition. This allows for example a message "power cut in a barrack" to hide the individual fault messages of each element in the barrack. Another example, the Fastbus Crate monitor can count how many crates are faulty and reports "Many faults" to hide all the individual problems, allowing other information to be displayed on the screen, which may explain the global problem. Unfortunately, this has been introduced quite late in the lifetime of the experiment, and is then not largely used. Nevertheless, we are convinced that a proper handling of disasters is very important, and difficult to test!

- Forwarding of messages. This is an essential feature. In fact, this is how the display screens are implemented, and is the mechanism by which tasks can react to specific messages. A task can register for a family of messages, and each existing or new message will be forwarded to it. The message hiding of the previous point is used only by the display task, all messages are forwarded, which allows a Fastbus Crate recovery task to handle all crates even if there were "many faults". The way to specify which messages the task is interested in should be reasonably smart, to avoid sending uninteresting messages, and to avoid implementing a smart filtering in the receiving task. In particular, the forwarding of the 'end-of-problem' messages should be automatically implemented.

- Help. A fundamental requirement for the operator. From the error screen, a single keystroke allows a specific help text to be displayed on the screen, giving instructions on how to handle the specific problem. This replaces big manuals to be searched in. Rules should be defined to allow several messages, corresponding to the same fault on different objects, to share the same help. One difficulty is to ensure that every possible error has an associated help file. The way we implemented it was to add a button to the Error Screen so that the shift crew can report that this specific error had no help, or only a dummy help.

- Hidden information. The error message contains a code, and a text to be displayed, readable by an operator. It may also contain computer readable information. We used this feature for reporting an anomaly in a histogram. The text of the message describes the anomaly, while the computer readable information describes the file containing the histogram and its ID, so that the Histogram Presenter can display it automatically. The same mechanism could be used to pass device identification in case of a slow control problem, allowing an expert system to identify easily the faulty component.

- Clearing alarms: It is usually simple for a task to detect that an alarm should be sent. It is less clear when to clear the alarm. This is when the faulty condition was there, and has disappeared for some time so that it will not come back in the next event or measurement. Some conditions cannot be cleared easily, so one had the possibility for the operator to clear an alarm. As a consequence, some real alarms were cleared by mistake and thus ignored, while they were real and serious. One solution was to re-send the alarm regularly, so that even if removed by the operator the error message will reappear. A related problem is that alarms may be "normal" if we are not in standard conditions, for example during shutdown. A mechanism to condition errors to a global running mode of the experiment is very useful.

- Remote display. As for all our screens, it was possible to start from any terminal in the world an error logger display. This allows the expert to see really the problems, from home or from his office.

3.12 - Log files

The main tools used to record the behaviour of the system were log files. As we run all tasks as batch jobs, each task had a log file. We also used dedicated output files for several tasks, usually to create a new file every day, so that the file size stays manageable. The points to emphasise are:

- Time stamp: Each entry must be time stamped else it is impossible to correlate one event in a task with another event in another task. The time stamp must be accurate at least to the second. Of course, the clock of the various processors in the system should be in phase also!

- Split files: A log file can become big, and it is impractical to handle huge files. For most of the dedicated files, we open a new file every day, using the VMS version number to distinguish them. Clearly, a name including the date would be needed on another operating system. But it is important in

this case to keep the time ordering, i.e. putting "ddmmyy" in the file name is bad, "yyyymmdd" is better as the next file in the directory will be the one of the next day.

- Keep the file for a long time. We use VMS version limits for normal log files, where only 10 versions are kept. This prevents proliferation of log files, as we have typically 200 tasks running in the system. However, for important files, a memory of several months (or even years) was found to be useful.

The location of these log files should be easy to remember. In Aleph, all normal log files were in the same area, but the dedicated ones, usually the most important ones, are each in its own area, not always easy to guess. A better standardisation would have clearly helped.

One problem frequently encountered with these log files is the difficulty to retrace the history of an event which affected several tasks, because we had no tool to build a 'merged' log file. The only way was to have several windows and follow the time sequence by jumping from window to window. Clearly a standard time stamp with synchronous clock would help to build a 'merger' to follow an event as seen by several tasks. Note that PBLOG, FBFIX and DEXPERT (see section 6.2) were logging in the their own log file information they received from several other tasks, thus merging the information in a readable way.

We also developed tools to extract information for logging databases, such as for the error logger, the Run and Fill databases and the like.

3.13 - User interface UPI

The Aleph user interface UPI was designed to be used with simple alpha-numeric terminals (VT220 Compatible). This choice was made in the early phase of the Aleph Online system (~1986), basically because of the absence of other technologies (Workstations, PCs, X-terminals, etc). While there is no doubt that the aesthetic appearance of the Aleph user interface was not terribly exciting (see Figure 3), there was a big advantage in the years of running Aleph, in that the menus and commands of the tasks in the system could be accessed via dumb terminals from any location, also and especially from home via a telephone modem. This allowed solving problems in the system extremely efficiently, since the on-call expert did not have to come to the control room. Care should be taken in future systems that this feature is preserved. Most likely, though, the standard in these systems will not be based on a line-oriented system.

UPI is basically the interface of a task to the terminal attached to it. But the most frequent use was the 'Server' version, where a single terminal was connected to many tasks. The main console had more than 30 tasks connected, with their menus. It was nice to have the same interface to the task in stand-alone (test) mode and in normal running, with the Server. However, we suffered from a basic design choice: The user code defines the lay-out of the menu by direct UPI calls, and receives the commands selected by the operator as menu-number, command-number. In order to control the task by another task, like an expert system, one has to implement another command decoding. It would be better to have a single interface, receiving a command string, which could be used from a menu interface or from a message interface. It would also allow implementing various types of menus, like a normal menu for standard operation and an expert menu to access features and parameters of the task that a normal user should not see.

Another feature of the user interface is the fact that many "operators" can act on the tasks in the system. This, while it is very useful for problem solving during running the experiment, of course has to be exercised with care, since there is no sequencing or locking done for the commands that are sent by the different operators. In fact, this is not really wanted, since the idea is to fix problems inside the system or task. Hence, it cannot be guaranteed that the sequence of commands would be what was expected at the time the software was written.

4 - CONTROLS

4.1 - Run Control and FSM

Data taking normally proceeds in cycles. Firstly, there is a setup phase when the detector is made operational and the readout system configured. This is followed by a data-taking period and finally there is a closedown phase, which may also involve a change in the operational state of the detector. The detector is partitioned into logical components, which could be subdetectors. Each subdetector may be partitioned into even smaller units. The choice of which components participate in data taking depends on the current partition and is a parameter, which also defines the tasks participating in the context of the current run.

The tasks involved when taking data from the equipment or subsets of it are controlled by the DAQ control program, normally referenced as the "Run Controller". The run controller is responsible for the synchronisation of the various steps necessary to prepare the hardware and all processes, which participate in taking data. The hardware and the readout is done by dedicated detector specific software under the control of the run controller which is responsible to ensure that non-sharable resources are not used simultaneously in concurrent active runs. Examples for such resources are FIOs or simply run numbers which must be unique.

In developing a control architecture for ALEPH it was found convenient to model the entire system in terms of a Petri-Net in which the behaviour of each component, i.e. task, is represented by a finite state machine (FSM). This consists of specifying in which states each component can be found and the possible transitions between these states. Each transition is characterised by a "condition" which causes it to be invoked and an "action" which should be performed on making the transition. An important feature of the Petri-Net is that each transition can be made dependent on other system components being in a particular state. Thus, by using these dependencies, a tree-like control hierarchy is constructed. At the top level the operator – either human or an external network message - issues commands to the run controller. The command is then forwarded to a number of subsidiary tasks controlling the setup or the readout of a given sub-detector. The commands are then forwarded in turn down the "tree" until they reach tasks implementing the data acquisition functions. In order to minimise the number of communication channels between different processing elements, it is convenient to have a dedicated control task running in each readout processor. The role of this task is to receive all commands from its parent controller and to forward the commands to any DAQ tasks running on its local processor.

The run control task initiates a transition to a new data taking state, but is the last task to actually complete the transition. New commands will not be accepted. This ensures that actions can be performed on all parts of the system in the required sequence. If an action cannot be performed successfully by a particular dependent task, the whole sub-tree including this task and the run controller will fail to make the transition and an intervention is required to fix the problem. A clear description which task failed the transition and possibly the reason for the failure helps to recover.

If on receiving a message a task finds it is already in the required state then it replies immediately without performing any actions. This ensures that actions are performed only when necessary and helps to keep setup times to an absolute minimum. A recovery procedure is provided for those occasions when some part of the system fails to respond. This procedure allows the operator to ignore errors, to cancel transitions or to reset the entire system to a defined state.

This control model also allows a reversal of the flow of control. This feature is used, for example, when a task finds a fatal error it cannot handle. It initiates a transition to put itself into the "Error" state and a dependency rule is invoked to put its controller into the same "Error" state. This dependency rule is defined for each control task such that the error flows up the sub-tree to the run controller. On making the transition to the Error state, the Main controller will disable the trigger and wait for an operator action to resume or abort data taking.

Thus, in addition to the readout tasks, there are also tasks to initialise, test and calibrate the various components. The system used to control data taking must be capable of sequencing these actions to ensure that every component has been set into its correct state at each step in the cycle. In large systems with many tasks, which control different detector elements, this can be a formidable problem.

Any number of finite state machines can coexist in each task, each responsible for dedicated actions like the setup of the hardware and the subdetector specific readout processors, the trigger, run changes or calibration runs. The machines are defined through a sequence of transitions between their different states. The definition of these transitions, the states and the rules for executing the transitions are defined in a database,

which is read at start-up. This initially was intended to allow flexibility. In reality the protocols never changed. Different machines and controlled tasks are grouped in baskets, where any dependent task can belong to exactly one basket at a time. This allows e.g. to move tasks from one FSM basket to another and back and temporarily act only on a subset of the tasks. This for example is used for error recovery.

The machines embedded in a task are independent. They can act in parallel, meaning that a transition in one machine can be completed independent of the state of another machine embedded in the same task. This feature was not present in the first implementation of FSM where any task could only implement one finite state machine. At that time for example DAQ errors during run changes could not be handled, the system was stuck and had to be “Reset” and reconfigured.

The state of each task and its embedded FSMs is published to the system in a global section (memory module on OS9) and is available for monitoring purposes and state displays. A generic display program was very useful for debugging purposes to capture a snapshot in case of failures in the readout processors.

Also useful was the possibility to stack transition sequences for known procedures like calibration runs, where the transition sequence is well known.

A description of the FSM implementation and of the protocols used within the ALEPH DAQ system can be found in Appendix A. In the following a short description of the user interfaces and existing monitoring tools will be given.

4.1.1 - FSM User Interface

All tasks incorporating an FSM have a standard user interface (see Figure 2). This interface shows the name of the embedded FSM machines, the current state of each machine and offers a list of possible transitions. This list allows the user to manually proceed to any target state reachable from the current state. However, nearly all tasks implementing this interface menu are slave tasks of the run controller. These tasks were manipulated by the run controller rather than the operator. Consequently, the menus were used rarely. However, in case of a failed transition the presence of a separate menu is an advantage.

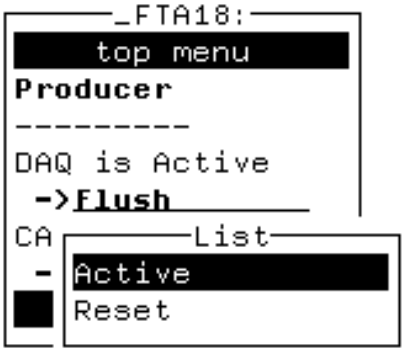


Figure 2: The standard user interface implemented by a task with two embedded finite state machines. The list shows the possible states reachable from the current state of the specified machine.

4.1.2 - User Interfaces of the Run Controller

The run control task steers all other tasks participating in the data taking process. To facilitate this, the run controller steers the machines of dependent tasks with rules that depend on the states of its FSM machines.

