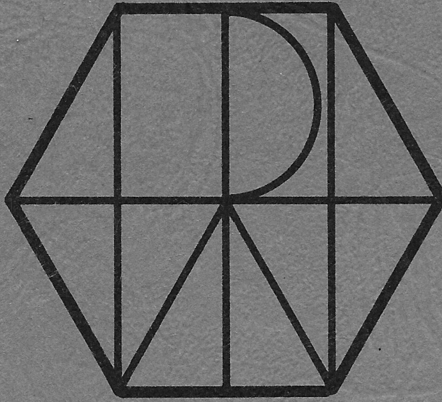


PITHA 93/46

Network and Process Communication for the H1 Experiment at HERA

Von der Mathematisch– Naturwissenschaftlichen Fakultät
der Rheinisch– Westfälischen Technischen Hochschule Aachen
genehmigte Dissertation zur Erlangung des akademischen Grades
eines Doktors der Naturwissenschaften

von

Diplom– Physiker

Patrick Fuhrmann

aus Neuwied

PHYSIKALISCHE INSTITUTE
RWTH AACHEN
Sommerfeldstr.
51 AACHEN, FR GERMANY

Network and Process Communication for the H1 Experiment at HERA

Von der Mathematisch- Naturwissenschaftlichen Fakultät
der Rheinisch- Westfälischen Technischen Hochschule Aachen
genehmigte Dissertation zur Erlangung des akademischen Grades
eines Doktors der Naturwissenschaften

von

Diplom- Physiker

Patrick Fuhrmann

aus Neuwied

Referent	:	Universitätsprofessor Dr. Ch. Berger
Korreferent	:	Universitätsprofessor Dr. K. Lübelsmeyer
Tag der mündlichen Prüfung	:	1.10.1993

Contents

1 Introduction . . . 3
1.1 The Startup Phase of HERA . . . 3
1.2 The expected Event Classes . . . 3

2 The Data Source . . . 8
2.1 An Overview . . . 8
2.2 The Subdetectors . . . 8
2.3 The Trigger Scenario . . . 11
2.3.1 The different Trigger Levels. . . . 11
2.3.2 Bottlenecks and Deadtimes . . . 12

3 What to do with all this data . . . 15
3.1 The Central Data Aquisition, CDAQ . . . 15
3.2 The Data Logging Unit . . . 16
3.3 The Reconstruction . . . 17
3.4 The Analysis . . . 17

4 The Problem and the L4 Filter Farm . . . 20
4.1 The Requirements . . . 20
4.2 The Solution . . . 20
4.2.1 The Hardware . . . 21
4.2.2 The System Software . . . 22
4.2.3 The Filter Software . . . 23

5 The Details . . . 25
5.1 The Uniform VME Addressing . . . 25
5.1.1 The Hardware Connections . . . 25
5.1.2 The VME Addressing Concept . . . 25
5.2 A Relay Race . . . 27
5.3 The RAID Program . . . 29
5.4 Information Exchange between inner System and Rest of VME . . . 32
5.5 A Pointers Paradise . . . 32
5.6 Informations for the Filter Program . . . 32
5.7 Informations from the Filter Program . . . 32
5.7.1 The Histograms . . . 32
5.7.2 The Monitorinformations . . . 33
5.7.3 The Events . . . 33
5.8 The Environment Simulation . . . 34

6 A general Data Distribution Concept . . . 38
6.1 The Network . . . 38
6.2 The Online Event Servers . . . 40
6.3 The passive Farm VME OS . . . 41
6.3.1 The VME Memory Allocation . . . 41
6.3.2 The VME Task Monitor . . . 41
6.3.3 Again the Event Server . . . 42
6.3.4 The Unification . . . 42
6.4 The Farm Startup . . . 43
6.4.1 The Installation Language . . . 43

6.4.2 The Startup Procedure . . . 45
6.4.3 The Database Access . . . 45
6.5 The H1 LAN . . . 46
6.6 The Alternative . . . 47

7 The Filter Program **48**
7.1 The Module Steering File . . . 48
7.2 The Modules . . . 48

8 The Performance **52**

A Network Details **55**
A.1 TCP/IP and the OSI Reference Model . . . 55
A.2 A Protocols Framework . . . 56
A.3 The N-Net . . . 57
A.4 The H1 LAN . . . 58
A.5 The Remote Server Request Protocol . . . 58
A.6 The Net - Event Request Protocol . . . 60
A.7 The Farm Event Server Protocol . . . 61

