EUROPEAN ORGANIZATION FOR NUCLEAR RESEARCH

CERN–EP/2000–121
31 March 2000

The Data Acquisition System of the CHORUS Experiment

A. Artamonov[1,a)], D. Bonekämper[2)], J. Brunner[3,b)], A. Bülte[4)], G. Carnevale[3)], M.G. Catanesi[5)], A. Cocco[6)], D. Cussans[3,c)], R. Ferreira[3,d)], B. Friend[3)], P. Gorbunov[1)], A. Guerriero[3)], R. Gurin[3,e)], M. de Jong[7)], M. Litmaath[3,f,*)], D. Macina[3,g)], A. Maslennikov[8,e)], M.A. Mazzoni[8)], R. Meijer Drees[3,h)], H. Meinhard[3)], C. Mommaert[9,i)], R.G.C. Oldeman[7,j)], H. Øverås[3)], J. Panman[3)], C.A.F.J. van der Poel[7)], F. Riccardi[3)], D. Rondeshagen[2)], A. Rozanov[3,k)], D. Saltzberg[3,l)], J.W.E. Uiterwijk[7)], M. Vander Donckt[9,m)], T. Wolff[2,n)], H. Wong[3,o)], P. Zucchelli[10,p)]

Abstract

In the years 1994–1998 the CHORUS Collaboration has recorded data in the CERN WA95 experiment. Here we describe the data acquisition system that has been used, featuring concurrent hierarchical state machines, a remote operating system, a buffer manager, a dispatcher, a control panel and a supervisor.

(To be submitted to NIM.)

[1)] Institute for Theoretical and Experimental Physics, Moscow, Russian Federation.
[2)] Westfälische Wilhelms-Universität, Münster, Germany.
[3)] CERN, Geneva, Switzerland.
[4)] Humboldt Universität, Berlin, Germany.
[5)] Università di Bari and INFN, Bari, Italy.
[6)] Università Federico II and INFN, Naples, Italy.
[7)] NIKHEF, Amsterdam, The Netherlands.
[8)] Università La Sapienza and INFN, Rome, Italy.
[9)] Inter-University Institute for High Energies (ULB-VUB), Brussels, Belgium.
[10)] Università di Ferrara and INFN, Ferrara, Italy.
[a)] Now at CERN, CH–1211 Geneva 23, Switzerland.
[b)] Now at CPPM CNRS-IN2P3, Marseille, France.
[c)] Now at University of Bristol, Bristol, UK.
[d)] Now at IBM, 135 chaussée de Bruxelles, B–1310 La Hulpe, Belgium.
[e)] CASPUR, c/o Università La Sapienza, 5 Piazzale Aldo Moro, 00185 Rome, Italy.
[f)] Now at Fermilab, P.O. Box 500 - MS 369, Batavia, IL–60510, USA.
[g)] Now at Université de Genève, Geneva, Switzerland.
[h)] Now at 8140 Lakeview Drive, Burnaby, BC, Canada.
[i)] Interuniversitair Instituut voor Kernwetenschappen.
[j)] Now at University of Pennsylvania, Philadelphia, PA 19104-6396, USA.
[k)] Now at Université de Marseille, Marseille, France.
[l)] Now at U.C.L.A., Los Angeles, USA.
[m)] Fonds pour la Formation à la Recherche dans l'Industrie et dans l'Agriculture.
[n)] Supported by a grant from Deutsche Forschungsgemeinschaft.
[o)] Now at Academia Sinica, Taipei, Taiwan.
[p)] On leave of absence from INFN, Ferrara, Italy.
[*)] Corresponding author: litmaath@cern.ch.

1 Introduction

In the years 1994–1998 the CHORUS Collaboration has recorded data in the CERN WA95 experiment [1], designed principally to search for evidence of the $\nu_\mu \rightarrow \nu_\tau$ oscillation phenomenon in the CERN SPS wide-band ν_μ beam.

A significant ν_τ component in the beam should reveal itself by leaving tracks of τ particles, arising from charged current (CC) interactions, in a 770 kg nuclear emulsion target located in the experimental setup. The τ particles are expected to traverse on average less than 1 mm before they decay into secondary particles, which may be detected by the electronic detectors downstream of the target. These detectors serve to identify and record events with signatures that are compatible with CC ν_τ interactions in particular. In the analysis stage the tracks of the outgoing particles are reconstructed from the recorded detector signals. The location of the neutrino interaction vertex is then determined by extrapolating and following the interesting tracks back into the emulsion, using automatic scanning microscope systems [2]. Evidence for a ν_τ interaction is obtained when the relevant tracks are found to have the correct topology nearby the vertex. For example, the signature of a decay $\tau \rightarrow \mu\bar{\nu}_\mu\nu_\tau$ is the presence of a kink between the short τ track and the outgoing μ track.

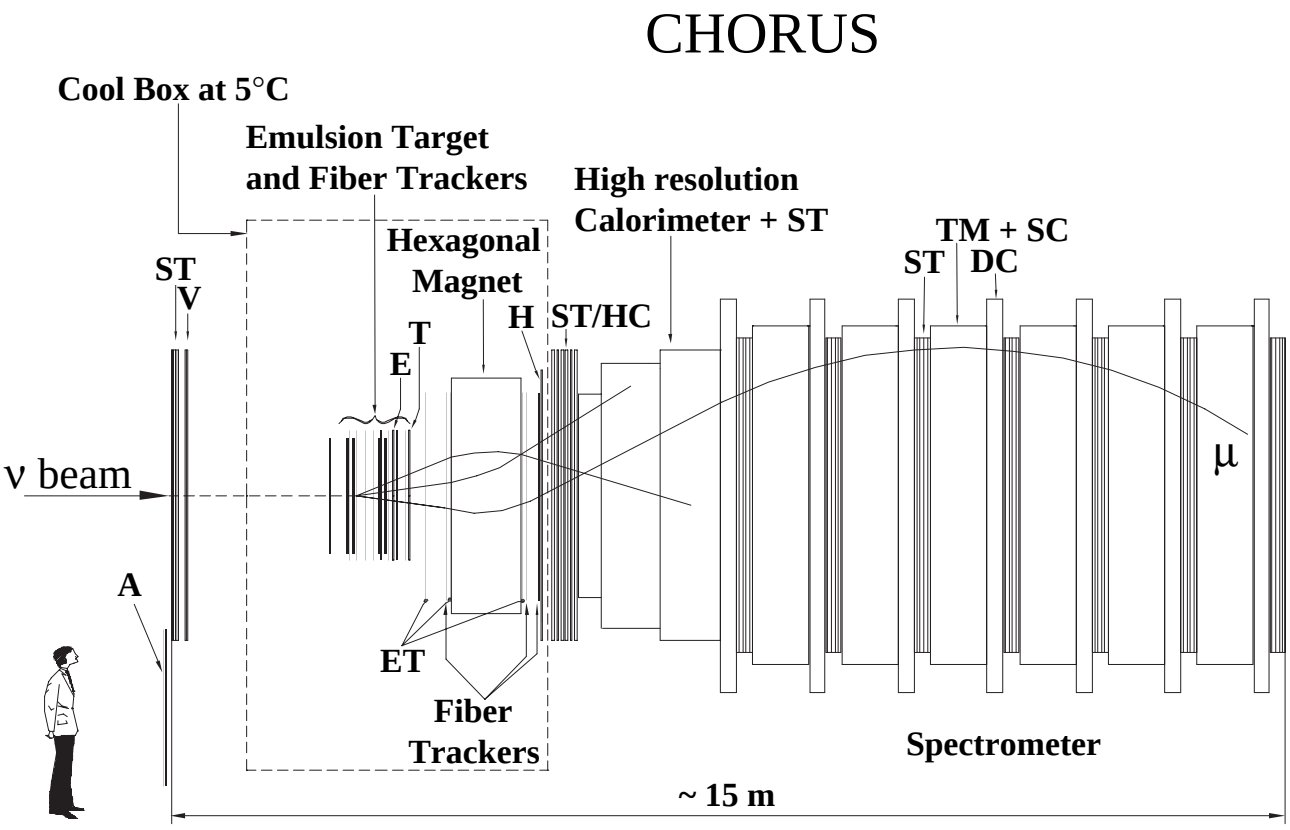

Figure 1: Schematic view of the CHORUS experiment. The meaning of the acronyms: A = anticounter (extra veto), DC = drift chambers, E = emulsion trigger plane, ET = emulsion tracker plates, H = horizontal trigger plane, HC = honeycomb subdetector, SC = scintillator layers, ST = streamer tube planes, T = target trigger plane, TM = toroidal magnet, V = veto.

Experimental Setup

The layout of the CHORUS experiment is shown in Fig. 1. The detector consists of various subdetectors, most of which are located downstream of the emulsion target. The target consists of four stacks, interspersed with and followed by planes of optical fibers for high precision tracking of charged particles. These fibers are connected in groups to CCD cameras, that record the signals of all channels whenever a trigger signal indicates that an interesting event just occurred.

Further downstream more fiber optics planes are located in front of and behind the hexagonal magnet, followed by a set of honeycomb wire chambers (originally by streamer tube planes). This combined system serves the measurement of the momenta of charged particles. The total number of optical fibers exceeds one million.

Next comes the calorimeter, which is used to determine the total energies and the directions of hadronic showers created by neutrino interactions in the emulsion target.

Downstream of the calorimeter the spectrometer serves to identify muons and measure their momenta with high precision.

The trigger system uses scintillation counters throughout the experiment to signal the occurrence of interesting events in the target region, in the calorimeter or in the spectrometer. A trigger signal causes each relevant subdetector to record its signals.

Functionality of the Data Acquisition System

In general the data acquisition (DAQ) system has to perform the following tasks:

- synchronization of the data-taking with the SPS cycle,
- readout of the data stored in the detector front-end electronics,
- event building, i.e. combining for each event the data recorded by all the relevant subdetectors,
- validation of the data,
- storage of the data,
- monitoring, logging and
- interfacing with the user.

Further requirements have been imposed on the DAQ, to facilitate its implementation, operation and maintenance:

- The system should be hierarchical, to allow a staged implementation, independent tests and replacements of subsections.
- Where possible the functionality should be implemented on a generic (i.e. replaceable) UNIX derivative.
- The back-end of the readout hardware should be based on the VME industry standard.
- The system should make use of object-oriented programming and graphical user interface tools.

The requirements have led to the following features:

- High-level interfacing with the user, data monitoring, logging and storage are primarily implemented on the DAQ **back-end** consisting of UNIX workstations and servers. The **front-end** readout concepts are implemented on top of the OS-9 [4] real-time operating system running on VME boards. A **publish–subscribe** server process is used for communication between the two domains.

- The front-end DAQ is organized hierarchically, such that a top-level **Event Builder** (EVB) system is in control of all **subsystems**, each of which integrates a group of (related) subdetectors. This is the normal mode of operation, called **global** mode. The subsystems are connected to the EVB through a global VICbus [7].
- Each subsystem is also able to run a **stand-alone** version of the DAQ, such that it is independent of the EVB and of the other subsystems. This mode is used for calibrations and tests of the subdetectors. In stand-alone mode each subsystem sends its events to its own virtual EVB: in this case the EVB processes are running on the CPU of the subsystem itself. The distinction between the two modes is kept minimal: the very same programs are run, with different configuration files.
- In global mode each subsystem is still able to take its own **local** events as soon as it has finished processing the global events taken in the preceding burst. Local events are inhibited shortly before the next burst.
- The real-time operating system has been extended with libraries that provide communication between a subsystem and the EVB through a **buffer manager** on top of **remote shared memory**.
- The readout in each subsystem is also organized hierarchically, with a VME crate at the top (back-end) and mostly CAMAC (some VME) crates at the front-end, all connected through a local VICbus.
- The instantaneous global data rate to be handled may be as high as 4 Mbyte/s.
- The front-end software is written in the **object-oriented** C++ programming language wherever feasible, and makes use of an object-oriented **finite state machine** framework. The back-end software is written in Tcl/Tk [17], Expect [19], C++, C, Fortran-77, and shell scripts.
- The DAQ is steered from a graphical **Control Panel**, which interfaces with a **Supervisor** server process ultimately in control of the data-taking.

2 Hardware Architecture and Operation

The CHORUS subdetectors are organized into four DAQ partitions, referred to as subsystems: global trigger (TRIG), opto-electronics (OPTO), calorimeter (CALO) and muon spectrometer (SPEC). For global data-taking all subsystems together form a single DAQ system, controlled by the EVB, which collects and combines the data from the subsystems into complete events and sends these to the UNIX back-end via a publish–subscribe server process (see Sect. 9). Each subsystem can also run in stand-alone mode and record its own (local) data, being completely independent of the EVB and of the other subsystems.

The front-end of the DAQ system communicates with VME modules in real-time mode. In total there are 35 intelligent VME processors in the subsystems, all running low-level DAQ programs on top of the Microware OS-9 [4] real-time kernel. The EVB and all subsystems can communicate with the UNIX machines via the Ethernet. Fig. 2 shows the DAQ network topology. The VSB+VME link is discussed in Sect. 9. The CAENET link is discussed in Sect. 10.