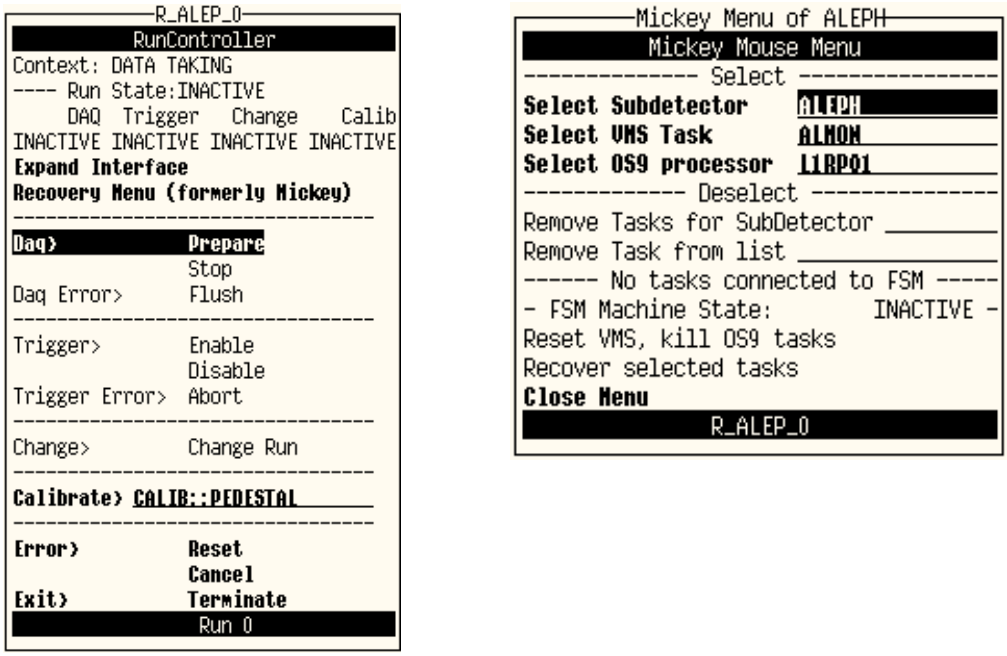


Figure 3: (a) The run controller user interface. (b) The error recovery interface.

Figure 3 (a) shows the four finite state machines of the run controller to steer the setup, the trigger, run changes and calibration activities. On top, the overall status of the run and the status of the individual FSMs is visible. If no calibration activity is selected the calibration machine will be deleted and disappears.

"Run State" indicates the state of the DAQ machine during the initialisation of the run. If FSM is not yet invoked, this status line indicates the current action: "Setting up", "Starting". Otherwise, an artificial super-state such as "Running" or "Error" is indicated.

The state is updated according to the state of the embedded machines. In addition, the shown action items correspond to the main actions performed on the embedded finite state machines. Action items are enabled or disabled in the FSM transition callbacks according to the state of the corresponding FSM machines. Effectively the menu items correspond in the same way to the possible transitions as in the simple task interface.

The "Reset" command is special. It triggers the transition of all embedded FSMs in the run controller and its dependent tasks from any state into the initial state, where no DAQ activity is present. This transition must not fail.

Figure 3 (b) shows the recovery menu used to handle data taking exceptions, dead tasks, etc. This menu allows individual tasks to be removed from the context of the run control machines. It also allows them to be manipulated e.g. to recreate these tasks and pass them back to the control of the main FSMs. The FSM machines of this recovery tool are the same as those of the main run controller, hence a subset of tasks can manually be manipulated in the same way as with the run controller itself. After the task manipulation has ended, the manipulated tasks are automatically passed back to the supervision of the main run controller's FSMs.

Very often not only one single task fails a transition, but rather a whole group of tasks. This happens typically after hardware interventions, when sub-detectors were left by the sub-detector experts in a state which was not compatible with the configuration of the run. Therefore the selection of entire sub-detectors was made available and all tasks corresponding to the readout of this sub-detector could be manipulated together. This possibility to recover or manipulate a configurable list of tasks synchronously was very useful. It was for example necessary in order to reconfigure both the VMS setup task and an OS9 readout processor in case of a readout processor crash.

The operator is at any time able to assess the state of the data taking either by using the summary information as displayed on top of Figure 3 (a), or by involving a display from the run-controller menu, which shows the full view of all depending tasks (see Section 4.1.4). This feature was very useful during the remote recovery of data taking exceptions, when not all status displays are visible, as it is the case in the control room.

These interfaces were designed for the shift crew. It is not intended to offer full functionality, but to be safe and intuitive. This intention also propagates to the recovery menu. It is important that the menu available to the shift crew is simple and intuitive. Otherwise, it is impossible to steer recovery actions orally by talking to the shift leader. In particular when the experiment becomes mature, the shift leader's knowledge about manual recovery procedures vanishes because more and more tasks are executed in automatic mode. These interfaces also must be simple and "handy", because error conditions during data taking translate immediately into lost luminosity. The look and feel of these interfaces can only be designed together with the shift crews, who supply important feedback.

However, during commissioning phases it is important that the full functionality is accessible. For this reason, a separate interface to the run control was implemented. Although this interface is freely accessible, it is usually only invoked by experts.

Usage of the calibration button depends on the calibration type. Although in the beginning only one "ALEPH calibration" was supported, it turned out that more and more sub-detectors need different configurations for calibrations. The calibration type can be selected from the menu. According to the choice, the selected parameter space, given by a table in the readout database, may vary and different calibration configurations can be executed.

The other important task the run controller must handle is the setup of the run configuration parameters. This must be done even before any task starts executing FSM transitions. Later, resources must be allocated according to these parameters. The configuration parameters include the event limit, maximum event size, resource allocation flags, readout processor control flags, activity/run type, data output destination and the definition of the data input source used. The specification of the event input source allows to either read events from the detector or to simulate data monitoring using disk files. These parameters must be accessible and editable if the run is in the Inactive state. Since these parameters also affect the readout configuration, it is important to be able to inspect the resulting task configuration.

4.1.3 - Programming Interface

In the beginning there was the operator. Nevertheless, with time the operator lost much of his knowledge. This experience is made by all long living experiments with a constant flux of members throughout the collaboration. For this reason known tasks become increasingly automated when the experiment becomes mature. For example, error recovery procedures are well known to experts rather than shift crews responsible for taking data. Instead of calling the corresponding expert, it is much more efficient to automate known error recovery procedures. Automatic recovery is supported through a network interface capable of listening to formatted network messages from expert tools. The interface implements a handshaking protocol which allows the execution of transitions, the recovery of tasks etc, i.e. all actions a human operator would invoke.

4.1.4 - The Run Control Display

The run control task has a dedicated display, which allows to quickly capture visually the state of the run controller and its view of the data acquisition system. The requirements for this display are conflicting: it should be both intuitively simple and complete. To satisfy both, the display only shows tasks, which are not in agreement with the state of the run controller or, during transitions, the state the task aims at.

4.1.5 - The Good, the Bad and the Ugly

The implementation described above generally worked satisfactorily and very reliably.

Problems occurred only with the programmable interface when several tasks simultaneously tried to manipulate the run controller. In the design phase, there was only one expert system present, which was responsible for handling data acquisition errors only. However, with increasing automation, several tasks simultaneously tried to manipulate the run controller, sometimes requesting conflicting commands. This situation is comparable with several human operators connected to the task each manipulating different actions. The interface was designed to handle one external intervention at a time. Clearly, if several new requests arrive while executing a previous request, some synchronisation between the participants is needed. In addition, the synchronisation cannot be steered by the run controller itself, unless its intended functionality would be greatly increased. Unfortunately, these situations do not occur as rarely as one might hope. For example if the beam is lost, some readout processors might crash; this results in a DAQ error and

DAQ error handling starts and at the same time the end of run calibrations should start. See section 6.3 for a detailed discussion of the multi master problem.

Sudden deaths of the run control task, although rare, are not handled in a satisfactory way. In this case, the data taking continues happily. Although the run controller listens to deaths of its slaves, nobody is listening to its death. Once dead, the restart is not straightforward. After reconnecting to the existing tasks, a *Reset* command must be issued and a new run started. Typically, this results in lost luminosity.

The start-run actions necessary to set up the detector for data taking and the corresponding stop-run actions are performed through several transitions (see Appendix A.1 for a detailed description of the different transitions). Dividing the start-run and stop-run actions using several steps must be done with care, since the integrated execution time is determined by the slowest task executing each transition. In order to optimise the time all tasks require, it is necessary to group into the same transition all the time consuming actions such as downloading data to the readout processors, reading sub-detector specific conditions or allocating resources. Of course, this requires communication between the software developers of each subdetector.

It is essential that in case of failures both experts and the shift crew are able to quickly recognise the current state of the DAQ system from the display. For this reason, it is mandatory that the names of the transitions, which are executed and the names of the individual states are meaningful. To make the shift leaders aware of the individual steps executed during start-run and stop-run sequences and to recognise the current state of the DAQ system, a poster in the control room showed the different transitions of the implemented protocols.

4.2 - Safety

4.2.1 - GSS

The highest level of safety alarms (level 3) which ensure personal safety is handled by the General Surveillance System, GSS. The main actions consist in transmitting alarms to the fire brigade and/or switching off main power circuit breakers to avoid major incidents, and in subsequently informing the experiments of the problems encountered. It also includes some elements of detector safety. GSS was designed as a common solution for the four LEP experiments which had the well-known often cited advantages of Common Projects for efficient sharing of resources. Since the approach of each LEP experiment to the rest of Slow Controls was not common, GSS does not fit into the ALEPH controls architecture, which resulted in having a stand-alone VAX in the Aleph Control Room with a user interface having a look and feel that was completely different from what the operators were used to from other screens. It also made data communication with the ALEPH computers tedious. This was an obvious disadvantage.

4.2.2 - Safety Crates

At the next lower level, a network of 8 Safety Crates (not to be confused with the "ALEPH slow controls") is installed that monitors and controls safety conditions in the electronics barracks in the cavern. This is a solution that is not common with other LEP experiments. Parameters include fire detection in barracks, rack temperatures, water faults, etc. Switch-off actions are taken locally when required. This network of crates – which uses the inexpensive UTINET as local area network - is based on standard MAC64 monitoring and acquisition crates with the Flex operating system on 6809 processors with the safety programs stored in EPROMs for reliability. It was in general reliable, but there was no flexibility! To be clear, the experts were unable to rebuild the code from existing source files when we discussed the possibility to upgrade this system after 5 years of operation! The data exchange of this network with the ALEPH Control Room (ACR) is described further down.

An additional set of four gas safety crates form the kernel of the gas safety chain which directly takes the necessary actions in case of a gas anomaly. These crates contain mainly relay-based logic and are interconnected to produce proper safety action on gas detection alarms, etc. Vital signals are transmitted to GSS. Information of general interest is passed on to the nearest general Safety Crate for further routing to the ACR.

4.2.3 - Gateways

Originally, gateway units based on a standard MAC-64 chassis were used in ALEPH to interface each Utinet segment via Ethernet to the VAX cluster. These were based on the VIP VME processor card (Motorola 68010) and included the Utinet head-end units. One such unit interfaced the unique Utinet safety-crate segment to the Online Cluster. This was later replaced for standardisation reasons with a Utinet-to-Ethernet interface implemented in a dedicated FIC. The gateways for the Slow Controls crates [see below] became obsolete when each controls crate was directly interfaced to Ethernet.

4.2.4 - Bubble Counter

Add-ons to this basic safety system included an intricate bubble counter. For the hadron calorimeter some 700 bubblers must be checked every few hours to check that gas is still flowing normally; the manual viewing was replaced by an automated system based on light reflection variations through the bubbles as they pass through a column of oil; this worked very nicely indeed. The software was based in a Data General PC running the AT & T flavour of UNIX and had been mainly written by visiting scientists and upgraded by several other persons who had since also left CERN. The PC type was used nowhere else at CERN other than once in the ALEPH magnet control system. A thunderstorm in 1998 that blew out the special Ethernet control card by Western Digital reminded us of the problems that non-standard solutions will cause for buying replacements in an emergency. Since the vintage PC was completely non-standard and the operating system outdated and no longer maintained, this setup gave us disproportionate worries during the last interval of running. Luckily it actually did not break down, else we would have had to scrap this automation altogether and revert to regular manual inspections.

4.2.5 - Xenon Gas Controls

An add-on that was hidden to most users was a set of MAC64 controls crates interconnected by UTINET and steered from the Online Cluster with the help of an expert system using Prolog. This was used remotely by the Marseille group to run and manipulate the Xenon/CO2 closed-loop ECAL gas facility from Marseille. Since this constituted a separate system with only a minor interface to the ALEPH General User, and since the experts who developed the system remained attached to ALEPH for the duration of the experiment, maintenance posed no problems.