B The Passive Farm OS **62**
B.1 The Task Monitor . . . 62
B.2 The VME Memory Management Part . . . 63
B.3 The External Requests for the I/O Processors . . . 64

List of Figures

1 The Energy Deposit . . . 6
2 The Neutral Current Candidate . . . 7
3 The Data Flow Channels . . . 13
4 Level 1 to level 3 Trigger Timing . . . 14
5 Event Builder Layout . . . 15
6 Current Dataflow of the Reconstruction . . . 18
7 Possible future Dataflow of the Reconstruction . . . 19
8 The RAID 8235 Block Diagram . . . 21
9 Physical Farm layout . . . 23
10 Logical Farm VME configuration . . . 26
11 Fast Dataflow Protocol . . . 28
12 State Transition Diagram for the RAID boards . . . 31
13 Pointer Layout . . . 36
14 Farm Event Request Protocol . . . 37
15 Event Reduction according to the Modules . . . 51
16 The Rejection Pie . . . 51
17 Input Event Length and total Execution Time . . . 53
18 relative process rates for 1, 2 and 3 I/O Buffers . . . 54
19 Performance for 2 Input Processors . . . 54
20 The Farm OS Kernel . . . 67
21 The Farm OS Structures . . . 68
22 The External Request Union . . . 69

1 Introduction

1.1 The Startup Phase of HERA

On October 19, 1991, the two detectors **H1** and **ZEUS** at the **HERA** collider were able to monitor the first electron - proton interactions caused by one electron and one proton bunch. The energy of the electrons was fixed to 26.6 GeV and for technical reasons the energy of the proton limited to 480 GeV. About 8 months later, at the end of June '92 the continuous experimental operation started with two times ten bunches and the designed energies for the protons of 820 GeV and 26.6 GeV for the leptons. The average luminosity for that setup was $L = 2 * 10^{28} cm^{-2} sec^{-1}$ with peaks in the order of three times that value. The design luminosity achievable with two times 210 bunches was calculated to be $L_{design} = 10^{31} cm^{-2} sec^{-1}$. From end of June until the shutdown at the beginning of August, the effective usable time for the experiments accumulated to 20% of the total time, with an integrated luminosity in the order of $30 nb^{-1}$.

1.2 The expected Event Classes

What kind of events can be expected to be found with the H1 microscope? The physics of inclusive inelastic e-p scattering is completely determined by two variables :

- $Q^2 = -q^2 = -(e_{in} - e_{out})^2$ which is the 4-vector transfer squared between the in and outgoing leptons. This can be interpreted as the virtual mass of the exchanged gauge boson.
- $y = (p_{in} \cdot q)/(p_{in} \cdot e_{in})$, the relative energy loss of the electron in a frame where the proton is in rest.

If the electron could be positively identified, which means, the scattering angle θ_e and its energy $E_{e'}$ is known, the calculation of Q^2 and 'y' reduces to

$$\begin{aligned} Q^2 &= 4\, E_e\, E_{e'} \cos^2(\tfrac{1}{2}\,\theta_e) \\ y &= 1 - (E_{e'}/E_e) \sin^2(\tfrac{1}{2}\,\theta_e) \end{aligned}$$

where θ_e is defined to be zero for particles going into the initial proton direction. Theoretically the experiment covers an energy transfer in the range from

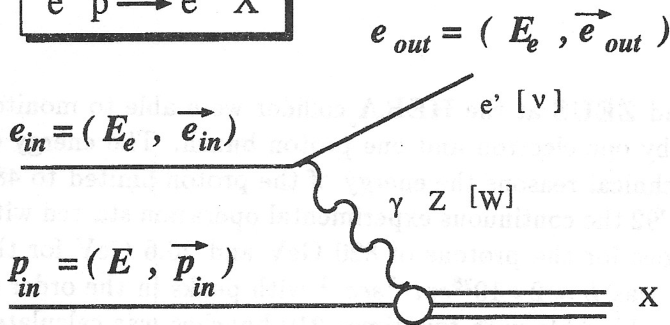
$$\begin{aligned} Q^2_{min} &= \frac{m_e^2 y^2}{1-y} \approx 5 * 10^{-8}\, GeV^2 \quad \text{up to} \\ Q^2_{max} &= s = 4\, E_p\, E_e \approx 0.87 * 10^5\, GeV^2 \end{aligned}$$

That means, in one experimental setup, physics can be studied from photoproduction up to very deep inelastic scattering.