In each subsystem the readout modules are located in VME and CAMAC crates interconnected by a local VICbus [7], whose bandwidth of 10 Mbyte/s is sufficient to handle the data rate. The most stringent requirement was set by the OPTO subsystem [3] (see Sect. 3). Two global VICbus branches connect all subsystems to the EVB. In stand-alone mode the subsystem's global VICbus connection is switched off-line. Fig. 3 shows the intelligent VME crates and their VICbus interconnections.

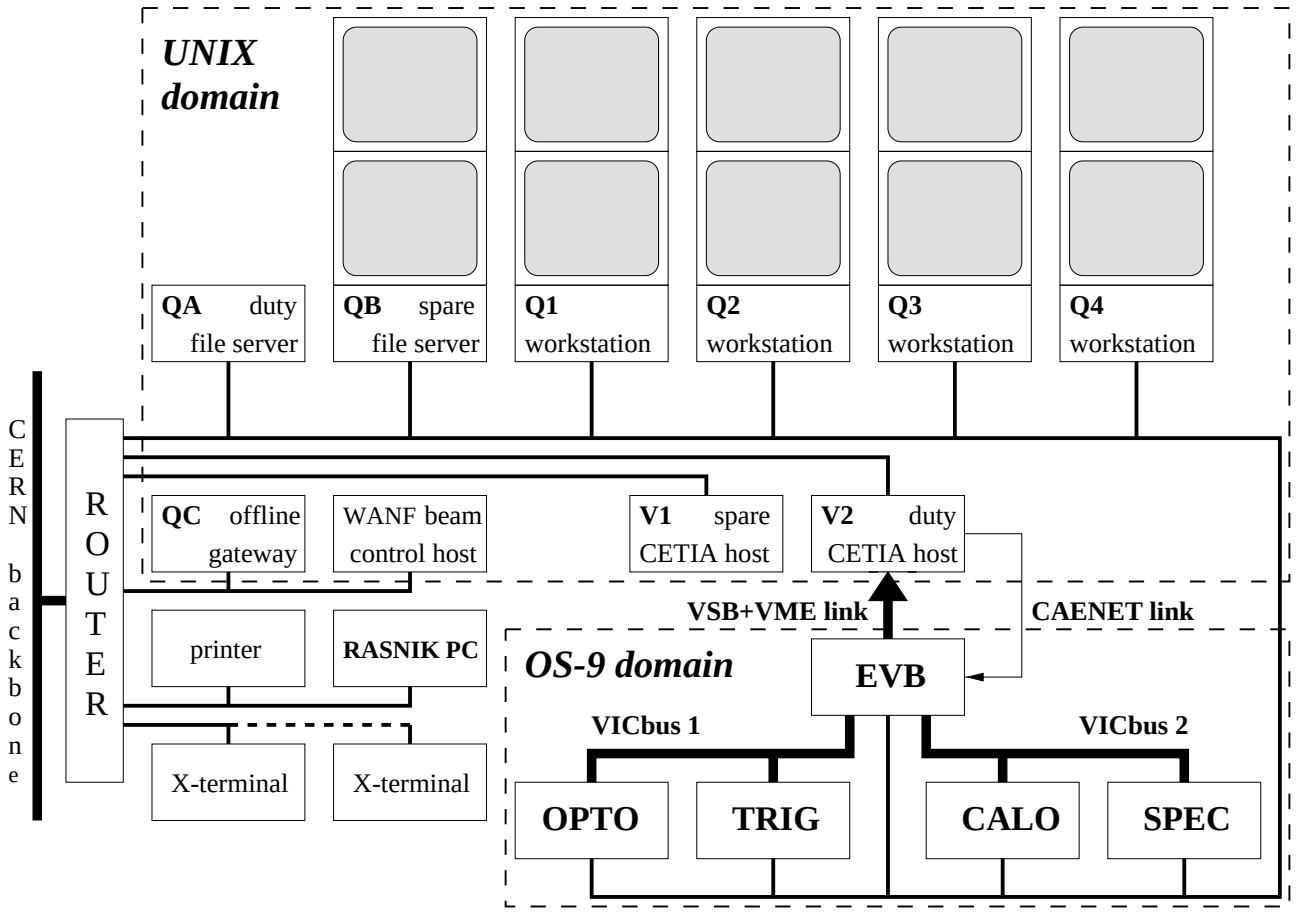

Figure 2: The CHORUS DAQ network. Each of the six thin lines connected to the router represents a 10 Mbps Ethernet segment. The CERN backbone is FDDI.

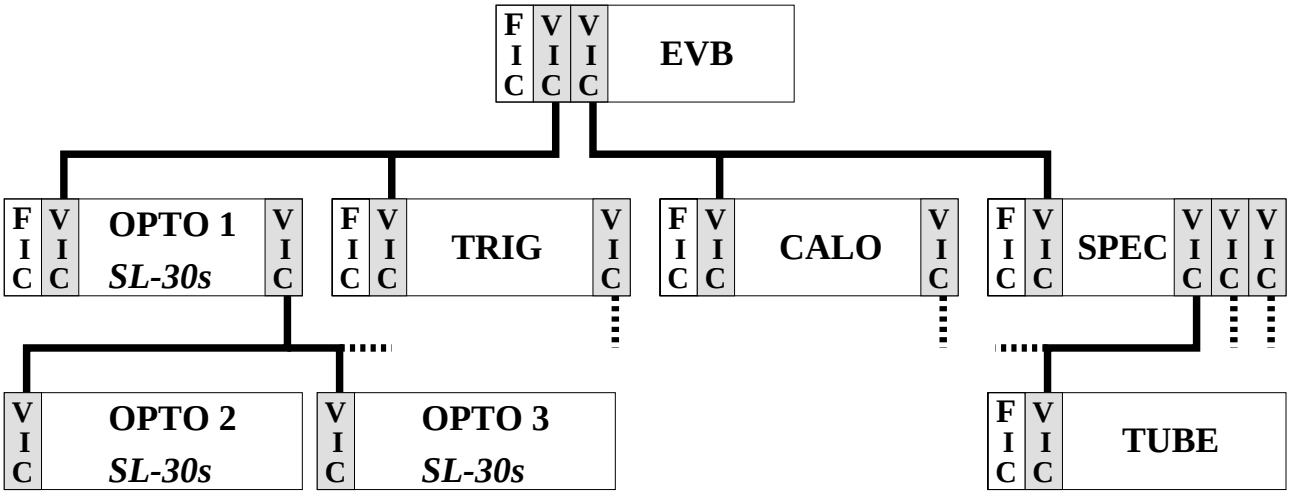

Figure 3: The intelligent VME crates in the CHORUS DAQ and their VICbus interconnections. The dashed lines represent VICbus connections to the dumb crates: 43 CAMAC and two more VME. The second FIC-8234 in the SPEC subsystem is called TUBE, because it handles the readout of streamer tube detectors.

The high-level back-end of the DAQ runs in a standard UNIX environment, for which IBM AIX 3.2–4.1 has been chosen, running on seven IBM RS-6000 PowerPC workstations ("QA"…"QC" and "Q1"…"Q4") and two CETIA [5] PowerEngine VME boards ("V1" and "V2"). The back-end is responsible for interfacing with the user, data monitoring, logging and storage.

All but one of the IBM hosts and all of the OS-9 hosts are on a single Ethernet segment connected to one port of a router, which primarily serves to shield the DAQ hosts from the rest of the CERN network. The remaining one IBM host ("QC") is on a separate port. It is used as a gateway between the DAQ machines and the CHORUS computing facilities in the computer center. It has access to both worlds and is not vital for running the DAQ. Its Ethernet segment is also used by a workstation ("WANF") for controlling and monitoring the neutrino beam line (see Sect. 11). Each of the two CETIA hosts has a private segment to the router (see Sect. 9). Another port is used for NCD X-terminals, used as desktops by the shift crew and by subsystem experts. A sixth and last port is for a printer and for a PC ("RASNIK") used for recording data from an optical alignment system (see Sect. 11).

The router was introduced in January 1996. Before that time the online hosts were only shielded to a limited extent by a bridge. The DAQ used to be frequently disturbed by broadcasts from the other side. Furthermore, all the online hosts were on a single Ethernet segment, which was barely able to handle the traffic. The router has been quite important in improving the stability of the DAQ.

Front-end Operation

In global mode the data-taking follows the pattern of the SPS neutrino burst cycle. In each cycle of 14.4 seconds protons are extracted from the SPS and steered onto the neutrino production target in two bursts lasting 6 ms each, separated by 2.7 seconds. During each burst on average five global events are recorded in the electronics modules of the subsystems, but the OPTO subsystem is typically involved in only one event per cycle. Immediately after the burst the TRIG subsystem sends the list of **event descriptors** to the EVB. For each recorded event the descriptor indicates which subsystems were involved and should now be contacted to deliver their data.

For calibration purposes the detector is occasionally (up to four weeks per year) exposed to π, μ and e beams. For this operation an extra global gate is enabled between the two neutrino bursts. The readout of the first burst is then delayed until after this gate or after the second burst. Such a delay normally is unacceptable, because the OPTO electronics can buffer only two events [3], whereas depending on the selectivity of the trigger (and on the beam intensity) there may be a non-negligible probability for any cycle to produce three or four events that may have a vertex in the emulsion. In the end that probability turned out to be only 7 %. This is due to the excellent performance of the trigger [10], which could not be taken for granted when the DAQ was being designed.

During normal data-taking the period between the two bursts and the rest of the cycle ($\sim$12 s) are used for local events: each subsystem is allowed to take such events as soon as it is ready with the global readout. Such events arise from cosmic rays traversing the detector and from test pulses sent to the front-end electronics. Local events serve primarily for monitoring the detector performance and for calibration purposes. These events are not buffered: as soon as a local trigger occurs, the data are read out by the subsystem and sent to the EVB to be mixed into the global data stream.

3 The OS-9 Real-Time Front-End

The task of the OS-9 section of the DAQ is to collect the event data from the electronics modules in the participating subsystems, assemble the portions into complete events, and send those towards the UNIX back-end, all in synchronization with the SPS burst cycle. Each subsystem runs a standard set of DAQ processes (described below) on its own VME processor board, a CES FIC-8234 [7] with one or two Motorola 25 MHz 68040 CPUs and a boot EEPROM that loads the OS-9 real-time kernel and standard server programs from the UNIX file server via the Ethernet. An OS-9 file system is mounted from the same server through NFS.

The complete DAQ is controlled by a group of EVB processes. In global mode these processes run on their own dedicated FIC-8234 and collect the data from the TRIG subsystem and any other participating subsystems. In stand-alone mode the processes run directly on the subsystem FIC-8234 and only collect the data from the subsystem in question.

The subsystem DAQ comprises a main process, a trigger process, optionally a data readout process and optionally a histogram process. The main process controls the other processes and in the absence of a data readout process it also deals with reading out all the relevant electronics modules at the appropriate times. Global events are read out after the neutrino burst, when the main process has received the list of global event descriptors from the EVB. A local event is read out when the main process receives a (single) local event descriptor from the trigger process. All events are prepared in the memory of the subsystem's global VICbus connection module, a CES VIC-8251 [7]. The EVB is then notified and reads the prepared data, through the VICbus in a global DAQ, through VME in a stand-alone DAQ.

The OPTO subsystem [3] has the most complicated DAQ of the four subsystems. In particular it comprises 29 ELTEC SL-30 [8] image processor VME boards (see Fig. 3), all running local front-end processes on OS-9. These modules can buffer only two events, but even after zero-suppression their combined data for one burst may add up to 4 Mbytes. When the DAQ was being designed it was not evident that the OPTO main program would always be able to deal with such an amount of data in the limited time between the two bursts of a cycle. However, as soon as the data of the first burst have been copied from an SL-30 video buffer to its main memory, it is ready for new events. The EVB does not need to receive the actual events immediately. Therefore the OPTO main process delegates the readout to the OPTO read process and only waits until all SL-30s have signaled that they can receive new events. The readout process is also responsible for loading OS-9 and the necessary programs into each SL-30 through the local OPTO VICbus.

The monitoring of event data distributions can be performed directly on the FIC-8234 of each subsystem by a local histogram process, which is also under control of the main process. It accesses the event data through shared memory. In a later stage of the experiment this functionality was fully moved to the UNIX back-end, because of negative interference with the rest of the DAQ processes on OS-9 (see Sect. 8).

A subsystem DAQ comprises multiple processes that need to communicate with each other and with the EVB. Each process uses a buffer manager C++ class to establish communication through message queues and semaphores (see Sect. 5). In a stand-alone DAQ all processes are running on the same CPU and the buffer manager only needs to make use of OS-9 system functions. In the global DAQ the EVB runs on its own FIC-8234, connected to each subsystem via the VICbus. For this case a new system service has been developed, the Remote Operating System (REMOS). It provides system functions allowing processes running on different FIC-8234 modules to communicate through the VICbus. This is described in the next section.