4.2.6 - Communication with the Online Cluster

The readout by the Online Cluster of the safety-crate system is essentially doing nothing, as we have (in principle) very few safety alarms. The only way to test that the system would be able to transport a message is to send test messages at regular intervals. These are 'bounce' messages (a request is sent and an answer should come in a reasonable time interval) and 'heartbeat' messages, where each processor runs a timer and produces a message every few minutes and spontaneously sends it to the host task. Absence of a heartbeat message indicates that the processor would not be able to send an error report (while still maintaining all safety functions locally through the EPROMs).

The second function of this software is to convert the reported error to a standard Error Logger message. The shift crew is informed this way of the fault and should act before the fault becomes an alarm, in which case the safety processor may take brutal actions like cutting the power. Each Error Logger entry should have its own HELP entry. Having a proper, clear and detailed HELP for each possible fault is a mandatory feature!

Apart from this data retrieval, there is the possibility of sending commands. This was used to implement the sending of RESET pulses from special outputs of the safety crates to the individual Slow Controls crates, in order to reboot these remotely in case they got stuck. This feature was of quite some importance.

4.3 - Slow Controls

4.3.1 - Hardware

The equipment controls, more commonly called "ALEPH slow control system", comprises some 30 MAC-64 crates which monitor and control many of the parameters of the detector and the electronics. The G64-bus-based crates contain standardised interface cards to handle the various I/O needs to the devices and sensors. While difficult to debug, these cards proved to be very robust once installed and running. Thus exchanges of such cards were extremely rare during the last years of running ALEPH.

The same was not the case for the processors and the local area network. Originally, Motorola 6809 processors were used as in the Safety Crates, with UTINET interconnections and VIP based Gateways to the main computers. This proved to be under-dimensioned and in the winter shutdown of 1992/93 the processor parts were replaced by VME-cassettes plugging into the G64 bus containing a VM20 processor card (68020) and a VLAN card as Ethernet connection. For the software, this was a good solution, although a tenacious bug in the OS9 operating system did cause the units to hang every few weeks for a long time until it was finally found and a fix installed. The hardware of the cassettes, by contrast, remained somewhat of a nightmare until the end of LEP. The back-plane connections were prone to bad contacts, exchange of cards in the cassettes often resulted in non-responding units, quite a few processor cards died suddenly and were hard to repair. It is now surmised that the cards may be more sensitive to electrostatics during handling than we expected.

While the I/O cards proved to be very stable, this cannot be said for many of the sensors. To cite an example: for the TPC about a hundred PT100 probes were installed with, for economy, a special 3-wire instead of the usual 4-wire connection, this being based on the fact that the resistance of two equally long cables with an equal number of plug connections in-between cancel out for the voltage measurement (application of Kirchhoff's law). In the real world, after several bendings and un- and re-pluggings they no longer do, so unfortunately the overall environment measurements remained approximate and variable. It is therefore suggested that sufficient resources in future be put into the primary elements of the measurement chain to reduce surprises. While this helps, it is not a guarantee of success. After the end of LEP we re-calibrated three high-cost temperature/humidity measuring units used to survey the cavern and found that while the temperatures were correct to $1/10^{th}$ of a degree, the humidity values of two of them were off by 15% relative to the third one which was used as a reference since it had an incorporated visual display.

More elaborate interfaces also got incorporated into the controls system of necessity. Thus, a nuclear magnetic resonance device, a digital multimeter and a special high voltage display came with the very flexible GPIB bus interface, for which a connection to a slow controls crate was incorporated. While this bus has very many excellent features for a test bench setup in the lab, the programming for standard monitoring is heavy and the programming effort should not be underestimated when planning to incorporate such items in future designs. A further anachronistic bus connection was a CAMAC branch driven from a Fastbus interface; although needed it did mean another non-standard element.

The magnet controls were partly developed in Saclay, and partly outsourced to a company, Gardy, that specialised in controls. The Saclay magnet supervision runs via an "automate programmable" [PLC] made by Cegelec which together with a local PC allows direct local control by the magnet operator. The PLC communicates via RS232 to a dedicated slow control crate and from there with the online cluster (task APMON) in the standard way. A second PLC for the "Système d'Acquisition" (task SA_READ) was provided by Gardy together with a local PC that communicated directly via Ethernet with the cluster. Generally, since this was a separate controls system with a clear communication interface to the ALEPH control room and it was reasonably maintained until the end of LEP, there were no serious problems with it. Actually, the outsourcing also allowed an extension of the task SA_READ at a critical moment to add control of a mass spectrometer via the online cluster, which was needed when the vacuum tank of the magnet sprang a leak. This addition was for a short period of time, so the problems of long-term maintenance did not creep in. By contrast, buying the bubble-counter PC, also from Gardy, without software and maintenance was surely a mistake for the reasons described above.

4.3.2 - Software

The kernel of the slow controls system used at the start of LEP was a central slow control server process on the VAX/VMS cluster called BRIAN, which provided the interface between the user applications and the distributed 6809 microprocessor system. This centralised server turned out to be a bottleneck for the transmission of data. When the front-end processors were replaced by the VME-based 68020 cassettes

connected via Ethernet, the software on the micro's and on the mainframe was completely rewritten. The main features of the new software that ran until the end of LEP were device orientation, distributed applications with no central server as well as enhanced local monitoring capabilities in the micro's.

Several digital and analogue channels could be assembled into "devices" in order to control and monitor complex items as single entities. Examples of these slow control devices are Fastbus power supplies, HV channels, special triple-voltage "Gating" power supplies, etc. The application software acted on a number of devices by performing high-level commands without the need to know the internal details of each device. The actual translation from device commands to low-level actions on discrete channels was done in the VM20 processors. The description of the devices was stored in a single database shared among all applications. To reduce overhead, the VM20 software did not access the database directly; instead all the information required about the device was sent together with the command. This was hidden from the application programmer, so writing programs was simplified.

The end-user applications, specialised typically around one sub-detector, could run on any of the computers of the Cluster. They communicated with the VM20 by Ethernet and using the AMS package. The main function of the sub-detector control applications was to produce a coherent picture of the state of the sub-detector control devices. This included handling and reporting warnings and alarms detected by the VM20 processors and providing a user-friendly interface based on menus. Most of the monitoring (cyclic reading of values and comparing with limits) was done in the local VM20 processors, thus offloading the network and online computers of that task.

Some global control applications dealing with certain specific parts of the overall system, for instance HV control of ALEPH as a whole, were implemented by interacting with all the sub-detector applications. The interaction was done at very high level, basically by setting each sub-detector subsystem to a global state and following some specific rules. This offered the operator in the Aleph Control Room the possibility to control the whole experiment in an integral fashion.

The central slow controls database was kept very flexible and free of imposed restrictions, safeguarded mainly by relying on the trustworthiness of the users. Device names could be freely chosen, multiple names for the same hardware channel were allowed, parameters could be changed throughout the database by any user, there were no enforced naming conventions or domains reserved for each sub-detector, although there were some common-sense habits, such as using names starting with "TPC_" to characterise that sub-detector. Amazingly, this guided anarchy did not lead to any serious problems throughout the lifetime of ALEPH, although it may not be the solution for future experiments.

Access to the full Slow Controls was open to all users logged into the Online Cluster, which opened the door for mischievous use by allowing, for example, high voltages to be changed from a station attached anywhere on the globe while data was being taken. It also allowed convenient debugging from home, which shortened delays for quite a few interventions by experts. During the lifetime of ALEPH, while errors did happen due to bad judgement of the expert remotely trying to repair something, there was no case of sabotage where somebody knowingly tried to disrupt the experiment. Again, this does not mean that for the next generation of experiments the risk of having such sabotage happen should be repeated.

What was missed was a logging of all changes made in the database with time stamp and operator information plus possibility of comments. Changes of limits, nominal values or scale factors, etc. were sometimes necessary due to drifting sensors or exchange of an I/O card with one of slightly different characteristics or such a banal thing as unplugging and re-plugging a cable, and it would have been useful to have a record of such changes. This would have been helpful when looking at historical data to differentiate between true changes of given parameters and those changes that were induced by interventions.

4.4 - Communication with LEP

The operation of the detector is tightly coupled to the status of the machine. We should take data when there are 'Stable Beams', get ready for that as soon as possible, and get ready for the next injection before LEP is ready to deliver it. We want also to log enough information to get the energy of the machine with the best accuracy. We also send back to the LEP operators information about the experiment, such as the status of our magnet, background estimators and luminosity numbers.

In the first years, a dedicated communication protocol was used, with predefined blocks of information. This was difficult to maintain, adding new information implied rebuilding of the communication software provided by LEP division. Sometime in the mid-90s, this software was replaced by a simple ORACLE client-server interface. Each piece of information is an ORACLE table, or a row in a table. Adding new

information means either adding a new table, or adding a new column in a table. Both are easily handled, as there is no need to synchronise the changes in the software with changes in the database.

For performance reasons, the client should not read all the data every few seconds, because most of them are not changing very rapidly. The mechanism of sending an interrupt to the client when something changes was not used, as this was not felt to be very safe and reliable. The working solution was to implement a small table, containing a list of available systems and the time of the last change. The client reads this table every few seconds, and decides to read only those tables that have changed and that he is interested in. Adding a new entry to that table is also transparent.

Since this was put into production, the communication with LEP became very smooth. The main source of the remaining problems was when the LEP ORACLE system had problems, was unreachable, or was returning only error statuses. Another source was due to the mechanism used to retrieve certain rows in history tables: one needs a big enough buffer, as the required buffer size increases with time. Proper handling of the return status from the ORACLE call is needed to diagnose this type of problem.

Sending to LEP was also performed with an ORACLE client, and never had any problem.

5 - MONITORING

Monitoring was performed by looking at the data and filling histograms or time charts. Many other parameters were also monitored and displayed, such as those accessed by the Slow Control system.

5.1 - Monitoring template

Each sub-detector group was responsible for developing the tasks used to monitor its own data. The Online group provided the framework to build such a task, so that the expert had only to describe his histograms in the histogram database, to fill them in the event analysis routine, and optionally to perform some processing at start and end of run. The rest was taken care of by the monitoring template: booking and resetting of histograms, initialisation of time-charts, connection to the event buffer, handling of the FSM control protocol and similar functions.

This approach proved to be very successful, avoiding every user having to understand all details of the system. Having only one or a few source files to maintain made life easy. However, any change in the template, or in any package used by the template, implies re-linking every user task.

5.2 - Histogram database

In practice there were about a dozen monitoring tasks, each typically making use of several hundred histograms. It proved to be very useful to describe the histograms in a database, and to use the information in the database to automate their booking, their saving at run change, and to describe standard pages to be displayed by the Presenter.

Typically we had many histograms of a similar type. For example hit distributions in the TPC were displayed on a per sector basis, one histogram per sector. We therefore introduced the concept of a "histogram type" which defines the HBOOK properties (number of bins etc.), and every histogram was just an instance of this type, with an individual ID and title. Histograms were also saved to disk at regular intervals, typically at the end of each run, but also after a given time or after a given number of events. These were called "Savesets", and all histograms of a given histogram type belonged to the same Saveset. Each saveset was associated to a task, namely the task responsible for booking the histograms. It could also be attached to an analysis task.

The database also contained a description of the about 50 N-tuples of TimeCharts (see section 5.4) and also held descriptions of about 400 pages of histograms that could be displayed by the Histogram Presenter.

5.3 - Automatic analysis

Once a saveset had been saved, a task was woken up to analyse the histograms in the saveset. The purpose was to measure peak position, detect unwanted spikes and holes in various distributions. As every run produced a few thousand histograms, with a run as short as 20 minutes in some periods, it was not realistic to ask the shift crew to look carefully at each histogram. Automatic analysis was then very useful to guarantee a standard check of all known problems. One useful feature was the existence of "reference histograms" to which the current one could be compared, inside the Presenter for display, or in the automatic analysis tasks. As the LEP energy was changing each year in the last few years, with some data taken at the Z0 peak for calibration, we in fact had several sets of reference histograms, for the various running conditions.