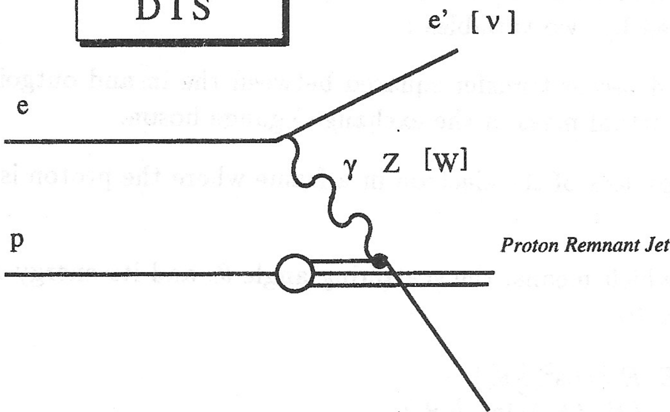
Photoproduction is defined by the limit $Q^2 \rightarrow 0$, whereas at high Q^2 also the exchange of the other gauge bosons, Z_0 (neutral current) and $W^{\pm}$ (charged currents) has to be taken into account. The photoproduction resembles the by far biggest cross section and therefore the majority of triggered events. Even with the restricted acceptance of the e^- tagging system

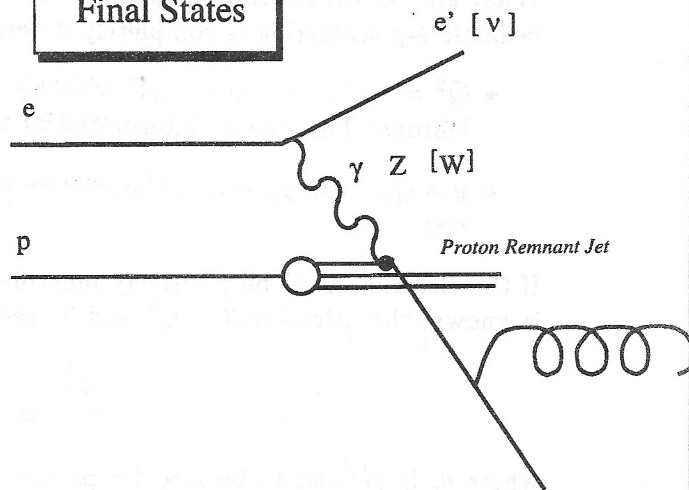
$$\begin{aligned} 0.2 &< y < 0.8 \quad \text{and} \\ 3 * 10^{-8} &< Q^2 < 10^{-2}\ GeV^2 \end{aligned}$$

the visible cross section is $43.6\,\mu b$, i.e. $4 \cdot 10^9$ events per year for $100 pb^{-1}$.

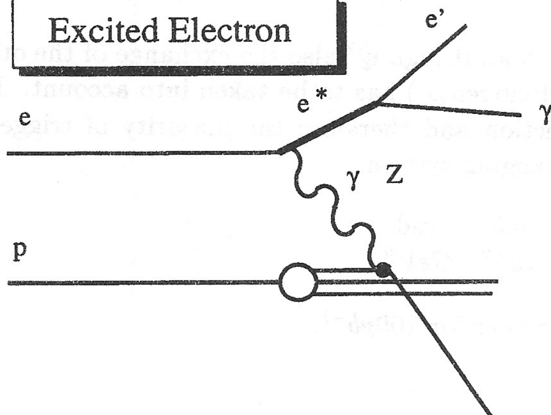


From the measured photoproduction[H1-1 93] of $\sigma_{tot}(\gamma p)$ = 159 ± 7(stat.) ± 20(syst.)μb for an average of the center of mass energy of the $\gamma - p$ system of 195 GeV up to the cross section of the deep inelastic scattering[H1-2 93], which amounts to σ_{DIS} = 92 ± 11(stat.) ± 12(syst.)nb for $Q^2 > 5\,GeV$, there is a difference in the order of a factor 2000. This rate is reduced by another factor if one investigates jet production in DIS, because the radiation of an extra gluon charges the cross section by roughly a factor α_s.