4 The Remote Operating System (REMOS)

REMOS [9] provides primitives for remote OS-9 processes to communicate through VME, via a VICbus when they run on CPUs in different crates. In REMOS the VICbus is referred to as **bridge**.

The VICbus allows parts of the local VME addressing space to be mapped onto similar or different parts of the VME addressing spaces in remote crates. This provides a local VME master with access to the backplane of a remote crate, such that it can read from and write to remote modules as if they were placed in the local crate. Not only can data thus be transferred between crates, but also interrupts may be generated in a remote crate. When a flag is set in a *mailbox* register of a VIC-8251, then that module will generate either a VME interrupt in its own crate, or a VICbus interrupt in a remote VIC-8251, depending on the register's configuration. The remote VIC-8251 is configured to generate a VME interrupt for each incoming VICbus interrupt. In either case we end up with an ordinary VME interrupt in the destination crate.

A local VME master may thus signal a remote master that data are present in some well-known location. A VME *slave* cannot transfer any data itself, but it is able to generate a VME interrupt, handled by the local VME master. The notions of master and slave are also found on the VICbus. On both global VICbus branches the EVB is the master and the subsystems are the slaves (see Fig. 3).

A basis for bidirectional communication over a VICbus is now established:

- The EVB FIC-8234 may put a message into a reserved portion of the memory of the subsystem VIC-8251 slave module, and then set a flag in a mailbox register in that module. The slave then generates a VME interrupt, handled by the subsystem FIC-8234.
- A subsystem FIC-8234 may put a message into another reserved portion of the memory of its VIC-8251 slave module and set a flag in another mailbox register in that module. The slave then generates a VICbus interrupt in the master VIC-8251 in the EVB crate. That module then generates a VME interrupt, handled by the EVB FIC-8234.
- The OPTO FIC-8234 may put a message into a reserved portion of the memory of an SL-30 VME slave and manipulate an interrupt register in that module to signal it.
  An SL-30 may put a message into another reserved portion of its own memory and generate a VME interrupt. In OPTO crate 1 (see Fig. 3) it is directly handled by the OPTO FIC-8234. In crates 2 and 3 it is handled by a VIC-8251 slave module, which has been configured to convert the VME interrupt into a VICbus interrupt in the master VIC-8251 in crate 1. That module then generates a VME interrupt, handled by the OPTO FIC-8234.
- In the SPEC subsystem the two FIC-8234 modules communicate through a local VICbus, with the SPEC crate acting as master and the TUBE crate as slave.

In each case the messages are sent through an OS-9 process named BRIDGE_SEND. The accompanying remote interrupts are handled by an OS-9 interrupt driver developed for REMOS, which uses OS-9 primitives to wake up an OS-9 process named BRIDGE_RCV, which has registered itself as the user-level handler for such interrupts. This process examines the header of each message and forwards it to the DAQ process indicated as the intended recipient.

It must be noted that REMOS provides no guarantee that a message has been received by the intended recipient. It is up to higher level DAQ software to determine when the remote end is misbehaving, e.g. when it fails to reply within a given time window. A hang-up of some OS-9 CPU ultimately leads to error messages in the UNIX back-end. The only cure is to restart the whole DAQ (see Sect. 10).

Times are in μs	Local OS-9	Local REMOS	Remote REMOS
Semaphore signal	17	133	133
Semaphore wait	18	230	825
Null RPC	205	205	800

Table 1: The performance of REMOS under optimal conditions.

It must also be noted that the communication scenarios described above are only used for very short REMOS *control* messages, for which small portions of the VIC-8251 and SL-30 memories are permanently allocated. True DAQ messages are exchanged through shared memory, as described below and in the next section.

Addressing in REMOS

In general, for communication between processes to be possible, each of them must have an address through which it is uniquely identified. In REMOS the address of a process is given by its **global process identification** (GPID), which consists of two parts: a globally unique part identifying the CPU on which the process is running, and a locally unique number for each process on the CPU. The global part is determined by the VICbus branch and crate numbers, and in essence by the VME slot number (i.e. VME address range) when the crate contains multiple VME boards running REMOS services. The local part of the GPID is fixed for the few REMOS service processes, and dynamic for the REMOS clients. On each participating CPU, the first process that intends to make use of REMOS services, creates by means of its own internal REMOS_MANAGER object a shared local table containing one REMOS_PORT object per process. The slot number of any process serves as the local part of its GPID.

A REMOS_PORT is the handle for communication between a client and a *local* REMOS service process. For example, when a service process wants to send a message to a client, it first establishes a lock on an OS-9 semaphore within the client's REMOS_PORT, after which it puts the message into the port's data buffer; then it signals on another semaphore that a message is waiting to be read, which wakes up the client, if necessary; finally, it releases the lock.

In the table of REMOS_PORT objects the first slot is reserved for the REMOS process. This is the process that manages the global process table, global shared memory and global semaphores. In this functionality it acts as a remote operating system indeed. On each subsystem that is participating in a global DAQ, the REMOS process is represented by a proxy, which forwards incoming messages to the true REMOS process running on the remote EVB CPU. The next two slots in a REMOS_PORT table are reserved for BRIDGE_SEND and BRIDGE_RCV, and the rest of the table is for client processes.

The table is identified by a name which indicates its role in the DAQ, e.g. EVB for communication between a subsystem and the EVB. This allows each subsystem to instantiate also a *local* REMOS to facilitate communication between CPUs in different crates *within* the subsystem. This approach has indeed been followed in the OPTO subsystem, where a local instance of REMOS is used for the communication between the FIC-8234 and the 29 SL-30s. Their REMOS_PORT table is named OPTO. Also for the SPEC subsystem a local instance of REMOS was prepared to facilitate the communication between its two FIC-8234 modules, but by that time a custom solution had already been in use for a while and in the end the SPEC REMOS was never used. The three REMOS instances are shown in Fig. 4. Table 1 shows the performance of REMOS under optimal conditions.

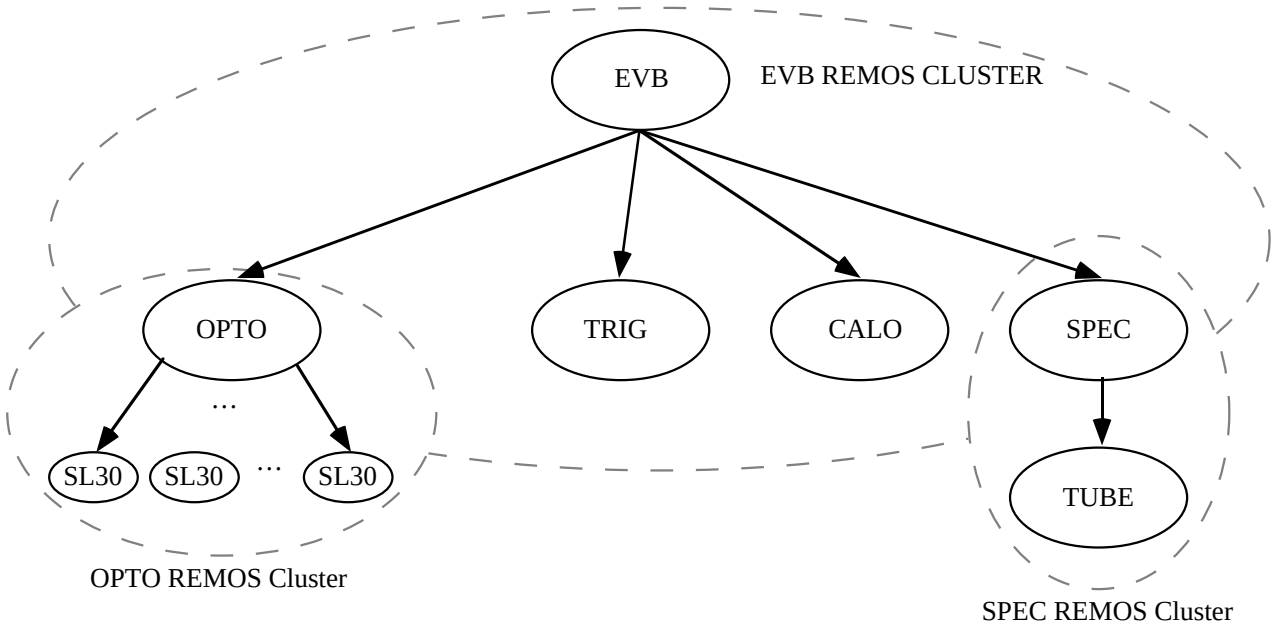

Figure 4: The three instances of REMOS.

Each process using REMOS facilities registers itself by name with the REMOS master process, which allocates a slot for it in the global process table. A process name starts with the name of the subsystem and ends with a piece indicating the role of the process, e.g. OPTO_READ and EVB_MAIN. A similar naming convention is used to allow any two DAQ processes to establish multiple communication channels between them. This functionality is implemented by the buffer manager, on top of REMOS as well as OS-9. REMOS is only used to register DAQ processes, to allocate or release global shared memory and to operate global semaphores. The corresponding messages are very short and use a permanently reserved part of the memory of each VIC-8251 and SL-30. True DAQ control and event data messages are exchanged through shared memory, which is formatted by and accessed through the buffer manager.

5 The Buffer Manager

Within each DAQ process on OS-9 a buffer manager C++ object handles multiple buffered communication links with one or more other DAQ processes. Each link is implemented as a shared message queue or semaphore with a predefined name that is known to both parties. A semaphore is essentially implemented as a FIFO of signal requests, and a process is only blocked when it issued a wait request while the FIFO is empty. A message queue is implemented within an identically named shared memory segment, which usually is obtained through OS-9 when all users of the queue are known to run locally, and otherwise through REMOS. In the latter case the EVB REMOS will allocate a portion of the on-board memory of the VIC-8251 module in the subsystem crate. The OPTO REMOS will allocate a portion of the on-board memory of the SL-30 from which the request was issued. This allows all subsystems and SL-30s to prepare their event data in parallel, with an acceptable level of contention for the VICbus and the REMOS master process.

To add a new message to a queue a process first establishes a write lock on that queue by waiting on a corresponding semaphore, thereby suspending any concurrent write attempts (control messages can come from multiple sources). If the queue is full, the process waits on a

semaphore to be signaled as soon as the (single) reader has drained the queue sufficiently. Next the process acquires a *mutex* lock on the queue, after which it can allocate a free buffer and update the message queue data structure. The mutex lock is then released to allow the reader to read any waiting messages. At the same time the writer process can copy the message data directly into the buffer, using at most the local VME bus resource. Next it acquires the mutex lock once more to update the data structure with the final size of the message, and releases the write lock. If the reader is currently waiting on a semaphore because the queue was empty, the writer signals that semaphore. Finally it releases the mutex lock.

Since the EVB is usually communicating with multiple subsystems, whenever a message has been prepared for the EVB, a signal is sent to a globally known REMOS semaphore, which wakes up the EVB process if necessary, and indicates which subsystem has a message ready. On each subsystem the main process has various message queue links with the EVB: an input queue for receiving control messages (also from other processes in the subsystem DAQ), an output queue for sending status messages to the EVB, and another output queue into which the event data are made available to the EVB. For simplicity these links are always obtained through REMOS, even in a stand-alone DAQ, in which case the subsystem's global VIC-8251 simply serves as a memory module.

6 The OS-9 DAQ Processes

The OS-9 side of the DAQ is divided into the EVB and the subsystems. The EVB is in control of the subsystems, each of which is in control of its own detector hardware. Both on the EVB and on each of the subsystems the implementation of the necessary functionality has been spread over various cooperating processes. This allows the DAQ to do various tasks in parallel, to proceed even when a particular process is waiting e.g. for an I/O request to be handled by OS-9. Most processes are in an infinite loop of waiting for the receipt of a message in the input queue and carrying out the associated action within a predefined time.

The EVB Processes

1. EVB_MAIN – This process controls all other DAQ processes, keeping track of their status. It is steered itself by user command messages from the EVB_IN process, and status messages from the other processes on the EVB and on the subsystems. On the receipt of a message it performs an action, possibly delegating the real work to one of the other DAQ processes. In that case the relevant process is supposed to confirm within a predefined time that the work has been done. This is enforced by having the EVB_TIMER process regularly send wake-up messages to EVB_MAIN, until the relevant timer has expired. If the confirmation still has not been received, EVB_MAIN gracefully stops the DAQ, after sending diagnostic information to the user.
   All status information is made available to the user through the EVB_DSP process, which forwards messages to the DISPATCHER process running on the UNIX back-end (see Sect. 9). After each neutrino burst EVB_MAIN receives the list of event descriptors from the TRIG subsystem. For each event (i.e. hardware trigger) the descriptor indicates which subsystems are supposed to have recorded data in their front-end electronics modules. The list is broadcast to all subsystems, which then put their data in their data output queues. Whenever a subsystem has data ready for the oldest incomplete event, EVB_MAIN signals the EVB_WRITE process to handle that portion.
2. EVB_WRITE – This process takes care of writing the event data to stable storage. Each subsystem puts the data for a particular event as a message into its data output queue

and signals EVB_MAIN. The message is formatted as a sequence of **banks**, each consisting of a header identifying the event, followed by the data of a particular subdetector. The task of EVB_WRITE is to combine all banks of all subsystems into complete events, and write those in ZEBRA [20] format to stable storage. The global DAQ originally used IBM 3480 tapes operated by a drive directly connected to the EVB FIC-8234. For a local DAQ originally a local Exabyte drive was used, or an NFS-mounted file system. In any case, the event data are first copied into a shared message queue on OS-9, such that the EVB_DSP process can forward them to the DISPATCHER process on UNIX, which in turn makes the data available to various monitoring tasks. This allowed the EVB tape drive to be abandoned in the course of 1996, when its operation had become unstable. Since then the event data are saved by a dedicated UNIX task (see Sect. 10). Also the local DAQs profited from this change.