The difficulties with this automation are of two sorts:

- False alarms due to statistical fluctuations, in particular for short runs. A careful tuning of the cuts is needed, with some checks on the statistics of the run. There is a tendency to increase the threshold to avoid false alarms, but then the detection efficiency is reduced.
- Known problems are producing an alarm every run. We have implemented in some cases a filter, which masks known alarms. Unfortunately, this was not made standard early enough, so it was not used by all tasks. One problem is that each task is maintained by a different person, often a graduate student who left after a few years. Changes to the task architecture are very difficult to obtain after a few years, because the task is perfectly working except that "minor" detail, and that manpower for maintenance is always difficult to get.

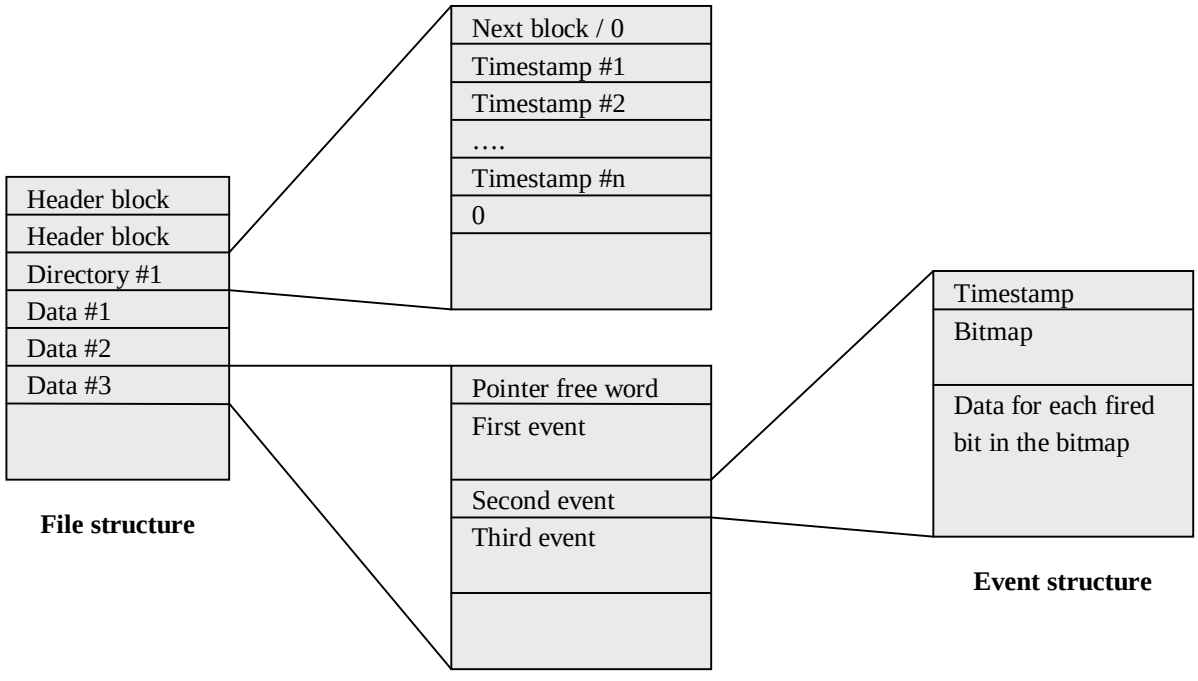
5.4 - TimeCharts

One important task of a monitoring system is to display the evolution of some values with time, so-called TimeCharts. Usually, many values are produced and stored as a block (N-tuple) such as low voltages of a detector, and many of these parameters are very stable i.e. fluctuating only inside the measurement error. We therefore built a software package to store and retrieve these values in an efficient way. The ideas of this storage format are the following:

- Seen from the user code, data are stored as an 'event' in an array of fixed length. A time stamp is stored with the event, an integer number of seconds since some origin.
- The stored data are zero-suppressed, i.e. values of a variable that are compatible with the previous stored value are not stored. If a complete event is suppressed, there is no entry at all in the file. The tolerance is specified in the database for each individual variable. A bit map is present at the beginning of every event, which specifies which variables are specified in this event. As we want each block of events on disk to be autonomous, the first event of every disk block is NOT zero suppressed.
- A directory indicates the first timestamp in each disk block. A possible chaining of directory blocks makes the format expandable ad infinitum.
- A few blocks at the beginning are used to store a description of the TimeCharts, such as variable names.

This format allows a fast access to the data, just by scanning the directory blocks, and then the data blocks to find the first interesting event. When reading a data block, the first event is read to become the current event in memory, and this current event is updated with the zero suppressed data of the succeeding events in the same block. This is how we reconstruct the original data block.

Using this compressed format, we can keep online more than one year of data, with many measurements performed every minute, such as all low voltages, and some measurements can even be stored every few seconds. As most values are stable, the zero suppression is very efficient, and allows a very precise storage in case of spikes without saturating the disk with constant values.



Another feature of the TimeChart system is that it has a simple programming interface, so that one can easily extract information from a given time range. This allows trends to be monitored automatically, or to build web pages on the fly with the TimeChart content.

5.5 - Presenter

This tool was used for visualising the histograms and TimeCharts produced by the monitoring tasks. It was based on the X-windows standard, on HBOOK for histogram storage and on HIGZ for the graphic interface. Several features and ideas are discussed in the following sections.

5.5.1 - Page description

The Presenter displayed only pre-defined pages stored in the Histogram Database, as already mentioned in section 5.2. There were of the order of 400 pages available, which meant that a hierarchy needed to be established in order for the operator to find and retrieve the page of interest conveniently. Pages belong to an 'owner', and selecting the owner restricts the list of available pages to a reasonable number. The 'owner' is the system being monitored, either a detector or a sub-part of a detector: one separates for example data monitoring from slow control monitoring pages when the number of pages becomes too big. A special owner defines a set of 'shift crew' pages, which have to be checked frequently by the operator during data taking. The page is described as a list of boxes. Each box is defined by its position and size in a grid defined by cutting the page in a number of lines and of columns. Usually all plots on the page occupy a box of the same size. Each box specifies the plot (histogram or TimeChart) to be drawn, and some optional display parameters to control the display of titles, axes, and other parameters. In fact, one can also superimpose plots (histograms or TimeCharts) in the same box.

Based on the standard database editor, a page had to be described by line mode commands. In order to put two plots on the same page, one has to create a new page, and specify the plot parameters and positions by numbers in the editor's menu. A graphical editing would have been easier to use. **One advantage** of this complex manual editing is that **the page layout stays constant over several months**.

Another important feature of the page description is that one can have many plots on the same page. With 48 faces in VDET, or 36 sectors in the TPC, it is convenient to display the same information on these 36/48 sub-components. This makes comparison easier.

5.5.2 – Plotting data

Once the page is defined, one needs to specify which version of the data one wants to look at. Usually, one looks at the current data (ONLINE mode). The histograms are spied in the monitoring task's common block, implemented as a VMS global section, and the TimeCharts are accessed from a specified duration in the past, specified by the number of hours or days. However, it is frequently useful to look at data from a given run, a given fill, or a specified time interval. All these possibilities were implemented. Access to the Run and Fill databases is needed to get the corresponding time interval. When displaying the data of a run or a fill, a button allows displaying data from the next or the previous run/fill, a useful function to analyse the evolution of a problem. Selecting a range of runs or fills displays the sum of the histograms of these runs and fills, which is useful to increase statistics when investigating a small effect.

As a guideline, it is essential to display also a **reference histogram**. For each saveset, a reference file exists, which is (optionally) displayed with the histogram in another colour. In fact, several reference files exist, corresponding to different LEP energies. The proper one is selected according to the machine energy at the time the data was taken. This may be ill-defined when a range of runs with different energies is selected. For TimeCharts, two reference values are available in the Histogram Database, which can be either displayed as guides (horizontal lines) to check the value, or used to force the vertical scale of the display. This permits to detect if the variations (usually scaled to fill the vertical range of the plot) are meaningful.

5.5.3 – Display Options

Clearly the Presenter is not an interactive analysis tool. Nevertheless, many options were available to change the presentation, mainly re-defining interactively the display flags that are in the database. One can also zoom on a plot, selecting the horizontal and vertical scales interactively. Another useful feature was to display in number the content of the bin where the cursor is positioned, in either a histogram or a TimeChart.

The definition of the time interval for TimeChart was interactive in "Online" mode. One plots the date from a certain time in the past up to the current time. This origin is defined by a number of hours, or of days, before the present time. Changing this time range should be easy, as one wants details (few minutes full

scale) on rapidly varying information such as background, while a range of a weeks is good for looking at slowly varying information such as temperature or pressure changes.

5.5.4 - Help information

Each histogram and TimeChart should be documented, so that the shift crew can understand what is plotted, and what are the meaningful features of the plot, i.e. what is a normal variation, and what indicates a possible fault. Having a folder with a description of each plot is impractical; it is better to attach a document to each histogram. This is performed by adding a help keyword to the histogram type, as the same help applies to all the plots of the same family. We used a VMS help file, with the name of the filling process as file name. The help item is defined by a mnemonic string. The user needs just to click on the "Help" button, and on the plot for which he wants some help, and the Help text is displayed. This help panel has also a button to report that the help isn't good enough. This report was processed by the operation piquet, who tried to convince the experts to better document their plots. Endless work!

It would have been nice to have a common approach to HELP information from the Presenter and from the Error Logger (and others). Much of the information concerning a monitored variable is the same in both cases, such as the location of the probe, the nominal value and the likely variations, the significance of an excursion of the value and the preventive actions to be taken. Any of this may change for one reason or another and the HELP description should be changed centrally once only to ensure consistency of information. To cite an example: the thermal contact of a temperature probe measuring cold water through the pipe mantle was improved, thereby changing the nominal value and with it the allowed deviations.

5.5.5 - Error histograms

As indicated in Section 5.3, an error may be generated according to the content of a histogram. When this occurs, the Presenter is informed, as it is a client of the Error Logger for this type of message. The Error Message contains the histogram number and file name, so that it can be easily displayed by the Presenter. The only difficulty is to find the related information to this histogram, mainly the reference plot and the help information. This is performed by massaging the file name to get back the saveset, and using the histogram ID to find the database entry. Being able to display the faulty plots with their reference and help is a very useful feature.

5.5.6 - Performance issues

The standard GKS driver was too slow for our needs. As most of the primitives are also available in the Xlib library, we have developed our own GKS interface, limited to the functions we needed. This was fast and reliable. However, this prevented us from using a second device driver for Hardcopy. We had then to re-invent a metafile for hardcopy: When requesting a hardcopy, each GKS interface routine is writing its arguments together with an identifier on a file, producing a metafile. A program, linked with the GKS real libraries, reads back this metafile and calls the real GKS routines, allowing use of a Postscript driver, the way we print the display. Printing a page is performed by opening a metafile, setting a flag, drawing the page as usual, closing the file and spawning a process running this metafile reader and sending the result to the selected printer.

5.6 - Event display

The online event display has two functions: spotting and understanding problems in the detector, but also having a nice support for explaining the experiment and the physics to visitors. Even if this 'public relations' function was not part of the design, it is an important aspect. Many 'open days' in the various home labs were using an online event display to show how things look like in an experiment.

The main function of the display is to provide a global picture of the data of one event, to check their consistency and correlate them on an event by event basis. All statistical analysis is performed using histograms and the Presenter.

5.6.1 - View selection

Several pre-defined pages are available, each page containing one or several views of the event. A standard page was providing the most useful views, x-y and r-z projections, plus end-cap views and zoom on the central detectors. Other views and pages are available on request, with calorimeter maps, LEGO plots and specialised views for some detectors. One useful feature was to automatically select different views for different types of events, selecting special views for the luminosity triggers, where only the two LCAL or SiCAL detectors where hit and readout.

Having non-editable pages is a restriction, but gives long-term stability to the system. The view you see has the same layout as a few months ago, and you understand easily what is what. For a tool shared by about a hundred persons, shift crewmembers and SD coordinators, it is better to be not too configurable.

5.6.2 - Event selection

The most natural source of events is the online event buffer, and that was the default mode. A facility was provided to select the events to be displayed according to their trigger mask, and/or their Level-3 classification. The main use was to suppress calibration triggers, like random events and laser shots, which are of limited interest in normal running mode. A selection of hadronic events was also available, and used to display nice candidates in the LEP control room, and also for visitors: The rate of these events was as low as one every few minutes at LEP II.

The event display was also able to read event files, which was very useful to understand pathological events. 'Typical' events were also extracted and collected in dedicated files, used during visits and open days to display the various types of events even when the machine was not providing physics.