The weak cross section at HERA (exchange of a Z_0 or $W^{\pm}$) are rather small ($\approx$ 0.02nb). Therefore for a charged current reaction only one candidate has been seen in the H1 detector(see Figure 2). Extreme low event rates are expected for the detection of event classes, not described in the standard model, which is of course one of the most favoured aims of a new collider.



For HERA this could be the detection of *Leptoquarks*, *Leptogluons* and *Excited Leptons*. With $100pb^{-1}$ and no events seen, one can exclude a production cross section of $3\,/\,L \approx 3 \cdot 10^{-2}pb$ at 95%

C.L. Thus the physics at HERA covers 7 orders of magnitude in cross section.

For the different H1 trigger levels, that means a controlled reduction of the photoproduction without touching the rare events classes. The question is, whether this can be achieved by simple hardware devices or with more sophisticated algorithms running on online processors. The H1 solution, described in the following, is a well tuned mixture of both.

An even more disturbing type of events, which superimposes the raw detection rate with more than 95% is the so called beam gas class, the collision of beam electrons or protons with the residual beamgas nucleus. The electron type simulates a normal $e+p \rightarrow e+X$ reaction with a CMS energy of $\sqrt{s} \approx 8.2\,GeV$ and the proton type looks similar to photoproduction where the electron disappeared in the beampipe and a CMS energy is in the order of $\sqrt{s} \approx 39\,GeV$. These events are background in respect to all other event classes HERA actually was designed for, namely with a CMS energy of 295 GeV. Beam gas events are dominated by p-gas reactions. The products have a large*Lorentz boost* in the forward direction and therefore the main energy flow is seen in that part of the detector(see Figure 1). Again a mixture of fast hardware triggers followed by special online processing algorithms lead to successful suppression of the background events.