3. EVB_DSP – This process is instructed by EVB_MAIN and EVB_WRITE to send status messages and event data to the DISPATCHER process running on the UNIX back-end (see Sect. 9). In fact there are two EVB_DSP processes. One is used to send event data and urgent messages. The other buffers its input to send bursts of 16 kbytes, thereby reducing the bandwidth consumed per message.
4. EVB_TIMER – This process is instructed by EVB_MAIN to send back a reply after a specified time-out. At the arrival of the reply, EVB_MAIN checks if in the meantime it has received the confirmation from another DAQ process that a particular command has been carried out. If the relevant timer has expired, the DAQ is terminated.
5. EVB_IN – This process converts user commands to control messages which are sent to EVB_MAIN. Under normal operation the user sends commands by invoking a UNIX front-end program that forwards them to EVB_IN through a direct socket connection. The front-end sends a copy of each command to the DISPATCHER such that it is logged (see Sect. 11). In principle EVB_IN could have been changed to receive its input from the DISPATCHER when that option became available.

The Subsystem Processes

1. SUBS_MAIN – This process controls all DAQ processes pertaining to the subsystem. It is steered itself by command messages from EVB_MAIN and by status messages from the other local processes. Once every minute it is asked by the EVB to confirm that it is still alive. A library provides the generic part of the code, which handles the communication with the rest of the DAQ. Programmers of the subsystem specific code ("subsystem experts") only have to fill in a few subroutines, which are invoked at well-defined occasions, e.g. when the DAQ is initialized, when a run is started, when event data are to be read out etc. For all subsystems except OPTO, SUBS_MAIN itself reads the data from the front-end electronics and prepares the corresponding banks in a shared message queue on OS-9, such that the SUBS_HIST process can access them for monitoring purposes. The banks are then copied into the data output queue of the subsystem.
   The presence of global events is signaled by a message from EVB_MAIN giving the event descriptors, whereas the presence of each single local event is signaled by a message from the SUBS_CIRQ process. Each global event descriptor contains an **event tag**, a 32 bit quasi-random number that should be identical to the value stored for the corresponding event by the subsystem's **buffered pattern unit** (BPU [10]). Upon each global trigger the BPU modules of all subsystems store the current value of a shared 500 kHz counter operated by the TRIG subsystem. By this mechanism it is easily detected whenever the

event data in a particular subsystem get out of synchronization with the rest of the DAQ. The subsystem specific code has to check itself if all its other electronics modules contain the correct number of events.
When the global events have been handled, and also whenever a local event has been handled, SUBS_MAIN sends a message to the SUBS_CIRQ process to allow for another local event. The reply from SUBS_CIRQ may be a local event descriptor message, but eventually it will be a **preburst** message, indicating that the next neutrino burst is supposed to arrive in about one second. Local events are then inhibited until the global events of the burst have been processed.
At the start of a run (requested by EVB_MAIN) the subsystem specific code can indicate that it first needs calibration events. In that case SUBS_MAIN delays sending an acknowledgement to EVB_MAIN and temporarily enables the local trigger. The subsystem specific code must ensure that the actual events are suitable for calibration, e.g. by arranging for a pulser to fire in synchronization with the trigger. Each local event will be marked as a calibration event, until the subsystem specific code indicates that the necessary number of such events has been reached. After that the local trigger is disabled and an acknowledgement is sent to EVB_MAIN indicating that the subsystem is ready for real events. When every participating subsystem has sent an acknowledgement, EVB_MAIN will usually let each SUBS_MAIN enable its local trigger. In a global DAQ, however, the local trigger of any subsystem may be kept disabled when its local events are unwanted.

2. SUBS_CIRQ – This process operates the **CIRQ** [10] VME trigger module of the subsystem. For each accepted *local* trigger or preburst signal the CIRQ generates a VME interrupt that is handled by SUBS_CIRQ, which then sends a local event descriptor or preburst message to SUBS_MAIN. The CIRQ ignores further input signals until it is explicitly reset by SUBS_CIRQ, which happens when SUBS_MAIN requests another local event. SUBS_CIRQ then starts waiting for an OS-9 semaphore to be signaled either by a custom-written VME interrupt driver, or by SUBS_MAIN itself, indicating that a message is waiting, in particular a request to disable the local triggers immediately.
Global triggers are recorded by the TRIG_CIRQ process running in the TRIG subsystem. During the 6 ms neutrino burst TRIG_CIRQ continuously polls a register in its CIRQ module, which signals that a trigger has occurred. Interrupts are not used during the burst, since their overhead would give too much dead-time. Another register shows which subsystems received the trigger. As soon as a subsystem has reached the maximum number of events it can buffer, TRIG_CIRQ applies a local veto to shield that subsystem from further global triggers. The 6 ms burst is handled in an 8 ms physics gate, the polling is done over a 32 ms wide gate. After the burst TRIG_CIRQ sends a list of global event descriptors to the EVB. Before the next burst, TRIG_CIRQ expects all participating subsystems to have signaled a shared REMOS semaphore, indicating that they are ready. As long as there is any subsystem not ready, further global events are inhibited and diagnostic messages are sent to the user. This situation usually occurs when the EVB and hence the subsystem accumulate a backlog in the processing of event data. Another possibility is that the subsystem has stopped working altogether.

3. SUBS_READ – This process, when present, is responsible for reading the event data from the hardware and storing them in the form of banks into the data output queue buffer allocated by SUBS_MAIN. This approach has only been taken in the OPTO subsystem, to allow OPTO_MAIN to signal that the OPTO is ready for new events as soon as every SL-30 has signaled that it has copied the current data from its video buffer to its main

memory. The communication with the SL-30 processes is based on semaphores and shared memory, all obtained through the OPTO REMOS. Extra communication links between OPTO_MAIN and OPTO_READ are provided by a semaphore and a shared memory module, both obtained through OS-9. OPTO_READ prepares the actual events asynchronously, makes them available to OPTO_HIST and copies them into the data output queue. Handling of the second burst in the cycle is delayed until the data of the first burst have been processed.

4. SUBS_HIST – This process originally served to calculate event data distributions for monitoring purposes. Its input was provided by SUBS_MAIN or SUBS_READ in a shared OS-9 message queue. The results were sent via the EVB to the DISPATCHER. Because of negative interference with the rest of the subsystem DAQ (see Sect. 8), this functionality has been moved completely to the UNIX back-end.

7 The Concurrent Hierarchical State Machine Language

Each of the OS-9 DAQ processes has been constructed to run as a finite state machine. First the set of states in which it can be, has been defined, followed by the set of allowed state transitions, and finally the set of actions that should be performed on these state transitions. Each state transition is provoked by the occurrence of a specific stimulus, generally called a state transition *event*, and typically caused by the arrival of a corresponding message. Each transition has an associated set of brief, non-blocking actions to be performed, after which the process is again in one of the defined states, waiting for the next transition message.

The first versions of the OS-9 DAQ programs were written directly in C++. The total amount of code, including configuration and test programs and support libraries, comprises about 50k lines (1.2 Mbytes) and 80 classes. In spite of the object-oriented features of C++, debugging the DAQ was hindered by the code being a mix of state machine concepts and transition actions.

A proper implementation of each program was greatly facilitated in the end by employing an extra programming language on top of C++, but only after it had first been improved significantly: **CHSM** [11]. This language allows for an easy implementation of Concurrent Hierarchical State Machines within a single program. The CHSM code defines the set of states, transitions, and transition events, using a syntax which is reminiscent of the YACC [12] compiler generator language. As an example a part of SUBS_MAIN is shown in Fig. 5.

The CHSM directives are interspersed with C++ code in which the transition actions are implemented. The CHSM compiler translates also the state machine framework into C++, such that the resulting code can be further processed by a standard C++ compiler.

The **concurrency** feature of CHSM allows a single program to be implemented as a **set** of parallel state machines, each of which dealing with a particular task and having its own collection of states, events, transitions and actions. The running program has a current state in each of the machines, and may make a transition in each of them on the receipt of a single message. The order of the transitions is specified by CHSM. As an example, the EVB_MAIN program defines a state machine to keep track of the global DAQ status, a second machine to handle communication with the subsystems, a third to communicate with the EVB_WRITE process that writes the CHORUS event data to stable storage, a fourth to deal with time-outs, a fifth to stop the DAQ gracefully in case of a problem, and finally a sixth that responds to user command messages.

Each of the state machines is implemented as a **cluster** of states. The running program has only a single current state in such a cluster. However, the **hierarchy** feature of CHSM allows

each state itself to be implemented as a set of state machines. For example, as far as its first state machine is concerned, EVB_MAIN can be either initializing, or functioning, or terminating; when it is functioning, it still has a lot of complexity to deal with; therefore the details have also been implemented as a set of state machines. One such low-level state machine is dealing with messages sent by subsystems, e.g. notifications that the data of particular CHORUS events have been prepared in data output queues. Another machine keeps track of the detailed status of the EVB, which either is idle, starting a run, running, pausing the run, continuing it, stopping it, or stopping the DAQ.

Using the CHSM framework one can only describe and control the states of a *single* process. To keep the various DAQ processes synchronized, the proper messages must be sent at the proper times. For example, when SUBS_MAIN receives a message requesting the run to be paused, it must itself send a message to SUBS_CIRQ to disable the local triggers. On the other hand, a message might also arrive unexpected. For example, due to a programming error there was a small probability that the EVB would still receive one extra local event descriptor after the local triggers supposedly had been disabled. Since such a message was not expected at that time, it did not generate any transition and was silently ignored. The corresponding event data were still pending at the head of the subsystem's data output queue, though. Those data would have been used erroneously to build the next (global) event, if it had not been for the mismatch of the event tag, allowing this desynchronization to be detected by EVB_WRITE. In a more robust derivation of CHSM, however, it should be possible to preclude such problems, e.g. by raising an exception whenever a particular message generates zero transitions.

8 Performance of the OS-9 Front-End

For the main operation of the CHORUS experiment (1994–1997), during each of the two neutrino bursts per SPS cycle typically five global events were recorded in 6 ms and had to be processed in about 1.5 s. The corresponding amount of data turned out to be less than 250 kbytes on average, but could be as large as 4 Mbytes when the OPTO subsystem was involved. In the 12 s interburst period, each subsystem could take local events. The combined local data rate was less than 50 kbytes/s on average.

Before the DAQ was designed, estimates of these data rates had been obtained from Monte Carlo simulations, but especially concerning the OPTO subsystem there was doubt if the real data would be sufficiently sparse to allow for the efficient compression (by a factor 100) finally achieved. Therefore it was judged wise to increase the speed of each FIC-8234 by some 30 % by disarming the Memory Management Unit of its MC-68040 CPU(s). This measure not only reduces the context switching overhead, but also allows each process to read and write anywhere in the CPU and VME address spaces, circumventing the operating system. At the same time, however, it increases significantly the difficulty of debugging the DAQ programs, since segmentation violations can no longer be trapped, allowing any process to crash another process, or even the operating system. In particular this is a problem when a DAQ program contains "user" code, i.e. routines filled in by the subsystem *detector* experts, who are not necessarily *programming* experts. To reduce the amount of such code running on OS-9, it was decided to move the data monitoring completely to the UNIX side of the DAQ and switch off the OS-9 SUBS_HIST processes, thereby significantly improving the stability of the DAQ. In the end the DAQ needed to be restarted once per day on average, usually because of a crash in the OPTO subsystem. The procedure to restart the DAQ takes a few minutes and is described in Sect. 10.

```
// ...
void gen::do_prepare_run() { /* ... */ }
%%
machine gen_mach : gen {
    set mach (the_main, responder) is {
        cluster the_main (undefined, active) is {
            state undefined {
                ev_init -> active  %{  do_init();  %};
            }
            set active (main, read, cirq, hist) is {
                cluster main (starting, ready, running, quitting, quit) is {
                    state starting {
                        ev_daq_ok -> ready  %{  finish_startup();  %};
                    }
                    state ready {
                        ev_prepare_run -> running  %{  do_prepare_run();  %};
                        ev_sys_quit -> quitting;
                    }
                    state running {
                        // ...
                        ev_end_run -> ready  %{  handle_end_run();  %};
                    }
                    state quitting {
                        ev_child_quit(active_children == 0) -> quit  %{
                            acknowledge_quit();
                        %};
                    }
                    state quit;
                }
                cluster read (undefined, running, quitting) is { /* ... */ }
                cluster cirq (undefined, running, quitting) is { /* ... */ }
                cluster hist (undefined, running, quitting) is { /* ... */ }
            }
        }
        state responder {
            ev_message -> responder  %{  handle_message();  %};
            // ...
        }
    }
}
%%
class gen_main : public gen_mach { /* ... */ };
// ...
int main (int argc, char *argv[]) {
    // ...
    gen_main *the_gen_main = new gen_main();
    the_gen_main->ev_init();
    // ...
}
```

Figure 5: Example of Concurrent Hierarchical State Machine code.