One important feature is the "automatic refresh" where a new event is displayed every few seconds. This is the normal operating mode, and the shift crew just watch the screen, detecting unusual patterns. The refresh frequency was controllable, but was usually 5 seconds.

5.6.3 - Analysis facilities

A few tools were added to the event display, to perform simple actions interactively. Standard commands to print or to save events were available. Most importantly, a few commands were implemented, to sum calorimeter data in a selected region, or to give the hardware address of a selected calorimeter cell. This worked only on the θ-φ view, and was frequently used during the first years, when the ECAL electronics cards had a failure every week or so.

5.7 - Status screens

One of the important interfaces for the shift crew was the large collection of status screens. Using simple text terminals, they were constantly displaying various numbers and parameter values. About 20 such screens were in front of the shift crew's eyes. The point is that the display programs were 'smart', and many displayed numbers were checked, and displayed with reverse video, bold face or blinking mode to attract the attention to their non-standard value. Beam loss, HV trip, change in machine energy, high trigger rate, high background are examples of these easily spotted anomalies.

Each of these screens was also available via a simple command, that any expert could run on his terminal in his office or at home. **Remote access to the same information as the shift crew** proved to be very valuable when answering calls or trying to fix problems without being in the control room. Moreover, a stupid VT200 compatible terminal was the only thing available at home, not counting that these displays were also running on Minitel, giving the possibility to access whilst on vacation!

5.8 – Monitoring System Efficiency

We also have tasks that monitored the efficiency of the complete online system. In particular, a task called **PBLOG** monitored the behaviour of the DAQ system, logging each incident using extracts from various log files automatically put together, and recording a measure of the lost luminosity. Each incident was then classified, and an analysis tool allowed identification of which problems were the most important to fix. We always assigned the whole loss to the initial cause, even if the attempts to recover had created by far a

worse situation: If the initial fault is removed, the full efficiency is recovered. An objective measure is really important, as many short incidents usually cost more than major but rare incidents, which are better remembered. At the weekly meeting of the Online group, the week statistics of losses per incident type was discussed, and work priority set according to the expected gain in performance. At the beginning, some DAQ errors were the main source. Later, hardware failure like crate trips became the main issue, showing that an automation of the recovery was a good investment at that time.

The efficiency was carefully monitored, and loss classified in 3 categories: HV not ON when LEP had physics, DAQ not running when HV was ON, detector busy while DAQ running. This was known as Operation inefficiency, DAQ inefficiency and Dead Time. Those numbers were permanently displayed on a screen in front of the shift crew, soon nicknamed "**Big Brother**" by the shift crew. Having the efficiency always visible motivated better operation and gave better efficiency. However we never tried to correlate efficiency and shift crew names, as the initial error is (almost) never due to the person on shift.

The daily report was also summarising the performance of the past 24 hours, and the cumulated performance was reported to the bi-weekly LEP schedule meeting.

This sort of measurement is never perfect. Firstly, because it is done by polling every 5 seconds, giving a quantification of the length of any incident, such as a HV trip or a DAQ stoppage. A waking up by incident would have reduced this problem. Secondly, the loss is measured not in time, but in luminosity, and we need the instantaneous luminosity. We used the last measured instantaneous luminosity, obtained by counting luminosity events during 2 minutes in a dedicated detector. This luminosity is then out of date by 1 to 3 minutes, usually not a problem as the luminosity is stable, except at the very beginning of a fill or after a partial beam loss. We started to take data just after the beams were put into collision in the last years, and the initial luminosity was underestimated. The solution would have be to gate the counting of the luminosity triggers, but as this counting was done in a very front-end processor, which had no information on the HV or DAQ status, this was not easily possible. Moreover, doing a major change just to improve the last significant figure of a simple indicator was felt not to be a good investment of our time.

6 - AUTOMATED OPERATIONS

The tasks of the ALEPH Online system were designed to be controlled by an operator via menus. This means that the main or only interface of most tasks is the UPI menu system. One exception is the FSM protocol used for run control, where a task receives commands from a master to execute transitions. These FSM tasks are then designed to be slaves, with limited use of the menu interface. In this note, we will consider only the "masters" which are the Run controller and all slow control tasks. They were first operated by the shift crew, or by sub-detector experts, but after a few years, it became clear that many actions could be automated, thus becoming more efficient. Several systems have been built to perform various tasks,

- DEXPERT: An expert system to handle problems in the DAQ system. Handles trigger protocol errors, DAQ errors and 'death' of tasks in the Run Controller environment.
- FBFIX: A dedicated task for handling Fastbus and VME crate trips, able to perform slow control actions and to drive the Run Controller to recover the errors generated by the crate's failure.
- AUTOCAL: A sequencing task handling the calibration activities performed at end of fill.
- ZEBEDEE: The central HV controller, able to decide to ramp the voltages according to the LEP state and to the background situation.

Before reviewing these various tasks, we want to comment on the way an automated system can be triggered.

6.1 - Detection of an event

Usually automated systems are triggered by external inputs to detect changes of conditions. This can be by incidents, in which case the system is woken up by a message, or by polling on status variables. Both methods are used, the most frequent one being the reaction to external stimulus, and we have two implementations.

6.1.1 - Error logger

This system is described in section 3.11, and its forwarding mechanism is the most frequent triggering method. Unfortunately, this mechanism was designed for internal use only. In fact the user receives a copy of an internal data structure, which is not very user-friendly. A layer of software should have been built to help handling in a simple way incoming-forwarded messages.

6.1.2 - The Incident Server

This separate system was implemented to broadcast some events (incidents), such as start of fill, end of fill or run change, to interested tasks. The idea was to avoid the difficulty of decoding the content of an error message mentioned earlier. Here, a user-specified routine is called when an interesting incident takes place. The incident must be declared by another task, which should produce only "interesting" incidents and thus understand more about the context. A second difficulty is that the producer of the incident doesn't know who is using it, and can then not modify the definition of the incident without creating possible side effects. In fact, only two incidents are used: the "Beam Lost" detection, to trigger the End Of Fill activities, and the "Open Partition" which informs that something has changed in the running context, and that the partition description must be scanned again.

One inadequate feature of this system is that the "incidents" are thought of as states, and so as soon as the task registers for a class of incidents, it receives the existing incident. However, the main use of this system is to receive notification of events, not of change of state. Each program must know this 'feature', and must ignore the first incident it receives.

The Incident Server is in principle a better method to trigger actions, as the logic to detect the condition and to filter spurious events is done in a single place. However it was implemented only late in the life of the experiment, and used only for new functionality. Changing is always difficult, and once the dirty implementation of decoding the Error Logger message was working, it is difficult to be motivated to replace it by a new one, with no real gain.

6.1.3 - Polling

This method is simply to look regularly (using an internal timer) at some variables in a global section, describing for example the status of LEP and the background numbers, and to decide that an action is now needed. This could be implemented also using the Incident Server, but it just means that the logic should be implemented in the task creating the Incident. There is no gain as long as the Incidents are used by a single task. Polling has the difficulty that the sampling rate should match the rate at which the information is updated, in particular if one wants to be sensitive to spikes or other short duration events. Polling also adds a variable delay between the change of the information and the system's reaction. Some information is already obtained by polling, the LEP data for example, which are moved to a common publishing area by a polling mechanism, and then polled by the final user. All the delays are adding up, so one should select a relatively high polling rate (few seconds) at each stage, provided this does not overload the system. This is the usual drawback of polling. But the other mechanism implies that another task knows what you want to know, and wakes you up by a message when this event occurs. This induces some coupling between tasks, a source of long-term maintenance problems.

6.1.4 - Self decision

A special case was the automatic run change decision, taken by the disk-writing task. For technical reasons of tape size, the file size of a run was limited. We also wanted to limit the duration of a run so that even at low luminosity we never run for more that two hours on the same run. This is because we rated the quality of the data per run, such that a run is entirely discarded is a problem is detected inside. Histograms are saved and analysed only at run change, for the same reason.

When one of these conditions was detected, the disk-writing task sends a message to the run controller. As this was the first case of someone controlling the Run Controller, an ad-hoc implementation was used. There is no protocol, no handshake, this is just a message. It would have been better to re-implement it later, after some experience with other systems. The decision could have been taken with maybe other system status parameters, and we would have used a more standard communication protocol, with acknowledge and retry.

6.2 - Examples of automated systems

6.2.1 - Expert system: DEXPERT

This was the first system introduced to help and/or to replace the operator. DEXPERT is a real Expert System, written in OPS5. The Error Logger triggers it, as any task that detects a trigger or event building problem reports that way. The actions are mainly commands to the Run Controller.

The tasks detecting a problem should give clear information to DEXPERT. The quality of this information is essential for an efficient operation of DEXPERT. For example, the error report generated by the trigger supervisor contains a hidden part for DEXPERT, listing the faulty processors.

DEXPERT is programmed in OPS5, which is technically a good choice. However, this language is used ONLY for this program. It was possible to have experts of that language when DEXPERT was built, but the maintenance is very problematic. One needs only a few changes each year, but they require a lot of effort from the only person who has some experience with the language, and who is the only one able to debug any problems. In practice, development is no more possible, only minor alterations can be safely performed.

As already mentioned in section 5.8, one important use of the expert system is to log properly all problems and their duration, allowing a measurement of the efficiency loss due to each type of problem. This is the way to put resources in the most useful place. For some infrequent problems, it was more economical to let DEXPERT handle them. A typical example is when the cause was a hardware bug, which could not be fixed with reasonable effort. A 15 seconds loss in average every day is acceptable if the only other fix is a major hardware intervention, which could generate other problems.

6.2.2 - Fastbus (and VME) crate handler: FBFIX

A classical problem requiring a lot of knowledge is the recovery from a Fastbus crate trip. When such an event takes place, the Slow Control monitoring system detects the failure. But also the DAQ system gets

into trouble, with various possible trigger and DAQ problems. The sequence of operations is reasonably simple, but involves many systems. It was not implemented inside DEXPERT due to the language problem mentioned earlier. The sequence of operation is as follows:

- Disable DEXPERT. As DEXPERT ignores the real cause of the problem, the rules it knows are not appropriate. This is the first action. It also participates of the good practice to have only one master in the system.
- Turn ON the faulty crate(s). The crate's status is checked again 5 seconds later, to make sure the crate is not severely broken. This is repeated at most 3 times. If unsuccessful, FBFIX gives up, and warns the operator loudly, as manual intervention by an expert is needed.
- Identify which readout processor has to be rebooted. A connection between the Slow Control and Readout databases is needed. In fact, one needs also to know if the faulty crate is used in the readout, as there is no need to stop the run for a fault in an unused crate. This is also not an easy question.
- Inform the Run Controller that these processors will be rebooted, and that it should not try to control them to reset the run. Reboot the processors by an external command.
- Reset the run. Wait for all processors to be rebooted. This is performed by polling on the existence of a monitoring data module, created in the last step of the processor's start-up.
- Re-initialise the Fastbus tree. One needs all processors to be ON for that. In case a VME crate was faulty, re-initialise the VME readout tree.
- Restart the run, with retries if needed, and usually this is needed!
- Enable the trigger, re-enable DEXPERT and go back to sleep.

The process is iterative, in the sense that if another crate trips, one handles this new crate alone up to the point where the other was, and then continues with all the crates. This can be seen as many state machines, one per crate, for the first part of the recovery, and then a single machine for the second part, handling the restart of the Run Controller. There are protections to avoid infinite loops. In particular, if the same crate trips again during the procedure, this indicates a real hardware fault, and the expert has to be called as soon as possible to change the crate. FBFIX should not loop forever with a crate that stays ON for only 5 minutes…

FBFIX takes over the global control, it has precedence over DEXPERT as mentioned earlier. It also prevents AUTOCAL from controlling the run. One may think that a coincidence of two problems is infrequent. This is not true: A power glitch will trip some crates, and may trip the machine RF: The beam is lost at the same time.

6.2.3 - Calibration and end of fill activities: AUTOCAL

This system handles the repetitive sequence of operations one wants to perform between data taking runs. It is triggered by the incident "Beam Lost", and is in charge of performing the three types of calibration and to set the high voltage to the required state. The first calibration, for the silicon detectors, has to be performed with their voltage ON, and so has to be finished before the next injection. Even if an interlock has been installed, preventing injection in LEP until the voltages are in the safe state, it is quite important to finish this operation as quickly as possible. One doesn't need a calibration after every fill. A minimum time interval has been implemented and calibrations are skipped if the last one is recent enough. Another constraint, the high voltages should not be turned ON by AUTOCAL, this is too risky without knowledge of the machine conditions. The "High Voltage ON" calibration is then skipped if none of the three silicon detectors is ON. However, the current run is always properly stopped and the high voltage set to the "Ready for Filling" state.