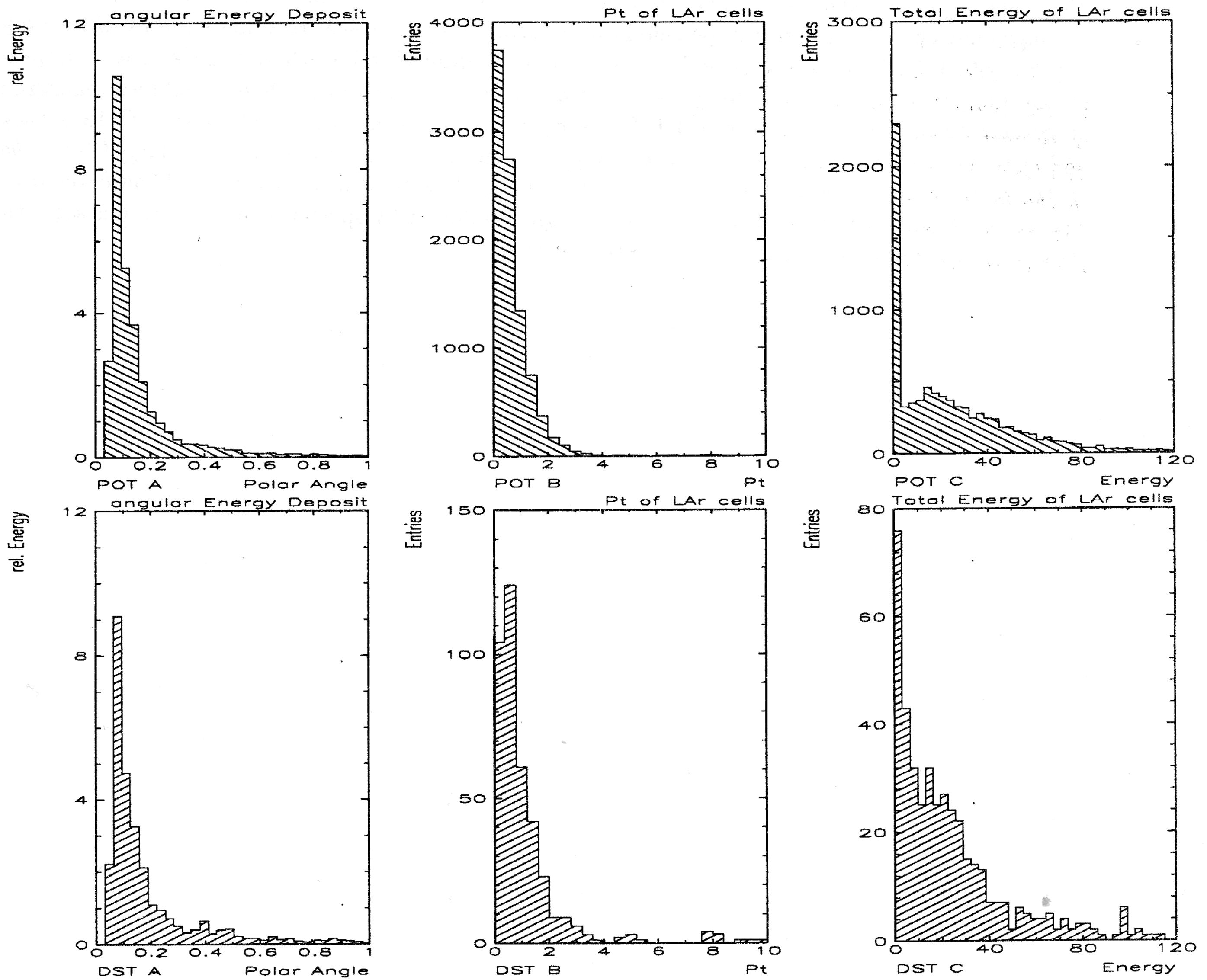


Figure 1: The Histograms show the energy deposits in the H1 detector for an arbitrary run (33846). In the upper row, the data contains a large amount of beam gas background, which is suppressed in the second row. For all pictures only the Liquid Argon Calorimeter had been taken into account. From left to right, the diagrams show **A** : an energy weighted histogram of all cells hit during the run, **B** : a histogram of the total p_t of the event, and **C** : a histogram of the total energy per event.

