

THE NEW IMPLEMENTATION OF THE EVENT BUILDING PROTOCOL FOR THE ALEPH DATA ACQUISITION SYSTEM

J.A. Perlas, J. Harvey, B. Jost and P. Mato
CERN, CH-1211 Geneva 23, Switzerland

PHYSIC 93-170
DATACQ 93-01
J. A. Perlas et al.
2.11.1993

Abstract

The flow of data through the ALEPH readout system is effected under the control of a special protocol between the various readout stages. The software library used to implement this protocol has been re-engineered using the Finite State Machine modelling technique. The state of each system component can be observed at any given time and this has greatly simplified the detection of, and recovery from, protocol errors. The model and its implementation are described together with their integration with the applications which are used by the operator to control data taking. Operational experience gained with this protocol implementation during the whole 1992 running period will also be given to show the advantages of this design.

I. INTRODUCTION

The structure of the ALEPH detector and the Fastbus system are described elsewhere [1]. Basically, the experiment is subdivided in several detector components, each one having its own readout system. In the first stage of the readout system special controllers are used for each detector component, each controller being adapted to the particular processing needs of their detector. In the second stage data from all the controllers belonging to a particular detector are assembled and in the third stage data from all the detectors are combined to form a full event. Thus the structure of the readout system is "tree-like" [2].

The ALEPH Data Acquisition (DAQ) system uses a data transfer protocol between processing elements in the *event-building* stages of the readout, in order to ensure that the data are collected and assembled correctly [3]. A special *readout library* was written in order to ensure that this protocol was obeyed.

In this paper we describe the functional specifications of this data transfer protocol and its implementation using the Fastbus standard. We then explain our experience with the original implementation of the readout library and the motivation for re-engineering it using a Finite State Machine model. This new version is described and our experience using it during one year of data taking is presented.

II. THE DATA TRANSFER PROTOCOL

The readout function has been separated into independent tasks running asynchronously in order to de-randomize the flow of events (see Fig. 1). The receiver task is responsible for reading portions of an event from the previous stage and putting this data into an event buffer. As soon as the read operation has finished and the event is declared in the buffer, the receiver is ready to receive the next one. The sender task is activated each time an event is declared in the buffer. It is responsible for releasing the space occupied by the event as soon as it has been transferred to the next stage in the readout. Thus, providing there is always sufficient space in the buffer to accommodate at least one event, the two activities of reading in the data and sending it onwards to the next stage can proceed asynchronously.

The data transfer protocol from one stage to the next is as follows:

- The receiver allocates buffer space and waits for a readout *request*.
- The sender asserts a request when it has data available.
- The receiver responds to requests until all its sources have been read.
- Finally, the sender releases its buffer and gets ready for the next event.

The implementation of the protocol uses features of the Fastbus standard, namely the Service Request mechanism and the associated Control and Status Registers. A special Service Request Handler implemented in software guarantees the correct distribution and handling of the various readout requests from the different sources.

The *readout library* ensures that the protocol is strictly obeyed and detects any violations that occur during event collection.

III. MOTIVATION FOR A NEW IMPLEMENTATION OF THE PROTOCOL

Experience during the first two years of running showed that there were some problems with the original implementation of this library. During execution of the protocol it was not possible to receive any other external stimulus such as a control message. In the case of protocol errors this could lead to the task becoming blocked and recovery was possible only by killing and restarting it. Hence no controlled error recovery was possible.

In addition, on protocol errors it was very difficult to trace what had happened since the state of all components was not clearly defined. In particular, it was difficult to see whether the problem was caused by the failure of a hardware component or whether it was due to a logical flaw in the implementation of the protocol itself. For this reason we wanted to have the possibility of *freezing* data collection upon detection of protocol violations to facilitate the diagnosis of errors.

IV. DESCRIPTION OF THE IMPLEMENTATION

During the last year, a new implementation of the read-out protocol was developed. This uses the *Finite State Machine* (FSM) modelling technique [4] to describe the state of each task during execution of the protocol.

Some of the benefits we expect to obtain from using this approach are as follows:

- It offers a good representation of the dynamic aspects of the protocol.
- It is very well adapted for systems that exhibit an *asynchronous* behaviour.
- Coordination between processes can also be modelled.
- It allows the model to be changed very easily.
- It produces more easy-to-maintain source code.

In order to implement FSM models, a general-purpose library has been written. This library has been used in both the receiver and the sender. The FSM diagram for the receiver is shown in Fig. 2, and that for the sender in Fig. 3.

There are three types of stimuli that can cause a transition to be invoked. The first is an internal detection of change of state e.g. when an event is assembled or when an error occurs in the case of the receiver. The second type results from inter-processor interrupts which implement the protocol itself. Finally, there are external control messages coming from the ALEPH Run Control task which allow the operator to control data taking.

In order to handle these asynchronous inputs it has been essential that all inputs are treated in a uniform way, i.e. in a single *queue*. This allows us, for example, to treat control messages during the event building phase and therefore