AUTOCAL is asking the Run Controller to perform each calibration, to stop and to start the run. As one may have failures, a retry mechanism is implemented with a maximum number of attempts. AUTOCAL checks also that FBFIX is not in control. If this is the case, it waits for FBFIX to finish, and then starts to process the end of fill operation. One can describe AUTOCAL as a simple loop in a finite state machine, but the number of states in the loop depends on the need to perform the calibrations.

An operator interface exists in AUTOCAL, so that the sequence can be triggered manually. In fact, one wants recent calibration BEFORE taking data, and for that the operator should trigger a calibration after a long stop of the machine, or of the experiment. The operator can of course disable AUTOCAL at any time.

6.2.4 - High Voltage Control: ZEBEDEE

Zebedee was first intended to give to the shift crew a common interface to all SD high voltages. A special protocol has been implemented, in which each task declares one or several systems, and for each system a list of states, and for each state a list of possible states that can be requested. The state of each system is determined by the task, and reported to the master (Zebedee) each time it changes. Zebedee is just a status and control panel, allowing requesting a change of state for each system. Note that the state of a system can change spontaneously, for example in case of a trip, and not only upon request from Zebedee.

Rules have been defined to ramp the high voltages of the various systems according to the LEP situation and to the measured background conditions. After a few years, these rules have been coded, and Zebedee started to handle global states, from "Ready for Filling" to "Physics" in 6 steps. The last improvement was an automatic mode, where a change in the desired state, computed from the rules, triggers an action on the high voltages. This automatic mode is delicate, for various reasons:

- Setting the wrong high voltage can damage the detector, or trip it.
- Some detectors cannot partially ramp down their voltage. One can only stay as is, or ramp up, even if the background worsens.
- The ramp-down at the end of a fill depends of what AUTOCAL wants to do. In fact, AUTOCAL is ramping down the voltages, and Zebedee should not ramp it up at the same time. These two tasks need to co-operate to avoid producing an infinite loop.
- The shift crew becomes used to the system behaving correctly, and is no longer checking carefully that everything behaves as it should. Misbehaviour can remain undetected. In particular, the system doesn't recover automatically from trips. When a detector trips, the global state becomes unknown, and as Zebedee progresses automatically only from known state to known state, there is no more automatic progression towards physics conditions. An independent alarm should then be issued to wake-up the shift crew!
- It is delicate to test the response, as it depends on the LEP sequence of operation. In addition, this is changing with time, with beam energy, with background conditions.

The various LEP and background values are looked at every few seconds, and a decision on the wanted state is taken according these values, to the previous decision and to the current high voltage global state. One limitation is that Zebedee does not handle trips. Only the normal ramping sequence is performed automatically, as deciding how to recover from a trip is too delicate.

6.3 - The Multi-master problem

From the description of DEXPERT, AUTOCAL and FBFIX, it is clear that these three tasks may want to control the Run Controller, which can also be driven by the operator from the menu interface. We have then a multi-master problem. The first aspect is the competition between the tasks. This is implemented as a hierarchy, FBFIX blocking AUTOCAL and FBFIX disabling DEXPERT. AUTOCAL and DEXPERT can co-operate. The second aspect is the Operator intervention, which has several aspects:

- How does the operator know who is in control? This is not always simple to see from the main control window, as a start of run or a calibration activity generates a lot of messages, and the single message of FBFIX or AUTOCAL saying "I'm now in control" scrolls quite quickly off the window.
- How can the operator stop a task? Each system has a "Disable" button, but it may take a long time for the real operation to stop, as one command as "measure the pedestals" may take many minutes to execute. So even if the control task is disabled, activity continues. This is usually what we want, but the operator may be confused.
- How to restore the normal running mode? DEXPERT must be enabled during data taking, as this is the most efficient way to recover errors. Once disabled, it complains every minute that it is disabled. But in case of crisis, this complaint may add to the stress of the operator.

The co-operation between the various tasks is not always easy, or simple to understand. For example, the high voltage is ramped up by Zebedee, and ramped down by AUTOCAL. Zebedee should then take into account the last request from AUTOCAL, but not permanently, as it has to detect the next normal LEP ramp.

In order to solve these conflicts, it may be tempting to build a single global master, integrating the various tasks, and having a single operator interface. This solves only partially the problem. As the various functions are well separated, they will correspond to quite independent sub-parts of the global master. The communication between these parts will not be easier than the communication between independent tasks. The only gain is a single operator interface, where the question of who is in control and how to stop an activity could be clearer for the operator. In fact, the best approach would probably be a referee task, which would arbitrate the requests from the various tasks, including the operator. A common publishing area could help the collaboration between tasks. It should be kept in mind that new tasks (new functions) are added in the life of the experiment, so this global master should have an open architecture.

7 - DETECTOR OPERATIONS

7.1 - People organisation

Supervision of datataking and of the operation of the detector was under the control of a shift crew of two persons. Experts from each subsystem were on-call to intervene in the case of problems. This way of organising the running of the experiment was implemented from the beginning. It implies that control and operation of the system must be made available centrally for all aspects, including the start/stop of datataking as well as the regular ramp up and down of the HV every fill. One member of the crew was designated the Shift Leader (SL) who had overall responsibility, the other was designated the Data Manager (DM) being primarily responsible for monitoring the integrity of the data being taken.

One of the basic tasks of the shift crew was to perform a safety tour, visually checking many safety systems. Performed every 3 hours in the early days, it was soon performed only during the shift change, where 2 crews are present simultaneously. The two SL's stay in the Control Room, the two DM's perform the Safety Tour. This overlap allows a good sharing of information about pending problems or new features.

Subsystem coordinators provided the first level of help in case of problems. Typically these were people on call for each subsystem (detector, trigger, DAQ etc.), usually for one week at a time but each subsystem had its own rules. These people had to be reachable at any time of the day or night, and they met every morning at 9 am in the conference room, near the control room. The Shift Leader also came to this meeting, which was chaired by the Run Coordinator. Every problem of the previous 24 hours was discussed at this meeting, together with plans for the coming days. This meeting was usually very short, but it forced every coordinator to perform regular checks just before the meeting and to present an up-to-date status report.

In order to help the subsystem coordinators, a "Daily Report" was printed every morning, one section for each coordinator, with relevant information needed to help pinpoint the problems: list of fills and runs, list of error messages produced by the detector's tasks, pending problems in the Run Quality Database, data quality rating of all runs and fills processed in the last 24 hours. The section for the Run Coordinator contains all the important information printed for each Sub-Detector, as the Run Coordinator's function was to ensure that all problems were addressed (and one knows that a natural tendency is to rate as unimportant problems difficult to address). The role of the Run Coordinator was to make sure the coordinators were really understanding the detected problems.

Each coordinator had a checklist to follow each morning, for some detectors 2 or 3 times a day. This was defined by the subsystem experts. The idea was to have checked the content of the Daily Report, and have performed more systematic checks before the daily meeting, so that the usual "OK" report is based on serious checks. The Shift Crew has also a checklist to be filled every half-hour during data taking, as a reminder that careful checks are needed.

A second level of help was provided by the real experts, who were contacted by the subsystem coordinator in case the problem was unusual or difficult to handle. These persons are not guaranteed to be reachable, and may not be based at CERN. In the first years of operation, these real experts were frequently called, and also the subsystem coordinators were well aware of hardware problems that could arise. After 10 years, the rate of problems became very low, and expertise was very difficult to acquire. New subsystem coordinators have to be trained to handle the usual problems (fuses, power supplies to exchange, cold start after power cut), but there was no way to train new experts.

7.2 - Control room organisation

The control room occupancy changes with time. During the start-up phase of the experiment, the room is always too small, without enough terminals. After 10 years, the room is too big, with only two people for 90% of the time, except for the peak period around the daily meeting where ten to twenty persons are working in front of an X-terminal.

7.2.1 - Main console

This is where the shift crew is working. They have many screens in front of them, and several keyboards. The function of each of them should be clear. Most of the screens are smart, and change the video attributes (reverse video, blink, bold, sometimes colour) when some value is not in the expected range. This is very

useful to attract the attention of the shift crew. We have not used special sound devices, only a normal beep to signal a new entry on the Error Logger screen. The main concern here should be to avoid saturation of stimuli during crisis, such as a power cut or a 'dirty' beam loss.

The number of control places should be small, typically one per person, no more. If one has to play frequently with several keyboards or mice, one will some day use the wrong one! The reaction of the system to the input should be clear and fast, to avoid a repetition of commands by the nervous operator, which may create trouble.

7.2.2 - Detector corners

Each detector has its own corner of the room, with one or two X-terminals, a few video displays, and all its documentation. A few non-allocated terminals were also available for general use. A compromise has to be found between terminals permanently allocated to absent users (paused screens) and keeping the detector-dedicated screens always available for the detector expert.

7.2.3 - Telephone

The ALEPH Control Room had 8 different phone numbers, and in the old days it had less numbers but as many phones ! It is useful to have several numbers to call directly an expert of a given detector. However, when only two people are on shift, it is not easy to answer a call at the other end of the room.

Wireless phones are available for the shift crew. This is great, as one can move freely in the room to look at various screens, or even go to the coffee machine to find the wanted expert. A headset + microphone system may be a bonus, allowing to work with both hands while talking to an expert.

7.2.4 - Printers

Paperwork is still frequent. One of the concerns is loosing a disk before it has been backed-up, so the information on each run is immediately printed and filed in a folder. Other information is easier to handle when on paper, particularly when one wants to compare two measurements performed at some distant times. Checklists and daily reports are also printed. In addition, a colour printer allows producing nice plots and transparencies for the daily meeting.

7.2.5 – White board

A large white board displays the list of 'on-call' people, for the various detectors. Their name, GSM, office and home telephone numbers are displayed. Other useful numbers are also listed for convenience. This helps the shift crew to find immediately whom to call in case of problems. And it has the flexibility to put temporarily someone else on call, or to give the number of a friend where the coordinator is, in case the GSM phone doesn't work there.

7.3 - Online documentation and help

The amount of documentation is very big. However, one should make a clear distinction between what is needed for a developer from what is needed by the shift crew to operate the detector. All the software reference manuals are in the first category. At the other end of the spectrum are instructions to react to a given alarm, which should clearly be accessed easily with a simple click on the error logger screen. In-between are the operational instructions, like start of fill procedure or crate trip recovery.

We have a single shift crew manual of about 100 pages, and a reference manual of about 200 pages describing the building blocks of the system. A good indexing is fundamental. These books are browsed once by each crew member during his first boring shift, but after that they are only searched in case of problem, with some stress and the need to be fast: How to handle a power cut, a magnet discharge, what to do in case of fire alarm… Clearly, the Web technology should be used for the coming experiments, to avoid rebuilding search engines and display systems. However, one should always keep a paper version of the manuals, in case of no power. Note that a torch is also needed in this case.

7.3.1 - Electronic logbook

An electronic logbook was introduced in the last two years of running. The clear advantages of an electronic logbook as compared to the usual paper book are that:

- ASCII characters are always readable. Hand-written ones are sometimes difficult to decipher!
- It can be read from several locations at the same time. Finding the paper logbook in the minutes before the daily meeting was almost impossible. Even the shift crew had problems accessing it, as all experts were trying to read it. With an electronic version, the whole collaboration was able to concurrently discover the events and problems of the previous night while arriving at work in any place in the world, as it was Web-browsable.
- Entries can be automatically written by the computer to log usual events like change of shift crew, start and end of runs and fills.
- Entries can be written by people not in the control room, like experts working from home or down in the cavern.
- Old entries could be searched by a computer interface.

Possible inconveniences were also found:

- It is not available in case of computer failure, something which is very infrequent, and when it occurs we have not a lot to log anyway.
- As many entries are automatic, the shift crew tends to forget to write into it.
- Only text entries were possible

Nevertheless, the overall feedback is that its wide availability is a very strong improvement. A first attempt to introduce an electronic logbook failed, because the interface was not adapted: One should be able to add a few lines by just typing them on any terminal window. The first attempt had only a UPI menu interface, where free text input was not very natural. New experiments should probably use also Web technology to allow inclusion of figures and links to other items in the logbook. Both interfaces should probably exist.