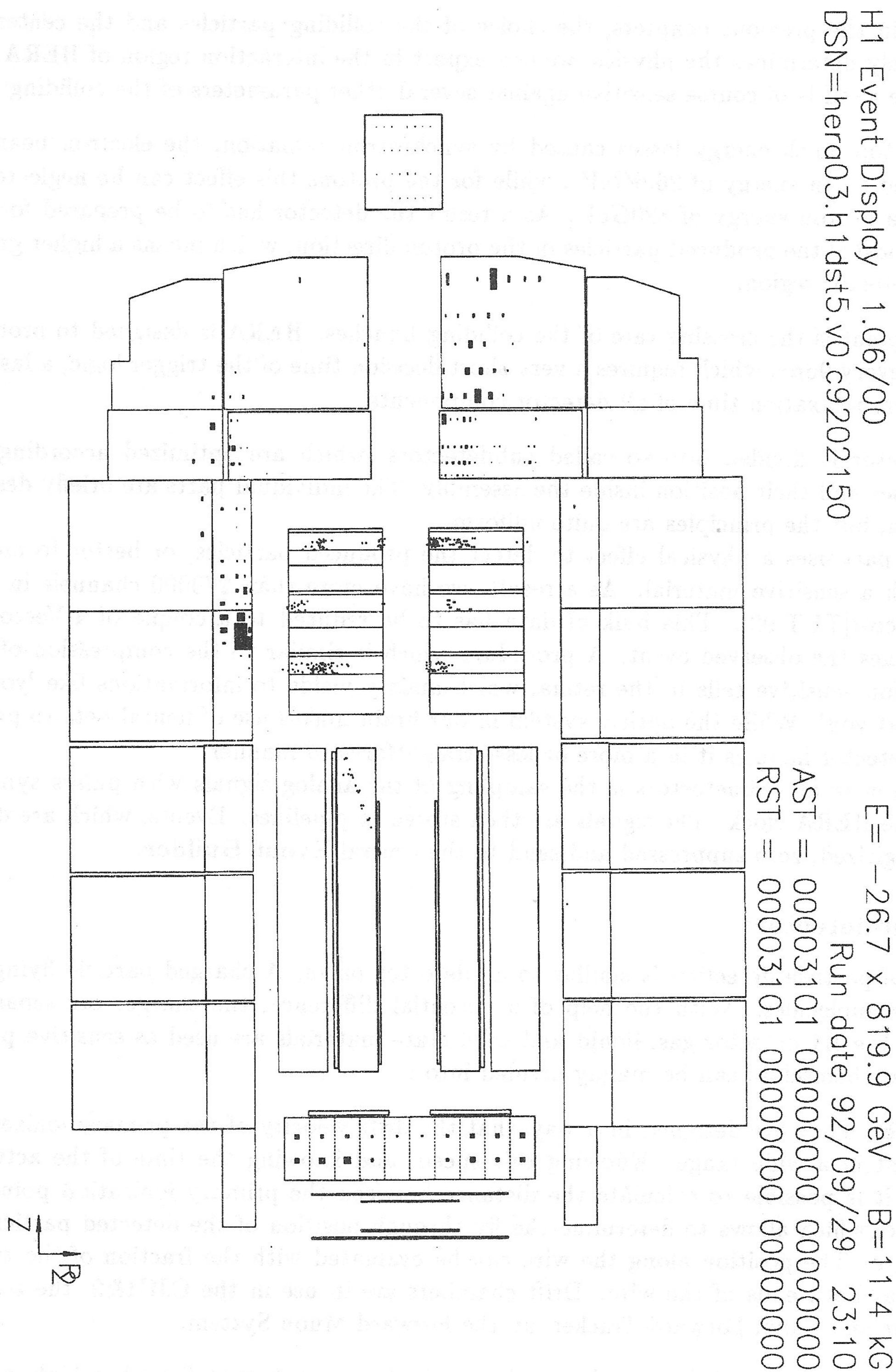


Figure 2: The picture shows the only Neutral Current Candidate up to the shutdown in winter '92. The neutrino is assumed to have left the detector in the top direction.

2 The Data Source

2.1 An Overview

As we learned in the previous chapters, the choice of the colliding particles and the center of mass energy exclusively determines the physics, we can expect in the interaction region of HERA. But the detection device itself is of course sensitive against several other parameters of the colliding machine.

- To avoid too much energy losses caused by synchrotron radiation, the electron beam is only accelerated to an energy of $26.6GeV$, while for the protons this effect can be neglected, which yields to a proton energy of $820GeV$. As a result the detector has to be prepared for a strong *Lorentz boost* of the produced particles in the proton direction, which means a higher granularity in that detector region.
- Another point is the crossing rate of the colliding bunches. HERA is designed to produce such an event every $96ns$, which requires a very short decision time of the trigger logic, a fast readout and a short relaxation time of all detector components.

The total detector is divided into so called subdetectors, which are optimized according to their physical purposes and their position inside the assembly. The individual parts are briefly described in the next section, but the principles are quite uniform.

A sensitive part uses a physical effect to detect the produced particles, or better to measure an interaction with a sensitive material. As a result, we have more than 270000 channels in total, for the whole detector[TUT 92]. This bulk of data has to be reduced to a couple of 4-Vectors, which sufficiently defines the observed event. A procedure which is similar to the compression of the data of the 120 million sensitive cells in the retina, which mainly yields to informations like 'you can eat it', or 'it will eat you'. While the optical system in our brain makes use of neural nets to process the data, the H1 detector handles it in a more or less straightforward manner.

Most common to all subdetectors is the sampling of the analog signals with pulses synchronized according to the HERA clock. The signals are then stored in pipelines. Events, which are decided to be good, are digitized, zero suppressed and send to the central **Event Builder**.

2.2 The Subdetectors

The principle of particle detection is similar to all detector parts. A charged particle flying through material ionizes molecules. With the help of a potential difference, the charges are separated and measured. For the H1 detector gas, liquid and solid state materials are used as sensitive parts. The gas type devices themselves can be mainly divided into :

Drift Chamber They are designed in a way, that the drift velocity of the primary ionized particle is constant in a wide range. Knowing this speed, and knowing the time of the actual bunch crossing, it is possible to calculate the distance between the primary ionization point and the sense wire, which allows to determine the fly through position of the detected particle vertical to the wire. The position along the wire can be evaluated with the fraction of the two charge portions at both ends of the wire. Drift chambers are in use in the CJC1&2, the z-chambers, the planar and radial Forward Tracker, an the Forward Muon System.