The DAQ has been designed to distinguish as little as possible between global and stand-alone operation. For example, even in stand-alone operation the DAQ is controlled by the EVB processes, in this case all running on the subsystem CPU, as indicated by dynamic configuration files. Furthermore, since the four subsystems differ quite a bit in their characteristics, the DAQ software has needed to be quite generic. This has allowed the software to be employed also outside the main context of the CHORUS experiment. First of all a stand-alone DAQ has been used in the CERN RD46 [13] experiment which investigated the performance of new optical capillary detectors in conjunction with the CHORUS experiment. The experiments had one trigger in common, already foreseen in the CHORUS DAQ software. Secondly a stand-alone DAQ has been used to debug the new Honeycomb subdetector [6] which was incorporated into the TRIG subsystem in 1996. Finally a modified stand-alone DAQ has been employed in two pion "testbeam" experiments at the CERN Proton Synchrotron.

For the operation of the CHORUS experiment it does not matter that for each event the total overhead of control messages is as large as 10 ms (cf. Table 1). During the burst the data are buffered in the electronics modules, and when after the burst the data are collected and the events are built, the overhead can easily be tolerated. For local events this overhead has also been acceptable. In the testbeam experiments, however, it turned out to be unacceptable.

In those experiments the data-taking cycle had a length of 14.4 or 19.2 s and contained two or three bursts separated by 4 s or more and lasting 350 ms each. Unfortunately most of the employed electronics modules did not have a buffering capability and thus had to be read out immediately after each trigger. In an unmodified stand-alone DAQ the maximum data rate would then be about 35 events per burst and the dead-time would approach 100 %. To improve the performance quite significantly, the following measures were taken:

- The SUBS_CIRQ program was incorporated into the SUBS_MAIN program, thereby eliminating the overhead of the local event descriptor message.
- The event data were temporarily stored by SUBS_MAIN in private memory, delaying the true event building and its buffer manager overhead until the end of each burst, signaled by a special trigger.
- After the event building SUBS_MAIN sent a message to a trivial helper process (a piece of the original SUBS_CIRQ), requesting an immediate "wake-up" reply ordering the preparation for the next burst, i.e. a tight loop of waiting for triggers and handling them. If in the meantime another message had arrived, that one would be handled first, allowing e.g. the run to be paused, in which case the wake-up reply would be ignored.

With these measures the readout overhead during the burst became negligible (< 0.1 ms per event) with respect to the time needed for the actual readout of the electronics modules ($\sim$5 ms per event).

9 The Dispatcher

The event data and status messages of the OS-9 DAQ processes are made available to the high-level side of the DAQ through the DISPATCHER process running on the UNIX host controlling the DAQ. The DISPATCHER is a general-purpose message-based data distribution program provided by the freely available CONTROLHOST package [14]. A library provides subroutines for a process to connect itself to the DISPATCHER, to send messages to interested processes, and to receive selected messages from them. Each message consists of a header and a body. The header indicates the length of the body and contains a **tag** which identifies the type of the data in the body. A tag is a string of up to eight ASCII characters. The body of a message is not interpreted. When a client process intends to receive data, it must first supply

a **subscription** list containing the tags in which it is interested. Then, whenever a message is sent to the DISPATCHER, a copy is in principle made available via shared memory or a socket, to all clients that are subscribed to the corresponding tag. Normally the message is only received by those interested clients that are actually waiting for a message. However, a process may also indicate in its subscription list that it wants to receive *all* messages with particular tags. Each such message is then kept around until all such processes have received it. The danger associated with this option is that the memory of the DISPATCHER may fill up when clients cannot keep up with the rate of incoming messages. Data monitoring processes do not use this option, since they can perform their tasks with only a fraction of the events. There are also processes, however, that cannot afford losing even a single message: the DISKWRITER has to save all events, and various control processes need to receive all status messages (see Sect. 10 and 11).

Messages are also used by some clients to send commands to other clients, all running on UNIX. In principle the DISPATCHER could also have been used to send commands to the EVB_IN process on OS-9, but by the time the DISPATCHER was available a direct socket connection had been in use for a while and in the end the code was never adapted. A copy of those commands is still sent to the DISPATCHER, though, to allow them to be logged together with other commands.

Commands can also be sent to the DISPATCHER itself, e.g. to obtain the current list of clients, their tag subscription lists, and statistics. Each client usually registers itself, with a name of up to eight characters. For each such registration the DISPATCHER sends a message itself to all clients subscribed to the tag "Born"; the message body contains the name of the new client. Whenever a process disconnects from the DISPATCHER a message with tag "Died" is sent. The DISPATCHER can also report if a particular client is currently connected. These facilities allow control processes to monitor important clients.

Each of the UNIX hosts in the CHORUS DAQ cluster starts a DISPATCHER process at boot time. For a stand-alone DAQ one of five IBM RS-6000 PowerPC workstations (see Fig. 2) is chosen and its DISPATCHER is used in the DAQ. In that case only the Ethernet is used for the communication between UNIX and OS-9. In a global DAQ the DISPATCHER runs on a CETIA RS-6000 PowerEngine VME board. Also this machine has IBM AIX 4.1 as its operating system. As Fig. 2 shows, originally a private high-bandwidth channel was foreseen for the communication between the EVB FIC-8234 and the PowerEngine, located in a different VME crate (see Sect. 10): first the FIC-8234 would write the data through a VSB connection into a 64 Mbyte memory module [15] located in the crate of the PowerEngine; then the DISPATCHER would be notified via the Ethernet; finally the PowerEngine would read the data through VME from the memory module. The VSB connection was driven by a pair of rear-mounted cards [16]. Buffers were allocated in the memory module following a simple round-robin algorithm. Each buffer had a flag that was cleared by the DISPATCHER as soon as it had no more need for the buffer contents. In principle this scenario worked as expected, but the FIC-8234 was observed to crash more frequently when its VSB interface was being used. Since the bandwidth of the Ethernet turned out to be sufficient after all to cope with the data rate of the global DAQ, the VSB approach was abandoned in the end. This was made possible by the router. A VICbus connection was not tried at all, because the newly available VSB link was supposed to be faster, simpler to set up, and more reliable. It was observed that VICbus modules occasionally needed to be reset by a VME SYS-RESET signal or even a power cycle, by which the stability of the UNIX back-end would have been impaired. The PowerEngine has a private Ethernet link to the router, such that the other DAQ hosts do not see the traffic generated by copying the data runs to the computer center (see Sect. 10).

10 Control Panel and Supervisor

The high-level back-end of the DAQ has been implemented on UNIX. Virtually all of the relevant processes are clients of the DISPATCHER and most of them have graphical user interfaces (GUIs). The experiment is controlled from five IBM RS-6000 PowerPC workstations (see Fig. 2), each having two large screens displaying the Control Panel and current results of various data monitoring processes.

The graphical control PANEL has been implemented in the Tcl/Tk [17] programming language, whose interpreter was extended with support for communication with the DISPATCHER. The code amounts to about 9k lines (∼300 kbytes). Since the PANEL is the primary interface between the user and the DAQ, it must try and ensure that all DAQ programs are functioning properly. Any problems detected are reported to the user through pop-up windows, but may also cause the PANEL to take a corrective action itself.

The PANEL is used to control the global DAQ as well as each stand-alone DAQ, the only differences being in the configuration files. This implies that multiple instances of the PANEL may be active at the same time. Each subsystem (and the EVB) can only be controlled by one PANEL at a time, either for the global DAQ or a stand-alone DAQ. Furthermore, to simplify matters, only a single PANEL instance is allowed per UNIX host. Therefore, when a PANEL is started, it first contacts its own DISPATCHER to see if another PANEL happens to be connected already. If that is not the case, it sends a message with tag "PANEL" and as body the string "WHICH-SYSTEMS" followed by the name of its own UNIX host, to the DISPATCHER on each of the other UNIX hosts. A connected PANEL, if any, would reply by sending a message to the indicated UNIX host, with tag "PANEL" and as body the string "SYSTEMS-IN" followed by a list of subsystem names. When any of the necessary subsystems is thus found to be in use, the user is notified and the PANEL quits. Otherwise, for a global DAQ the PANEL then prevents the user from including any subsystem already taken by a stand-alone DAQ.

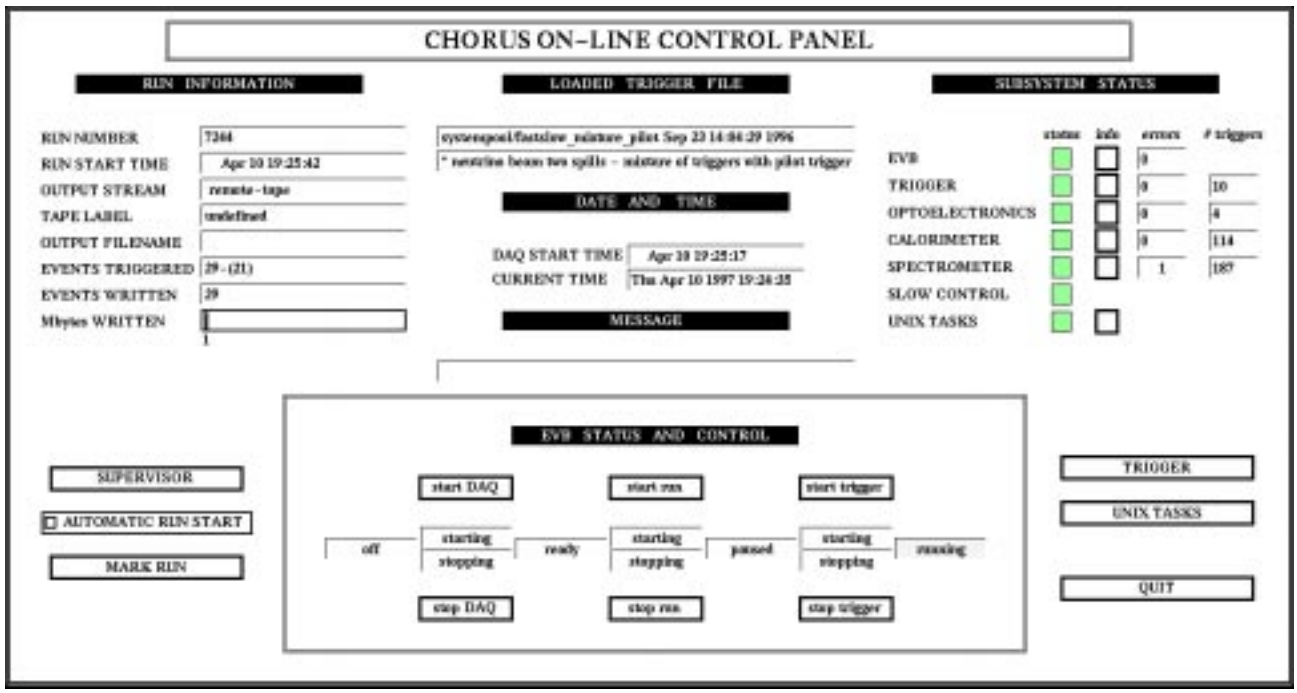

Figure 6: The CHORUS control panel. Details are described in the text.

The Panel User Interface

A realistic appearance of the global DAQ PANEL is shown in Fig. 6. However, most of the coloring has been suppressed here, to allow all texts to be readable. The canvas is divided into various sections showing information about the current status of the DAQ, and allowing the user to control various aspects. The many status fields and "lights" are updated according to the messages received through the DISPATCHER. The "RUN INFORMATION" section displays the progress of the current run as far as global events are concerned. The "OUTPUT STREAM" text field indicates here that the events are sent from the EVB to the DISPATCHER, to be written to a "remote tape" by a client of the DISPATCHER. The "EVENTS TRIGGERED" text field shows the number of global events and between parentheses the number of true coincidence triggers, the difference being the number of neutrino bursts, each of which leading to one extra event for the readout of burst scalers. Such scalers are used e.g. to count the number of beam-induced muons that traverse the emulsion target.