7.4 - Shift scheduling

Organising the shift list was always a pain in all experiments, as many constraints have to be minimised together. The rules we adopted in ALEPH are the following:

- Shift crewmembers have to be registered in a database. They are assigned a qualification, from beginner to expert shift leader, and a number of shifts. Their mail address is stored together with some wishes like preference for grouped shifts, driving and French speaking capabilities, and the training dates. A minimum number of 20 shifts per year is requested, to give enough experience to people. This was however not a completely strict rule. Shift registration was closed as soon as we had enough volunteered shifts, but students arriving during the year were accepted for more shifts. This worked very well, but in the last two years it became difficult to get enough volunteers, and some arm-twisting by the management became necessary.
- A tool is available to indicate, for every single shift, if a person can take this shift. It is the responsibility of each person to update this VETO file. A week before building the shift list for a given period, a mail is sent to all shift crewmembers as a reminder to update their VETO.
- A program is then run by the Shift Manager to allocate shifts. For each shift, a weight is computed for each user, taking into account his availability, qualification, training, previous and next shifts, and making sure that at least one of the crewmembers is a driver, and at least one speaks French. The weighting includes also some factors to make sure everybody will have some night and some day shifts, and that those with the lowest fraction of their quota already assigned will be preferred. Shifts are assigned in blocks of 3 shifts at the same hour on consecutive days. The program proposes the list of possible candidates, sorted according to their rating. Usually, the first one is selected! However, some manual decision is still used for special requests.
- Once built, the shift list is sent to every shift crewmember. In case of problems, mainly due to a change in the availability, the person had to find a qualified replacement (a tool gives the sorted list), make the deal, and then inform the Shift Manager for the database update.

- The shift database is interfaced to many tools allowing displaying at various places the name of the persons on duty. The database contains also the list of detector experts on call, as this is useful to know, and avoids putting on shift a person also on call.

This tool was quite useful, and allowed building one month of shift list in 5 minutes. However, there are still some problems:

- There is no global optimisation, i.e. one looks for the best person for a given shift, but not the best shift list for a given person. In particular, people unable to come to CERN during the academic year must have more shifts in summer, and this is done by manually forcing the program.
- The VETO program can be used in an unfriendly way, for example refusing all nights and weekends. A manual check of the VETO files is then performed before assigning the shifts. However, one cannot prevent people from vetoing several months if they have teaching constraints. It is then difficult to automate these checks.
- People may be unavailable at the last minute. A piquet is then available to replace any shift crewmember. A call for volunteers can also made in this case, but without guaranteed response.
- Tools such as VETO were implemented using VMS features, which cannot be ported to Unix. It is clear that today we would implement them using Web technology, with some protection using user authentication.

Another issue, which can generate endless debates, concerns shift starting times. The compromise we ended with was based on the availability of public transportation to CERN from Geneva. This means we started at 6:30, 14:30 and 22:30, and ended 8 ½ hours later, giving a 30 minutes overlap for information exchange and to perform the safety pit tour.

7.4.1 - Shift Training

Training is mandatory for everybody every year. The problem is to train about 70 persons each year, half of them being newcomers. After several years, it became clear that the training has to be tailored to the expertise level, with dedicated sessions for newcomers. The training was also different for DM and SL, as they have to know different systems. One common training was on Safety Rules, a common and mandatory session. For experienced people, the training was concentrating on new or changed features. The length of a training session was half a day. For newcomers, we forced them to take three shifts during physics as "number 3" so that they are on shift but not in charge and such that they can learn with a trained person in real conditions. During these 24 hours of "number 3" they could get practice which cannot be given by training, and this worked very well. Unfortunately, it is difficult to schedule that there will be physics during a given shift, particularly at the start of the data-taking year, which is just at the time we need to train people! The training material was available in the control room for reference, and on the web.

7.5 - Interface with the Offline world

The Offline world was seen as a quite separate world, and not many tools were shared between Online and Offline. However, some information was passed to the Offline world in the data tapes, and by dedicated tools to fill the bookkeeping database. The Online system was also the host of the online reconstruction farm called FALCON, which was also used for winter reprocessing.

7.5.1 - Information in the Data Stream

The main component of the raw data tapes was of course event data. However, Start Of Run (SOR) and End Of Run (EOR) banks were also produced. The mechanism was that each task in the system could produce EOR or SOR banks, which were stored in a dedicated directory. At end of run, all EOR and all SOR banks were collected, and put BOTH at beginning of tape. Then the event data, which was written to disk during the run, are copied to tape. This allows the reconstruction program to get, before processing the first event, the run parameters, like machine energy, list of dead channels, etc, but also the list of hot channels detected by monitoring the data while they were taken. Writing the data to disk before copying to tape allowed having end of run information before the data. This was a very useful feature.

During data taking, tasks were allowed to inject pseudo-events in the data stream. This is how slow control information was injected, but the main use was to log all the LEP machine parameters in the data stream, as soon as they were received from LEP, with typically a block every minute.

7.5.2 - Database information

The online Run database was used as a seed for the Offline database, called SCANBOOK. Every night, a job scanned the run database and extracted the information, one line per run, to feed SCANBOOK. In fact only the update was sent, i.e. only the changed information with respect to the previous job. A complete update was triggered manually from time to time. The information exchanged was the run number, start and end date, file size, tape number, and online quality flags.

The Run Database was a direct access file, with the run number as index. Information on the run condition, the machine parameters, statistics and counters of the run, and status of its processing, was kept. It also contained a field to store Run Quality information for this run. A similar database existed for the Fill information.

The Data Quality database, called **CIA**, was problem-oriented, it contained the description of every problem that could affect the quality of the data. A run range was one of the main attributes of each problem report, together with a severity flag. Most problems were eventually understood as not affecting the data quality, some were only a small problem, giving a MAYBE run quality rating, while serious problems were quoted as DUCK, indicating a run unusable for physics. Problems were recorded per detector, with a possibility of common problems. A later addition was also to have problems affecting physics tool, like luminosity measurement, tracking, and energy flow. Those "physics" problems were in fact links to existing detector problems, indicating that due to this specific detector problem that tool was not usable. As many additions without re-design, the implementation of these "physics" quality flags was dirty. Re-implementation is a larger investment, but pays when the system has to run for several years.

The entries in this database were reviewed by fortnightly data quality meetings, where experts of each detector really reviewed each problem: Was it serious, is data quality affected, can it be recovered by a fix for the next reprocessing.

7.5.3 - FALCON

The first processing of the data was performed on the Online cluster. In the old days, FALCON was an independent cluster; several disks were dismounted and mounted to exchange data between the two clusters. A run was processed by sending 1/12 of the events to one of the 12 machines, and collecting the outputs together once all had finished. With the slow machines we had, this was the best way to have fast feedback, something really important at the start of the experiment. After the upgrade to AXP machines, FALCON became a subset of dedicated machines in the same cluster. The reason to have independent clusters at the beginning was to minimise bad interference, as the online system was supposed to be more unstable than a simple offline production cluster. Eventually FALCON became just a software system, using batch queues in various machines of the cluster, from a small part of the main AXP servers to the whole system when performing a winter reprocessing.

FALCON was essentially a sequencer of tasks performed on each run, some of them being run in parallel. Typical tasks were data preparation, extraction of interesting candidates (leptons), first pass processing to measure the TPC drift velocity, main processing, conversion to EPIO of the output tape, sending to various destination via the network, clean-up. The list of tasks, their sequencing and parallelism, the list of batch queues to be used and other parameters were kept in a database, together with the status of the runs being processed, until they were archived and deleted. A dedicated task was handling the reprocessing, which is mainly copying the data form a tape to a disk file, and then triggering a normal processing of the run, followed by a deletion of the copied file after successful completion.

As usual, the main problem was a proper handling of the errors. The sequence of tasks was stopped if any task failed, but some monitoring tasks were allowed to fail without stopping the processing. Of course, it was possible to retry after failure, at the failed point or from the beginning. For a reprocessing, the main difficulty is to keep track of the jobs already done, being in progress, to be done, and to restart properly after a crash or a failure. As a reprocessing can last several weeks, problems are encountered each time: Network failures, tape copy problems, job crashes, software bug in the control system. Problems were fixed on the spot, but the number of possible problems is close to infinity!

In the early days, a dedicated team was running FALCON, handling both the system and the software run inside. In the last few years, FALCON was running fully automatically. Task failures were reported to the appropriate people by mail, SMS messages on their GSM phone, or using a beeper. This was enough, as the amount of buffer space on disk allowed more than 24 hours stoppage without problem.

8 - CONCLUDING REMARKS

The ALEPH experiment ran for 12 years. During this period the online system was adapted to take into account a changing environment and new features were continuously applied in order to improve the data taking efficiency and the quality of the data collected. The efficiency of the data-taking system, as measured by 'Big Brother', was continuously monitored and the average for the year used as a yardstick for measuring the overall quality of the system. Over the last few years of data-taking average overall efficiencies (Luminosity on tape/luminosity delivered) of ~95% were typical, with the performance of the DAQ system reaching >99%.

There were several ingredients that were essential for achieving this level of performance. In order to operate an efficient system with limited manpower a very coherent approach is needed for the design of the system and its implementation. Great efforts were made to establish a cohesive group responsible for the infrastructure, both hardware and software. The composition of this team comprised a core of 5-6 people who remained together over a long period of time (1986-2000) and this was essential for providing continuity. It was also essential to establish very strong links to the subdetector groups who had the responsibility for applying the basic tools and frameworks to their special domains. In actual fact the coordinators for two of the main detectors (TPC and ECAL) also participated in the core DAQ activities and were themselves responsible for developing many of the standard online components. Although these were the largest subsystems they were perhaps the easiest to integrate into the overall system, and in fact most problems tended to occur in those sub-systems where the groups were less well integrated in the overall effort.

At the beginning, great emphasis was placed on defining the architecture of the DAQ system and on minimising the dependence on a particular technology. Over of the lifetime of the experiment major upgrades were made to both the DAQ system (replacement of Fastbus processors by VME) and the control system (G64 processors by VM20), but the architecture of the system remained essentially unchanged.

The online software comprised a large set of frameworks, templates and common tools that were used by every group for configuring hardware, collecting data, monitoring data quality, recording operational status etc. This provided a very good paradigm for communication between developers from all sub groups. It also allowed a very coherent interface to be presented to the shift crew and this eased the task of identifying problems and recovering from them. The software was organised in a standard area and was easy to locate and access by every software developer. The Software Manager devised procedures for making and announcing new releases such that everyone was well informed of the status and evolution of the whole software base.

Finally, there were meetings! The daily meeting attended by on-call coordinators ensured that representatives from each subsystem were present in the same room at the same time so the latest incidents and problems to appear over the last 24 hours could be addressed. These contacts helped to spread knowledge about the individual subsystems throughout the collaboration. In addition there was the weekly online meeting attended by the online group (6-10 people) in which the DAQ coordinator reported on that week's data taking efficiency and described each error that had occurred. The meeting was a vehicle for ensuring that each problem was followed up and the system adapted to minimise the effect of the problem recurring in the future. Some of the meetings were organised to celebrate notable achievements. These took place sometimes at Echenevex, accompanied by lots of champagne, sometimes in the local pizzeria. The importance of these particular meetings should never be underestimated for maintaining morale and for rewarding all the effort required to build, operate and maintain such a complex system.

References

[1] ALEPH collaboration, *ALEPH: A detector to study electron-positron annihilations at LEP*, **Nucl.Instrum.Methods Phys.Res., A 294 (1990) pp.127-178**

[2] P.Mato et al., *The new slow control system for the ALEPH experiment at LEP*, **Nucl.Instrum.Methods Phys.Res., A 352 (1994) pp.247-249**

APPENDIX A - FSM AND PROTOCOLS

The concept of finite state machines is used to describe the synchronised execution of transition sequences of the tasks participating in data taking. The participating tasks are synchronised after the execution of every transition between two states. One transition consists of several meta-states as described in section A.1. Several logically connected transitions form a protocol. In the ALEPH data acquisition system the following protocols were used:

- The DAQ protocol to steer the setup of the detector as described in section A.2.
- The TRIGGER protocol to start and stop collecting events as described in section A.3.
- The CHANGE, OUTPUT and CHANGE_CTRL protocols to execute run changes (section A.4).
- The CALIBRATION protocol (section A.5), which extends the DAQ protocol.