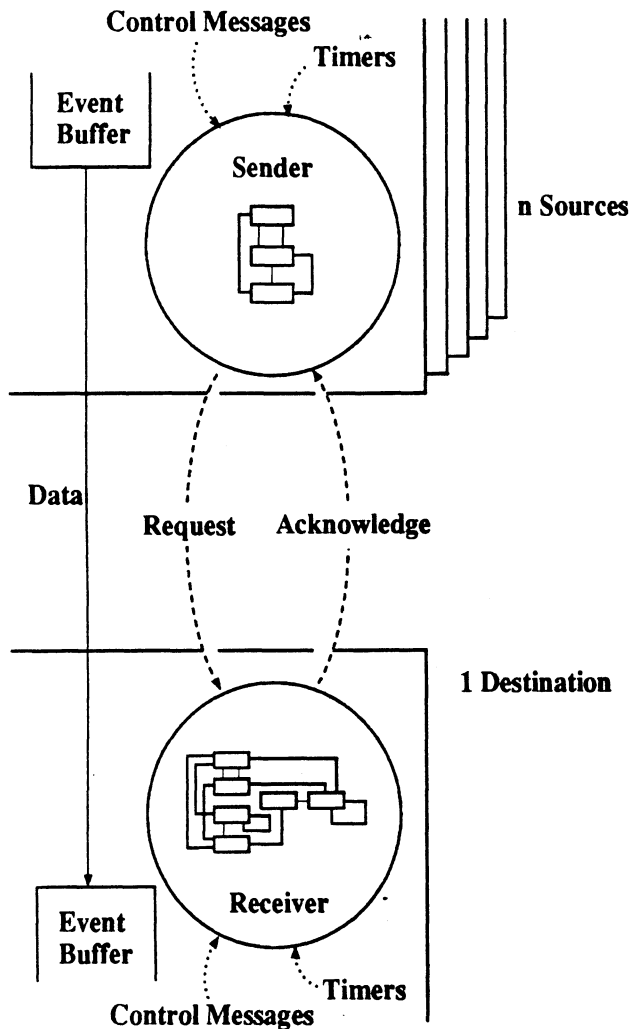


Figure 1: Architecture of the software components implementing the event building protocol.

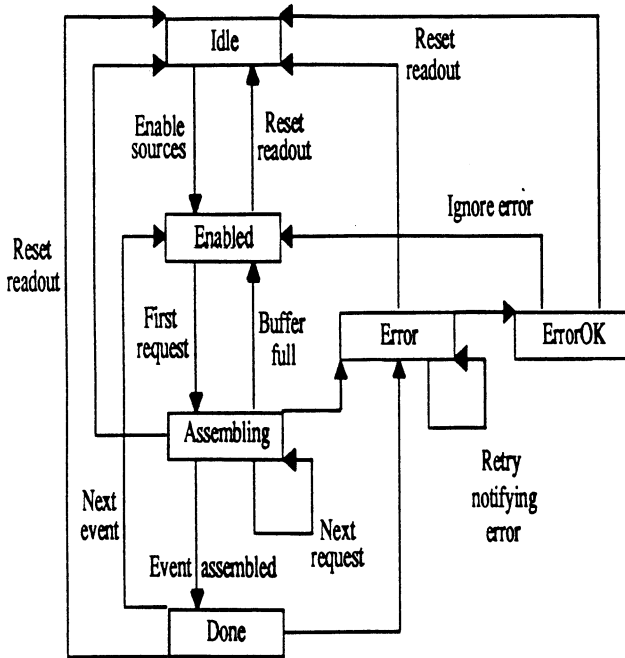


Figure 2: FSM diagram for the receiver in the ALEPH readout protocol.

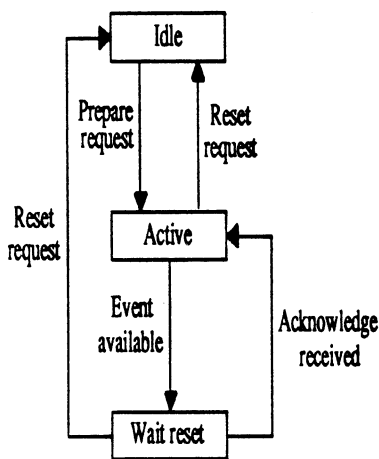


Figure 3: FSM diagram for the sender in the ALEPH readout protocol.

gives the possibility of aborting the readout of the current event.

Apart from re-engineering the readout library, a better error recovery needed two more additions to our system. The first was to build a *monitoring tool*, which can be run at every readout node, displaying the FSM states and several useful counters for both the receiver and sender of this readout element. This has proved to be very useful while debugging the new software and also in cases where we found a readout protocol problem.

Secondly, we changed the FSM model of the ALEPH run controller in order to implement a feature that allows the complete data pipeline to first be "frozen" in order to analyse the state of the complete system and then to be emptied. This permitted the system to be set into a well-defined state from which recovery could be guaranteed.

V. EXPERIENCE DURING RUNNING

The new library was used during the 1992 running period and a significant improvement in the error diagnosis and recovery procedure has been observed. In particular, the recovery procedure has been found to take approximately 10 seconds and thus has speeded-up considerably.

In addition, once this had been demonstrated to work reliably, the procedure was included in the ALEPH Expert System rule base such that errors could be treated automatically, i.e. without intervention from the operator.

The ability to identify the state of all tasks executing the protocol greatly facilitated monitoring and debugging during the commissioning phase. We believe that this approach of implementing complex protocols using state models has significant advantages which have been demonstrated in a realistic example, through practical experience and over an extensive period.

Errors coming from flaws in the implementation of the protocol in the various readout processors in the ALEPH system were gradually identified and eradicated. By the end of the commissioning period, the data taking efficiency was approximately 98 %, which is higher than that of the previous years. The remaining inefficiencies are mainly due to hardware failures and general software problems.

VI. REFERENCES

- [1] W. von Räden, "The ALEPH data acquisition system", IEEE Trans. on Nucl. Science Vol. 36, no. 5, 1444-1448 (1989).
- [2] A. Belk et al., "DAQ software architecture for ALEPH, a large HEP experiment", IEEE Trans. on Nucl. Science Vol. 36, no. 5, 1534-1539 (1989).
- [3] ALEPH Dataflow Group, "ALEPH Data Acquisition System Hardware Functional Specifications", ALEPH DATAQ note 85-21, CERN (1985).
- [4] J.L. Peterson, "Petri Nets", Computing Surveys Vol. 9, no. 3, 223-252 (1977).