The "LOADED TRIGGER FILE" section displays the operating conditions of the TRIG subsystem. Here they reflect the normal data-taking, with two neutrino bursts per SPS cycle and a mixture of enabled trigger types, including the trigger shared with the RD46 "pilot" project (see Sect. 8). The "DATE AND TIME" section shows when the DAQ was last restarted according to the EVB, and the current time according to UNIX host running the PANEL. From the example picture can be concluded that the system clocks were not always precisely synchronized. The "MESSAGE" section informs the user of the progress whenever the PANEL is busy with a time consuming procedure, e.g. starting the DAQ. Such messages are also sent to the DISPATCHER, allowing easy monitoring through the SCREEN_LOGGER (see Sect. 11), which displays all control messages sent by all DISPATCHER clients.

The "SUBSYSTEM STATUS" section has a "status light" not only for each of the subsystems (and the EVB), but also for the "SLOW CONTROL" (see Sect. 11) and the "UNIX TASKS," the latter being the important data monitoring DISPATCHER clients. A status light is green when its "subsystem" is functioning properly, and red when there is a serious problem. The lights of the four true subsystems are yellow when the DAQ is ready for a run to be started. During the starting of the DAQ they go through various shades of gray, reflecting the progress. The next column of "info" buttons allows the user to pop up a window containing recent warnings and error messages pertaining to the subsystem. The number of error messages for the current run is counted in the adjacent column. Whenever such a counter increases, its background changes from white to yellow, to indicate that new error messages are to be read in the associated pop-up window. The last column contains the number of local events for each of the true subsystems.

The lower half of the PANEL contains various elongated buttons on the left and right sides. The "SUPERVISOR" button allows the user to restart the DAQ SUPERVISOR program (see below). Here the button is disabled, because the SUPERVISOR is working fine, as far as the PANEL is concerned. Each minute the PANEL sends a message to the DISPATCHER asking the SUPERVISOR to respond, thereby confirming that it is active. Only when the SUPERVISOR fails to respond, is the restart button enabled. The "AUTOMATIC RUN START" button allows the user to let the PANEL start the next run automatically, when the current run terminates normally after reaching the output file size limit. The "MARK RUN" button produces a pop-up window allowing the user to stamp a problematic or special run and to provide some details in support of that decision. The "TRIGGER" button produces a pop-up window for changing the conditions of the global trigger or, per subsystem, the local trigger. The "UNIX TASKS" button produces a window allowing the user to restart any of the important data monitoring clients, or send some command to it via the DISPATCHER, e.g. to save histograms on disk immediately or to send the current plot to the printer. The PANEL starts most monitoring clients itself.

The "QUIT" button allows the user to stop the PANEL process and the monitoring clients, in principle *without* affecting the data-taking. Conversely, when the PANEL starts, it allows the user to have it attach itself to a running DAQ, if any. In that case it asks the EVB and the SUPERVISOR (see below) about the current parameters and statistics, and fills in the various text fields accordingly.

Finally, the "EVB STATUS AND CONTROL" section of the PANEL shows the current status of the DAQ and allows the user to modify it. At any time most of the buttons are disabled, to enforce a proper sequence of operations. In the example the DAQ is in the running state and only the "stop trigger" button is enabled. When the user clicks on that button, the PANEL highlights the "stopping" text field above the "stop trigger" button and sends the appropriate command to the EVB_IN process. The EVB then forwards the command to the subsystems. After receiving their confirmations, it sends a message to the DISPATCHER, with tag "PANEL" and as body the keyword "PAUSED." When the PANEL receives this message, it highlights the "paused" text field, disables the "stop trigger" button, and enables the "start trigger" and "stop run" buttons. When the "stop run" button is clicked, the PANEL ends up in the "ready" state in a similar way, with only the "start run" and "stop DAQ" buttons enabled. Clicking on "stop DAQ" leads to the "off" state, with only the "start DAQ" button enabled. The procedure to start the DAQ is a bit more complicated, since there is no well-behaved DAQ yet to communicate with.

Starting the DAQ

The first step in starting the DAQ is to reboot all participating FIC-8234 boards, resetting their operating system, server processes and memories to well-defined states. For a global DAQ this procedure takes advantage of the VICbus connection between the EVB and each subsystem. Each FIC-8234 has been configured to reboot itself when either its front-panel reset button is pushed, or an internal reset register is written to, or the VME SYS-RESET line is pulled. Furthermore, a front-panel or register reset leads to a VME SYS-RESET as well. The VIC-8251 master modules in the EVB crate and in the subsystem crates have been configured to convert the VME SYS-RESET into a VICbus SYS-RESET. The VIC-8251 slave modules in the subsystem crates have been configured to convert the VICbus SYS-RESET into a VME SYS-RESET. These configurations allow all participating subsystems to be rebooted automatically and all readout crates to be reset whenever the EVB FIC-8234 is reset. As a by-product it also inhibits the UNIX back-end CETIA PowerEngine board from being in the EVB crate (see Sect. 9): not only would a VME SYS-RESET crash the DAQ back-end, but also it could lead to file system corruption. Unfortunately the board could not be shielded from the VME SYS-RESET line. All the UNIX hosts and their disks, for that matter, were powered from uninterruptible power supplies.

When OS-9 and the relevant server processes are running properly, a FIC-8234 can be reset by launching a remote shell command that writes the appropriate value to the internal reset register. Otherwise the user must push the front-panel button. These are the alternatives for a stand-alone DAQ. For the global DAQ a superior approach was introduced in the course of 1996: a true VME SYS-RESET could now be generated via the CAENET [18] bus, driven by a CAEN A200 SY127 controller [18] operated by a program running on the UNIX back-end CETIA PowerEngine board (see the CAENET link in Fig. 2). This allows the EVB and all participating subsystems to be rebooted by the PANEL without further user intervention. This procedure usually worked fine, but occasionally some subsystem crate would mysteriously fail to get fully reset. The ultimate cure for such a problem was a power cycle of the crate in question.

Each participating FIC-8234 downloads a file from the UNIX file server, containing the OS-9 kernel and a number of important server and utility programs, one of which is started by OS-9 as the last step of the reboot. This program initializes the networking, mounts the OS-9 file system from the UNIX file server through NFS, executes a start-up shell script located on that file system, and executes in an endless loop a login program on the serial terminal port. To allow the PANEL to monitor the progress of the reboot, the start-up script invokes at specific steps a command which sends a message with tag "PANEL" to the DISPATCHER. The name of the DISPATCHER host is found in a configuration file prepared by the PANEL. The body of each message contains the name of the subsystem and the current phase of its boot procedure. The PANEL expects to see the phases "BOOTED," "VIC-CONFIGURED" and "REMOS-CONFIGURED," after which the subsystem should be ready for the DAQ to be started.

When all participating subsystems have thus rebooted, the PANEL launches via the remote shell daemon on the EVB FIC-8234 a script that in turn starts the EVB processes participating in the DAQ (see Sect. 6). It then waits for the EVB_MAIN process to send a message to the DISPATCHER with tag "PANEL" and body "EVB-READY." The PANEL then checks itself if the other EVB processes are present, by launching a remote shell command on the EVB FIC-8234 to list the running processes. In principle the DISPATCHER could have been used here, at the cost of additional modifications in the OS-9 programs. Next the PANEL communicates via the EVB_IN process the list of participating subsystems to EVB_MAIN, which then launches on each of them the relevant DAQ processes, through a remote shell connection. Each SUBS_MAIN process is supposed to put a confirmation message into the EVB input queue, as soon as SUBS_CIRQ and any other dependent processes have registered themselves with SUBS_MAIN. When the EVB has thus received all confirmations, it sends another message to the DISPATCHER, with tag "PANEL" and body "READY," which leads the PANEL into its own "ready" state, allowing the user to start a run etc. In a stand-alone DAQ the subsystem DAQ processes are launched in parallel with the EVB processes, all on the subsystem FIC-8234, but the confirmation procedures are the same.

The Supervisor

Now that the DAQ is up, a run can be started. It is here that the SUPERVISOR comes into play. The SUPERVISOR is a daemon process started on each of the UNIX hosts when it is rebooted. Whenever the SUPERVISOR fails, the PANEL allows the user to restart the process. It is the SUPERVISOR who controls the data recording and saving on the UNIX side of the DAQ. When the user clicks the "start run" button, the PANEL pops up a window allowing the user to select the type of the run and its output stream. A "test" run would usually not be recorded, but only used for sending data via the DISPATCHER to monitoring tasks, to observe the behavior of subdetectors. A "real" run would originally be recorded by the EVB_WRITE OS-9 process, on a tape drive attached to the EVB or subsystem FIC-8234, or on a disk mounted from the file server through NFS. With the SUPERVISOR three new options became available: "remote disk," "remote tape" and "remote stage." The first option selects the recording by the DISKWRITER client, and is a prerequisite for the other two. The option "remote tape" instructs the SUPERVISOR to have a backup copy of the run written to a 10 GB DLT drive attached to the UNIX host. The option "remote stage" instructs the SUPERVISOR to have the run copied through an FDDI link to the CERN central stage pool in the computer center. There the run is split into various streams corresponding to selected trigger types, and each of those streams as well as the entire run are saved on tapes managed by the computer center. Furthermore a program is run to obtain a measure of the data quality, and its results are copied to the DAQ file server and made available in the ONLINE_MONITOR archive (see Sect. 11). For normal

data-taking all three SUPERVISOR options are enabled and none of the OS-9 options.

When the user has selected the output streams, the PANEL informs the SUPERVISOR through the DISPATCHER. If requested, the SUPERVISOR then starts the DISKWRITER when necessary, and instructs it to record the event data (in ZEBRA [20] format). Upon receipt of a confirmation message from the DISKWRITER, the SUPERVISOR informs the PANEL, which then proceeds by broadcasting to the DISPATCHER clients that a new run is about to start, and by sending a start command to the EVB, indicating the run number and the local output stream, if any. The PANEL then expects a message from the EVB that it has reached its "paused" state, indicating that all subsystems have obtained their calibration data, if any, and are ready for triggers. The PANEL then sends a "start trigger" command and expects the EVB to confirm that it has entered the "running" state.

During the run the PANEL instructs the monitoring tasks every 10 minutes to save their histograms, such that they can be viewed in the online monitoring archive (see Sect. 11). It also expects a message from the EVB every minute, confirming that the OS-9 processes are functioning well. Other messages from the EVB report e.g. the number of events triggered or the number of megabytes sent to the DISPATCHER. Finally it expects the SUPERVISOR to respond every minute, indicating that the data recording proceeds correctly.

A run is stopped either by the user through the PANEL, or by the SUPERVISOR through the PANEL, or by the EVB itself. The EVB would stop the run either when one of the subsystems fails, or when the local output file, if any, reaches its size limit ($\sim$200 MB, determined by the capacity of an IBM 3480 tape). The SUPERVISOR instructs the PANEL to stop the run either when the DISKWRITER fails, or when its output file reaches the size limit (200–800 MB). A normal run lasts between 1.5 and 6 hours. During normal data-taking the next run is started automatically by the PANEL, when the "AUTOMATIC RUN START" button has been selected and the current run has terminated normally. In any case the SUPERVISOR has to launch commands that copy the run, when requested, to the DLT drive and the remote stage area.

The SUPERVISOR has been written in the Expect [19] programming language. This extension of Tcl [17] allows multiple processes to be controlled, which are connected to the SUPERVISOR through *pseudo terminals*. These processes include the DISKWRITER, the copy commands, a logger for control and status messages, and a process managing a database holding the status of each run. Since almost all their messages are only of interest to the SUPERVISOR itself, using pseudo terminals avoids cluttering the DISPATCHER. Furthermore, when the output stream is a pseudo terminal, any UNIX utility only buffers its output up to a line, such that messages are immediately available to the SUPERVISOR. This allows e.g. the copy commands to be implemented as ordinary shell scripts. Whenever the SUPERVISOR is restarted, through the database it can find out if a run still needs to be copied. When all copies have been made, a run is marked for removal. Whenever the disk occupancy exceeds a threshold, the oldest marked run is removed. Since the main DISPATCHER host and its backup machine each have about 18 GB available for event data, and the rate is about 100 MB per hour, unmarked runs can be allowed to accumulate for more than a week. Occasionally runs remained unmarked up to a few days, due to the remote stage area filling up. The user can control the SUPERVISOR through the DISPATCHER by means of a GUI written in Tcl/Tk (see Sect. 11).

The 1998 Run

The main data-taking of the CHORUS experiment has taken place in the years 1994–1997. The data recorded during this period are used in the search for the neutrino oscillation phenomenon. In 1998 the experimental setup has been used to collect data for other analyses. To use the available manpower efficiently, the experiment has been run *without* shifts for the

larger part of that last year. Fortunately the OPTO subsystem could be excluded from the DAQ most of the time, so that significantly fewer crashes were expected. A run coordinator and various experts checked the status of the subdetectors and the data-taking a few times per day. To increase the efficiency of this largely unattended operation, the PANEL has been modified to play a more active role. Firstly, when this option is selected, the PANEL restarts the DAQ automatically after a crash, unless too many restarts have taken place in a short time, which points to a permanent problem, e.g. a VME crate fan failure. In such cases human intervention is needed, and by means of a modem the PANEL leaves a small numeric message on the pager of the relevant expert. Similarly the PANEL will start the next run whenever it finds itself in the "ready" state.