A.1 - The Implementation of the Finite State Machines

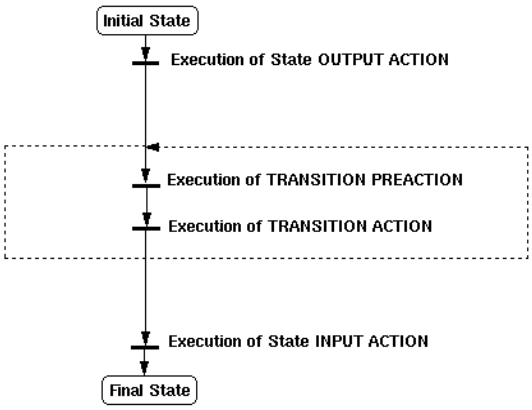


Figure 4: The actions and the state model of one transition of the Finite State Machine.

The logic used to process one transition can itself be described using the state diagram shown in Figure 4. The execution of a transition is triggered by an external stimulus, which is either a network message or an operator request. Each command received is processed as follows:

- The tables are checked to see if there is a valid transition associated with the command.
- If so, the first step in the transition is invoked, which is the state output action, a call to a user supplied action routine. This action is followed by the transition pre-action, the transition action and the state input action of the target state (see Figure 4). The return status is used to signify whether any of these actions was successful and is used by FSM to determine whether the transition should be continued or aborted. The user routine can also return a status indicating that the pre-action or the action is not yet complete. In this case processing of the transition is suspended and control returned to the application. When the task has determined that the (pre-) action has completed, it informs FSM of the result, which then resumes processing the transition. If the pre-action is ok, the rules associated with this transition are checked to see if there are any dependencies on other tasks being in a particular state. For each rule found a message is constructed and sent to the appropriate task informing it to get into the state specified in the rule. The transition sequence is interrupted in case a "Cancel" command is issued. In this case, the system remains in the state it was before processing the request started.
- Each control task waits for replies from its dependent tasks. Each time a reply is received the new state of the subsystem is checked against the required state. If the two states agree, the rule is marked as satisfied. As soon as all rules are satisfied the task executes the action routine.
- If the state-input action is completed successfully then the transition is made and the current state updated to the new state.
- In case the original command invoking the transition came from a higher level controller, a reply is constructed informing the controller of the new state of the subsystem.

Transition rules define automatic state transitions in depending FSMs which are not of the same type. For example the trigger could automatically be disabled (an action in the TRIGGER protocol) in case a DAQ error occurs (a transition in the DAQ protocol). In practice, this mechanism only works if the machines are not at the same level in the hierarchy. Otherwise, an additional hierarchical level would have to be introduced with the corresponding rules. If the FSMs are at the same hierarchical level as the run controller, the steering must be done in code.

Although the FSM machinery itself was implemented in C++, the user callbacks were implemented in several language bindings. Bindings were implemented for user code in C++, C and Fortran. Fortran and C are needed to avoid changes to existing tasks, which were implemented mostly in Fortran on VMS and in C on the readout processors. A mechanism to dynamically connect the callbacks at run-time allowed the implementation within a shared library.

A.2 - The Setup (DAQ) Protocol

The setup of the hardware and the resource allocation is done during the preparation phase for the data acquisition. Several steps are required as shown in the transition diagram in Figure 5. When a task is created the initial state of this protocol is the "Inactive" state. The setup procedure starts executing the "Config" transition. During this transition, the run controller creates all tasks participating in the readout. This takes of the order of a couple of minutes during a cold-start, but is considerably faster when all participating tasks already exist. The start-run sequence proceeds through the "Initialise" and "Go" transitions until the *Active* state is reached. Calibration constants are read at the start of each run and loaded into the readout processors during the "Config" transition where they are picked up by the tasks running in the readout processors during the "Initialise" transition. The "Go" transition can be used to complete any initialisation of the readout, since once the system has reached the "Active" state it is assumed that each component is ready to receive triggers.

The stop-run sequence proceeds via the "Pause", "Stop", "Close" and "Finish" transitions, which are three steps analogous to the three steps of the start-up sequence. The "Finish" transition is used by the output task to complete the data file on the output medium i.e. by adding end of run records. Once the system reaches the "Closed" state, all sub-detector end-run activities have been completed.

There is a special transition from each state back to the "Inactive" state called "Reset" to allow a fast reset of the entire system into a well-defined state. This transition may not fail.

In the case, that any task involved in the dataflow (i.e. producers and consumers in the readout processors or data monitoring tasks) detects an error, the task invokes the transition to the "Error" state. The run controller will automatically disable the trigger when executing this transition. The operator can then ignore the error to resume data taking, abort data taking using the "Reset" transition to the inactive state or try to recover. Typically, all recovery procedures flush the event pipeline ("Flush" transitions) before data taking continues to ensure that event fragments corresponding to the same event are merged in the event builders.

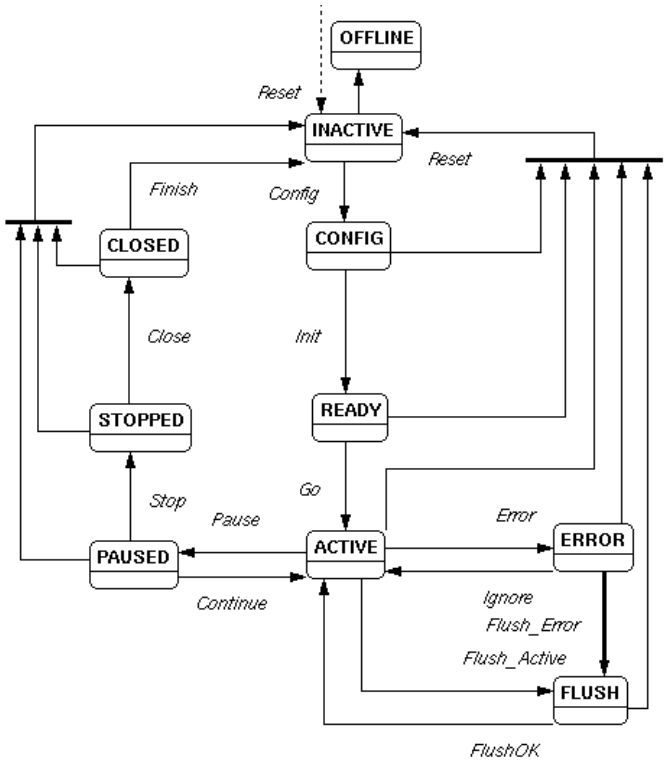


Figure 5: The state diagram of the DAQ protocol used for the setup of the various detector components.

A.3 - The Trigger Protocol

Transitions in the trigger protocol can only be executed once all tasks participating in data taking, which implement the DAQ protocol, are in the "Active" state and are ready to receive triggers. Otherwise, an attempt to enable the trigger would fail, because the hardware necessary to read the detector data is not configured according to the requested setup. Initially in the "Inactive" state, "TriggerEnable" causes the trigger supervisor to enable the trigger and sets the system into the "Running" state. From here, the trigger can be disabled by the operator ("TriggerDisable") or an exception of the trigger protocol can force the run into the "Error" state. The "Error" state of the DAQ protocol is distinct from the "Error" state of the TRIGGER protocol. Errors can be ignored ("TriggerIgnore") or the operator can choose to force the trigger machine to the "Inactive" state ("Reset"). As in the DAQ protocol, this transition may not fail. This protocol is implemented in all tasks which either control the trigger hardware directly or control these tasks, such as the run controller.

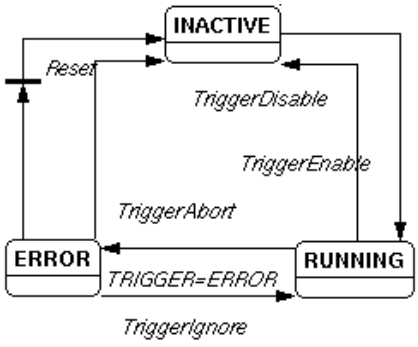


Figure 6: The states of the TRIGGER protocol

A.4 - The Protocols used for Run Changes

During a run change, data taking continues and thus luminosity is not wasted. Run changes involve various actions to be performed, such as saving the detector conditions, initialising statistics, initialising run-

specific histograms, etc. Several tasks that are involved in a run change act together through the synchronisation of the FSM using the protocols in the sequence described in the following. Subdetector tasks run the CHANGE protocol, the task responsible for writing data to disk implements the OUTPUT protocol and all are sequenced by the run controller implementing the CHANGE_CTRL protocol. The CHANGE_CTRL protocol implements the rules necessary to steer the CHANGE and the OUTPUT protocol by applying the appropriate rules.

The sequence of a run change is as follows:

- The run controller executes the transition to the "Prepare" state. This forces the disk writer task into the "Changing" state. From then on, all events to be written to disk will correspond to a new run. The run number has already been updated by the run controller and exclusively end of run records will be accepted for the old run.
- The run controller executes the transition to the "Change_Run" state. During this transition, tasks e.g. prepare end of run records and insert them into the data stream, where they are to be picked up by the disk writer task.
- The run controller executes the transition to the "Output" state. Tasks that implement the CHANGE protocol return to normal running conditions i.e. to the Inactive state. During the transition they are able to produce the start of run records for the new run, inject them into the data stream, re-initialise histograms, statistics, etc.
- The run controller executes the transition to the "Inactive" state. The disk writer task implementing the OUTPUT protocol will return into the "Inactive" state as well and during this transition close the previous run. When executing this transition, the output task initiates the hand-over of the data files to the online reconstruction facility FALCON and the bookkeeping task produces the end of run summary. The run change is finished.

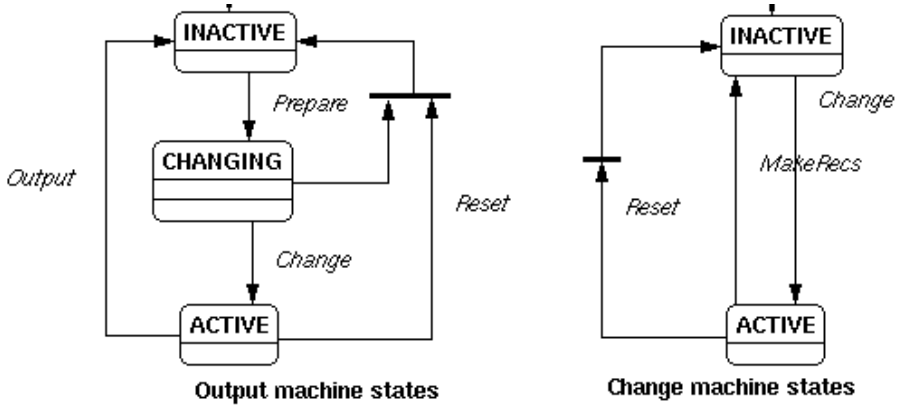


Figure 7: The OUTPUT and CHANGE protocols.

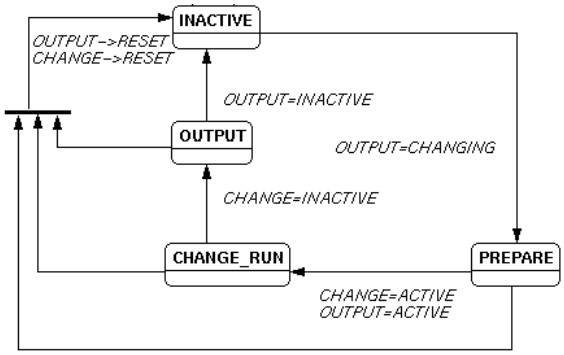


Figure 8: The CHANGE_CTRL protocol implemented by the run controller to steer the OUTPUT and the CHANGE protocol of the dependent tasks.

A.5 - The Calibration Protocol

When the detector requires calibration, the participating tasks execute a sequence similar to the one executed during normal data taking. In addition, two transitions are executed as part of the "Calibration"

protocol, before and after collecting a specified number of events. The exact number of events to be collected depends on the requirements of the calibration. The additional transitions permit the VMS tasks to first download additional information needed by the readout processors and the readout processors to coherently pick up these parameters before using them when processing event data. The same procedure is executed in reverse after the events are collected. First the parameters are processed in the readout processors and then picked up by the VMS tasks for further processing and storing. This leads to two additional transitions before and after events are collected for calibration purposes.