Experience with the data-taking in the preceding years allowed a list to be compiled of certain warnings and error messages indicating that data of dubious quality are being recorded. The PANEL counts such messages and whenever too many are received in a short time, it restarts the DAQ, which usually cures the problem, as all readout crates are reset. An important error is the mismatch of event tags (see Sect. 6) among the subsystems. Such an error would point to the loss of synchronization in one of the subsystems. However, occasionally a single event was observed to have a bad event tag in one of the subsystems, whereas the next event would be fine. This might be ascribed to noise in the hardware, or to a coding error in the subsystem DAQ. In any case it would be counterproductive to restart the DAQ immediately, unless such complaints are reported in a continuous stream. Event tag mismatches and other problems with the event data structure are reported by the EVB_WRITE OS-9 process and by the EQUIPMENT_CHECK client of the DISPATCHER (see Sect. 11). Another important complaint would come from the TRIG_CIRQ OS-9 process, when a subsystem is not ready to take global events. Finally, quite infrequently the EVB_DSP OS-9 process would complain that the DISPATCHER no longer accepted large event messages. This could sometimes be cured by pausing the run temporarily, thus allowing the clients to decrease the backlog. At other times the cause was a bug in the UNIX operating system itself. In that case also the SUPERVISOR would fail and an expert was paged immediately, to reboot the DISPATCHER host, restart the PANEL, and recommence the data-taking.

All in all the DAQ efficiency has been higher than 99 % even when it was running unattended, but controlled by the PANEL.

11 Monitoring Tasks and Utilities

During and after periods of data-taking various tasks and utilities are run on the UNIX back-end to monitor the behavior of subdetectors, of the neutrino beam, and of the DAQ itself. Almost all these programs are clients of the DISPATCHER and have been written in C++, C or Fortran-77. Most of them have graphical user interfaces (GUIs), implemented in the Tcl/Tk [17] programming language or using the CERNLIB HPLOT/HIGZ [20] library. Many of these programs are launched automatically by the PANEL. Here we describe those tasks and utilities that are important in ensuring that data of good quality are being recorded.

- FILE_LOGGER – This process is started on each UNIX host at boot time. It immediately connects to the DISPATCHER and logs all interesting control and status messages from any DAQ related process into a file, whose name reflects the current run number, or the current day when no run is being taken. After a week each file is compressed, but kept available for inspection, if ever necessary.
- SCREEN_LOGGER – This process is started by the PANEL and shows in a scrolling window on the screen the control and status messages currently being sent.

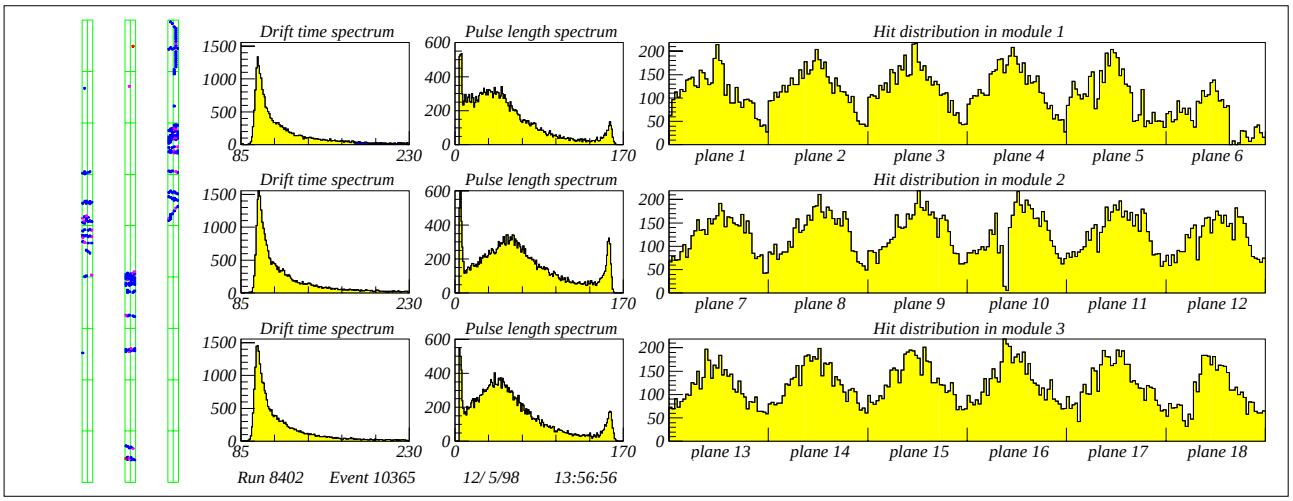

Figure 7: The CHORUS Honeycomb subdetector monitor. Besides drift time spectra, pulse length spectra and hit distributions the window also displays a snapshot event in a framework representing the subdetector's three modules, each having two rows of nine readout cards. Each wire with a signal is represented by a dot. Track segments can be distinguished.

- ERROR_LOGGER – This process is also started by the PANEL and keeps a list of the last 1000 warnings and error messages sent to the DISPATCHER. When the user clicks on one of the subsystem "info" buttons in the PANEL (see Sect. 10), the ERROR_LOGGER is instructed by the PANEL to pop up a scrolling window containing all the kept messages pertaining to the subsystem. New messages are added to the window as they are received by the ERROR_LOGGER.
- LOGBOOK – This utility is not a DISPATCHER client. It is started by the user to create or retrieve descriptions of the data-taking conditions, of problems with the subdetectors or the DAQ, and of actions taken. Its database is also updated by the PANEL with a short summary for each completed run.
- EQUIPMENT_CHECK – This process checks the structure and internal consistency of the data banks for each event it receives. Anomalies are reported as error messages to the DISPATCHER.
- SUBS_HIST, SUBS_AUTO_HIST – Each of the four subsystems "SUBS" has a corresponding SUBS_HIST task to calculate from the event data various distributions that serve in monitoring the performance of subdetectors. A histogram class library was written for the C++ clients, including an interface to the HBOOK [20] histogram format used by CERNLIB [20] utility programs and by the Fortran-77 clients. Every few minutes selected histograms are sent to the corresponding SUBS_AUTO_HIST task, which subsequently updates its display window on one of the workstation screens. All histograms are saved in an archive on disk, on command from the PANEL, sent every 10 minutes and at the end of a run. As an example Fig. 7 shows a window with data distributions of the Honeycomb subdetector [6].
- ONLINE_MONITOR – This utility is not a DISPATCHER client. It serves to view the archive of histograms created e.g. by the SUBS_HIST processes.
- EVENT_DISPLAY – This process shows the hits and reconstructed tracks of snapshot events in a model picture of the complete CHORUS detector. Fig. 8 is an example.

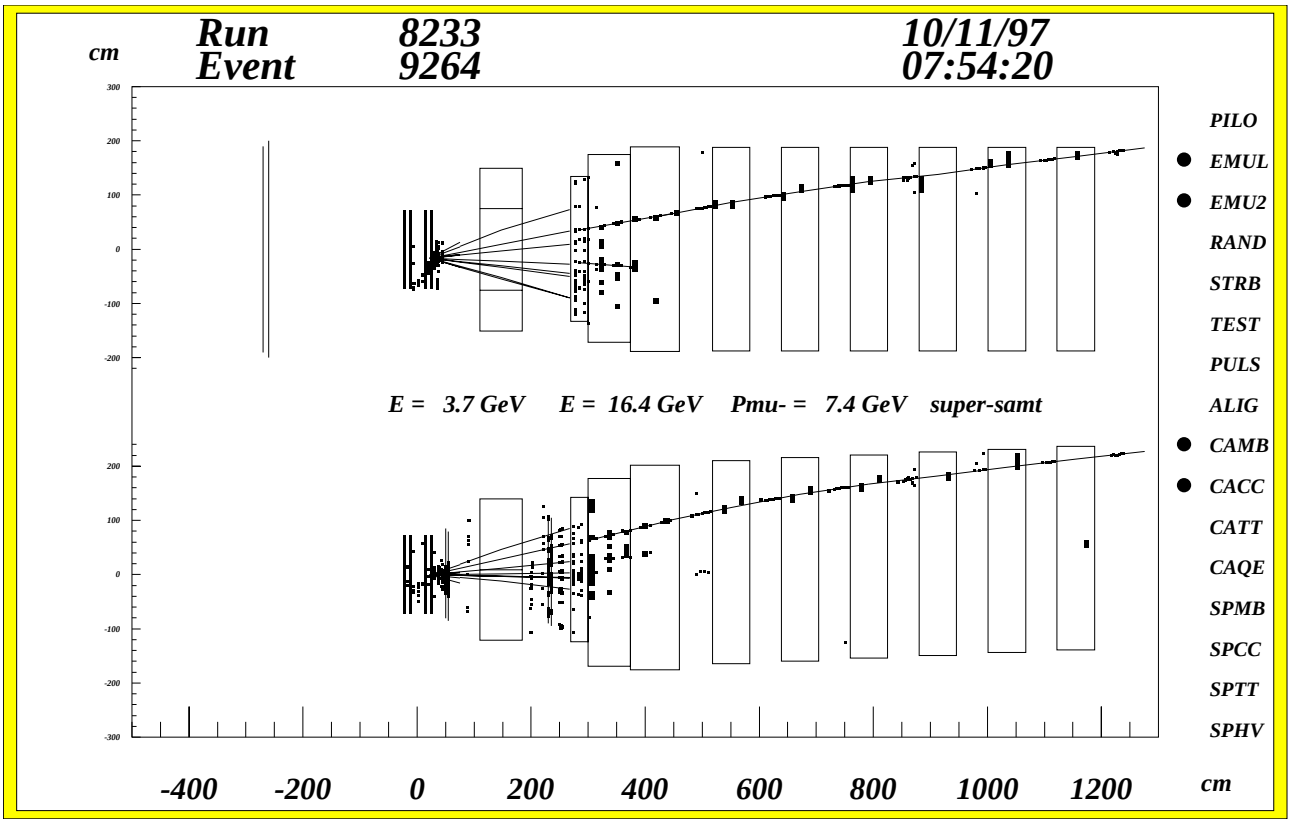

Figure 8: The CHORUS event display. The window shows the top and side views of the hits and tracks, superimposed on representations of the trigger planes and the heavy elements in the CHORUS setup: the four emulsion stacks, the hexagonal magnet, the three calorimeter modules, and the six spectrometer magnets. The top view includes the veto planes upstream of the target, the side view includes those trigger planes that have horizontal elements. The right edge of the picture shows which trigger bits are set for this event.

- CHORUS_EFFICIENCY – This process reconstructs tracks to obtain estimates of the efficiencies of the subdetectors. Since a proper calibration of the subdetectors can only be done off-line, this process can only approximate the true efficiencies. The results are recorded in histograms that are archived at regular intervals and can be viewed with the ONLINE_MONITOR.
- TRIGGER_RATES, TRIGGER_DEAD_TIMES – These processes update windows showing the rates and associated dead-time percentages for various trigger types, as well as the timing and structure of each neutrino burst, from information obtained by recording the corresponding muon flux in the muon pits upstream of the experiment. An example is given in Fig. 9. The run summaries of these processes are archived.
- ARCHIVES – A utility for browsing archives of summaries from various monitoring tasks.
- REMOTE_RECORDING – This process communicates with the SUPERVISOR and updates a window with the status of each unmarked run, i.e. a run that has not yet been fully saved. The user can also control the SUPERVISOR with it. For example, whenever there was a persistent problem with copying runs to the DLT drive or to the computer center, the corresponding part of the SUPERVISOR would "trip" and stop trying the operation. After correcting the problem a user could "reset the trip" via the GUI.

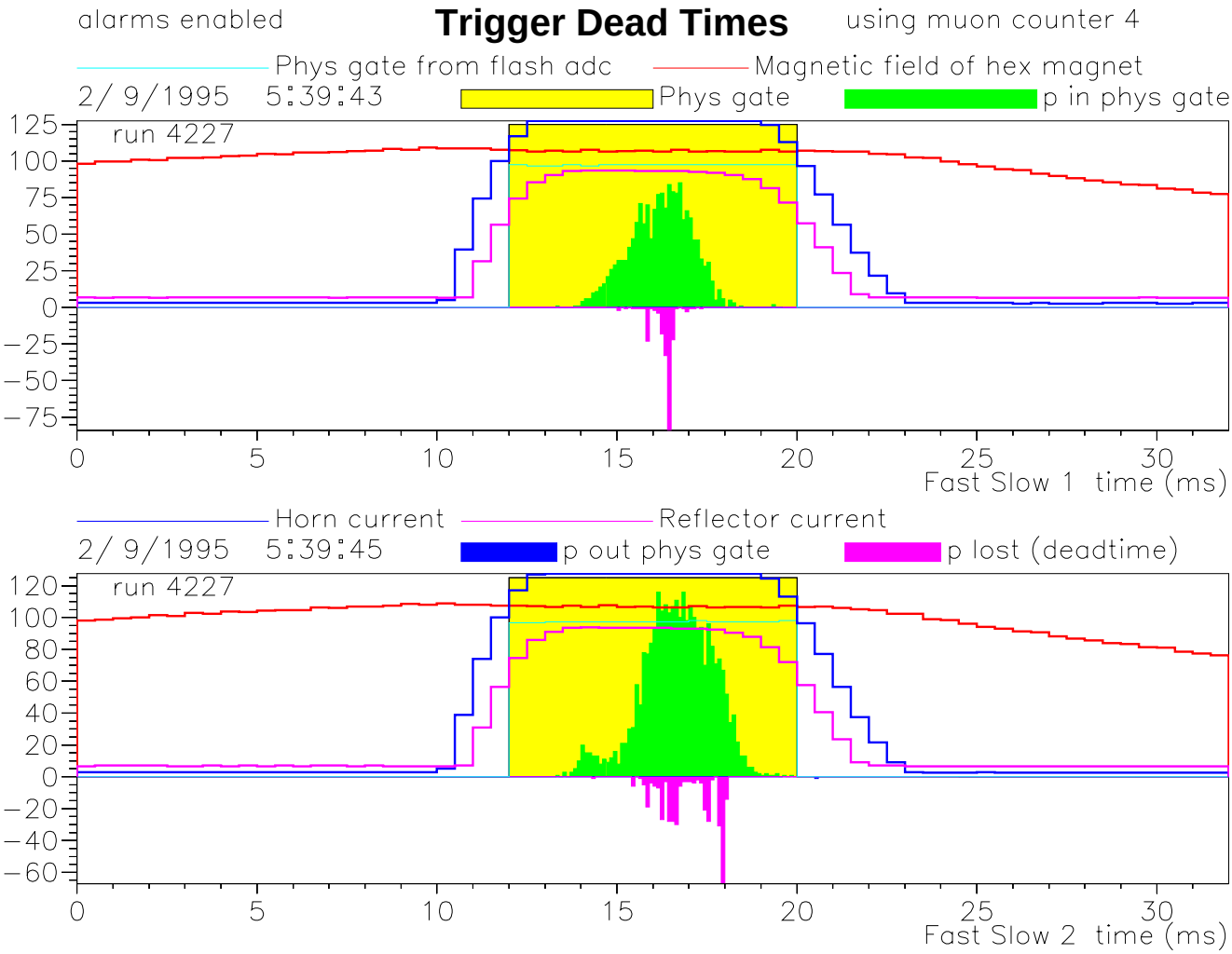

Figure 9: The neutrino burst timing and structure. The shaded rectangle containing the burst represents the physics gate. The fraction of the burst lost due to dead-time is shown upside down below the zero line. The picture also displays measurements of the currents in the Horn and the Reflector, two crucial magnets in the neutrino beam line: their currents rise steeply just before the physics gate and fall steeply just after it. The remaining curves display measurements of the magnetic field in the hexagonal magnet: it rises slowly before the physics gate and falls slowly after it.

The Slow Control

Closely related to the DAQ is the "SLOW CONTROL" system [21], which monitors hardware parameters that may change during data-taking. These parameters include the low and high voltages of the subdetectors and their readout hardware, temperatures, gas flows etc. The system only allows for monitoring the parameters; subsystem experts use specialized programs when parameters must be modified, after which the SLOW CONTROL database is updated accordingly. For the low-level acquisition of parameter values a FIC-8234 is used running OS-9. The SLOW CONTROL crate contains various VME-based monitoring modules, like CAENET [18] high voltage controllers, which are read out directly. There is also a G-64 [22] crate housing a CERN-built monitoring system previously used in the CHARM II experiment [24]. That system is based on voltage-to-frequency converters and pulse counters. Besides G-64 I/O modules

the crate contains a CERN-built VME 3U adapter casing, which holds an MC-68020 based processor board and an Ethernet card [23]. Also this processor boots OS-9 from the file server. The UNIX high-level part of the SLOW CONTROL then starts a process on the G-64 host to deal with reading the G-64 I/O modules, under control of the similarly started process running on the FIC-8234. The DISPATCHER on the file server is used to pass control commands and status information between the three parties, and to send the measured values to the UNIX client, which records the status of all the 4589 monitored channels every two minutes, and updates its display with an alarm indication whenever a channel's value lies outside its allowed range. The history of all channels is archived on disk, and the client allows easy viewing of trend plots which go back up to a month.

An optical alignment monitoring system, RASNIK [25], was introduced with the Honeycomb subdetector in 1996. The alignment data are collected and first processed by a dedicated PC ("RASNIK" in Fig. 2), which then transfers the results through FTP to the DAQ file server, where they are further processed and archived. The SLOW CONTROL expects to receive a status message at regular intervals, indicating that all is well with the alignment data, otherwise it displays an alarm for the RASNIK. Similarly the SLOW CONTROL itself has to send a message to the PANEL at regular intervals, otherwise the latter would change the color of its "SLOW CONTROL" "status light" from green to red (see Sect. 10), indicating a potentially serious problem.

For the 1998 run (see Sect. 10) the SLOW CONTROL has been modified to play an active role. First, each channel was assigned a level of importance. Then, whenever for a period of 15 minutes the consecutive readings of some unreported channel were all lying outside its allowed range, a small numeric message would be sent to the pager of the corresponding subsystem expert. The message gave the number of bad channels and the highest level of importance among them. This allowed the expert to judge what delay would be permissible before coming in to fix the problem. The 15 minute accumulation period served to filter out occasional misreadings caused by the SLOW CONTROL hardware itself.

The Beam

For precise measurements of the neutrino beam flux and structure, data are collected from detectors operated by the SPS West Area Neutrino Facility group, who use a DAQ system controlled by the commercial product FactoryLink [26]. A workstation ("WANF" in Fig. 2) running a graphical interface to FactoryLink updates various histograms and profiles, providing monitoring facilities for the neutrino beam performance. An RPC server makes the recorded beam data available to the CHORUS beam histogram task, which archives the data for analysis purposes and calculates various distributions, which can be viewed through the ONLINE_MONITOR.

12 Conclusions

We have described the Data Acquisition system used in the CHORUS experiment in the years 1994–1998. The performance of the DAQ has been satisfactory with an efficiency of more than 99 %. The DAQ has turned out to be versatile, in part due to its novel implementation of important concepts, including: a remote operating system (REMOS), a finite state machine language (CHSM), a buffer manager, a dispatcher, a control panel and a supervisor. The descriptions of these concepts and of their interplay may serve in the design of new data acquisition systems.

13 Acknowledgements

The realization and operation of the Data Acquisition of the CHORUS experiment was possible thanks to the skill and dedication of many people. In particular we wish to thank: D. Bourillot, F. Cataneo, J. Dupraz, E. Falk, G. Roiron, L. Thibaudeau. Furthermore, we gratefully acknowledge the financial support of the many funding agencies, in particular: the German Bundesministerium für Bildung und Forschung, the Istituto Nazionale di Fisica Nucleare (Italy), the Foundation for Fundamental Research on Matter FOM and the National Scientific Research Organization NWO (the Netherlands). Finally, we wish to thank our referees R. van Dantzig (CHORUS) and F. Formenti (CERN) for their careful reading of the drafts of this document and their precise suggestions for enhancements.

References

[1] E. Eskut et al. (CHORUS collaboration), The CHORUS Experiment to search for $\nu_\mu \to \nu_\tau$ Oscillation, Nucl. Instr. Meth. A 401 (1997) 7-44;
E. Eskut et al. (CHORUS collaboration), Search for $\nu_\mu \to \nu_\tau$ oscillation using the tau decay modes into a single charged particle, Phys. Lett. B 434 (1998) 205-213.

[2] G. Rosa et al., Automatic analysis of digitized TV images by a computer driven optical microscope, Nucl. Instr. Meth. A 394 (1997) 357;
S. Amendola et al., Status of Salerno Laboratory (Measurements in Nuclear Emulsion), Proc. The First International Workshop of Nuclear Emulsion Techniques (12-24 June 1998, Nagoya, Japan), hep-ex/9901034.

[3] P. Annis et al. (CHORUS Collaboration), The CHORUS scintillating fiber tracker and opto-electronic readout system, Nucl. Instr. Meth. A 412 (1998) 19–37;
P. Annis, The CHORUS scintillating fiber tracker and its monitoring systems, Nucl. Instr. Meth. A 409 (1998) 629–633.

[4] OS-9 (version 2.4): Microware Systems Corporation, Des Moines, Iowa, USA; http://www.microware.com.

[5] CETIA Inc., Burlington, MA, USA; http://www.cetia.com.

[6] J.W.E. Uiterwijk et al., The CHORUS honeycomb tracker and its bitstream electronics, Nucl. Instr. Meth. A 409 (1998) 682–686.

[7] VICbus, VIC-8251 VME master/slave (16 MB), VCC-2117 CAMAC crate controller: CES, Petit-Lancy, Switzerland; http://www.ces.ch.

[8] ELTEC Elektronik, Mainz, Germany; http://www.eltec.com.

[9] G. Carnevale et al., "REMOS: a portable object oriented environment for multiprocessor real time applications," Procs. of CHEP'94, San Francisco (1994), Univ. of California, Berkeley, LBL 35822, 311-314;
G. Carnevale et al., "The CHORUS data acquisition system," Procs. of CHEP'94, San Francisco (1994), Univ. of California, Berkeley, LBL 35822, 101-103;

G. Carnevale, Laurea Thesis, Università "Federico II," Naples (1995);
F. Riccardi, Ph.D. Thesis, Università "Federico II," Naples (1996).
[10] M.G. van Beuzekom et al. , The trigger system of the CHORUS experiment, CERN-EP-98-131, Nucl. Instr. Meth. A 427 (1999) 587-606.
[11] P.J. Lucas, "An Object-Oriented Language System for Implementing Concurrent, Hierarchical, Finite State Machines," M.Sc. Thesis, Univ. of Illinois at Urbana-Champaign (1993), UIUCDCS-R-94-1868; http://www.best.com/∼pjl;
F. Riccardi, Ph.D. Thesis, Università "Federico II," Naples (1996);
F. Riccardi et al., "OO-CHSM: Integrating C++ and Statecharts," Procs. of CHEP'95, Rio do Janeiro, Brazil, September 1995.
[12] YACC/Bison: available on most UNIX derivatives.
[13] P. Annis et al. (RD46 Coll.), A new vertex detector made of glass capillaries, Nucl. Instr. Meth. A 386 (1997) 72;
S. Buontempo et al. (RD46 Coll.), The Megapixel EBCCD: a high-resolution imaging tube sensitive to single photons, CERN–EP/98–36 (1998).
[14] R. Gurin (ruten.gurin@caspur.it) and A. Maslennikov (andrei@caspur.it), "ControlHost – Distributed Data Handling Package," User Manual, CASPUR Program Library Note C001, Rome, Italy (1995);
"ControlHost: field-tested package for Distributed Data Handling," HEPiX95 Meeting, Institute of Physics of the Academy of Sciences (FZU), Prague, Czech Republic, 31 May to 2 June 1995;
http://www-hep.fzu.cz/computing/HEPiX/HEPiX95/talks/conhost.ps.
[15] MM-6326AD: Micro Memory Inc., Chatsworth, CA, USA; http://www.micromemory.com.
[16] Struck VSC 1000: Struck Innovative Systeme, Hamburg, Germany; http://www.struck.de.
[17] J.K. Ousterhout, Tcl and the Tk Toolkit, Addison-Wesley (1994).
[18] CAEN, Viareggio, Italy; http://www.caen.it.
[19] D. Libes, Exploring Expect, O'Reilly & Associates (1995).
[20] CERNLIB: CERN, CH–1211 Geneva 23, Switzerland; http://www.cern.ch.
[21] D. Bonekämper, Dipl. Thesis, Westfälische Wilhelms-Universität, Münster, Germany (1993).
[22] Gespac, Geneva, Switzerland; http://www.gespac.com.
[23] VM20 and VLAN: PEP Modular Computers, Kaufbeuren, Germany; http://www.pep.com.
[24] J.P. Dewulf et al. (CHARM II Coll.), Nucl. Instr. Meth. A 252 (1986) 443;
J.P. Dewulf et al. (CHARM II Coll.), Nucl. Instr. Meth. A 263 (1988) 109;
D. Geiregat et al. (CHARM II Coll.), Nucl. Instr. Meth. A 325 (1993) 92;
T. Bauche, Ph.D. Thesis, University of Hamburg (1988).
[25] J. Dupraz et al. , The optical alignment monitoring system of CHORUS (RASNIK), Nucl. Instr. Meth. A 388 (1997) 173.
[26] E. Heijne, CERN Yellow Report 83-06 (1983);
G. Acquistapace et al. , CERN–ECP/95–14 (1995).