Streamer Tubes Streamer Tubes are designed quit similar, except, that there is a high gas amplification near the sense wire. This saves the electronic preamplification of the sense wire signals.

Multi Wire Proportional Chambers MWPC are small sized gas chambers. This geometry results in a very short drift time. Consequently they are mainly used for trigger purposes.

The Calorimeter is realized as condensers in a **Liquid Argon** bath. The advantage is a 10^3 times larger density than in gases, which leads to an increase of dE / dx in the same order. As a result, nearly the total expected energy can be detected inside the calorimeter, and the shower shapes are small enough to distinguish nearby particle tracks.

The **Backward Electromagnetic Calorimeter** is realized as solid state detector. Escaping particles build electromagnetic showers inside a small lead plate. A scintillator material converts the produced electrons into low energetic light . This light is converted in wavelengthshifters to the optimal detection wavelength of the 472 linked photodiodes.

Some facts of the most important detector parts. (see also Figure 3)

Central Tracker CT The Central Tracker is divided into an inner and an outer part. Both containing MWPCs for trigger purposes, chambers for the x-y-detection (CJC) and chambers for the z-direction. From inside to outside the components are :

Inner MWPC – Inner Z – CJC1 – Outer Z – Outer MWPC – CJC2

In total, it covers an angle of about $25° \leq \theta \leq 155°$.

inner and outer Multi Wire Proportional Chambers CIP,COP

- provide fast ray trigger and vertex trigger.
- 3936 Channels.

inner and outer z - drift Chambers CIZ,COZ

- 60 wires for inner an 96 wires for the outer z-chamber.
- z - measurement accuracy in the order of $350\mu m$ for single and $3mm$ for double track resolution.
- $r - \phi$ resolution is about 25 mm for the inner and 58 mm for the outer z-chamber

inner MWPC and z - Chamber CIP,CIZ and COZ

- allows quick calculation of the vertex position along the beam.
- is a part of the first level trigger.

inner and outer Jet Chambers CJC1 and 2

- 2640 sense wires.
- provide good measurement of the $r - \phi$ coordinates in the order of $210\mu m$.
- readout at both ends for charge division and dE / dx measurement, which leads to a z-resolution of about 23.5 mm.

The Forward Tracker FT

- Accurate measurement of charged particle tracks in forward direction.
- e/π separation to supplement the information of the calorimeter.
- fast ray trigger by means of MWPCs.
- 3 supermodules à one Transition Radiator, one radial and one planar Drift Chamber.
- radial chambers divided into 48 cells in ϕ à 12 sense wires in z, which means 1728 in total.
- planar chambers has 1152 sense wires in total.
- planar chamber resolution $\theta \leq 1mrad$
- 2048 channels in total for Forward Tracker.

The Forward Muon Detector FM

- high precision drift chambers.
- covers an azimuth angle of $3° \leq \theta \leq 17°$
- $\theta - \phi$ track resolution of about 0.5 mrad.
- Momentum precision in the order of 30 % in the energy range between $1.5GeV$ and $150GeV$,

The Liquid Argon Calorimeter LAr

- 45,000 readout channels are multiplexed onto 55,000 channels to increase the dynamic range of a quarter of them.
- covers an azimuth angle of $4° \leq \theta \leq 155°$
- divided into an electromagnetic and hadronic part.

The Backward Electromagnetic Calorimeter BEMC

- BEMC closes calorimeter angle $150° \leq \theta \leq 176°$ for electromagnetic energy flow
- lead/scintillator sampling calorimeter with 472 photodiodes.
- Energy resolution is $\frac{\sigma}{E_e} = \frac{10\%}{\sqrt{E_e}}$

The Forward Plug Calorimeter PLUG

- Hadronic Calorimeter for the extreme forward direction $12.5mr <= \theta <= 60mr$.
- uses small silicon wafers where the pn-structure is realized by metal semiconductor surface barrier.

The Instrumented Iron Calorimeter IRON and the **Central Moun Detector**

- plastic tube gas counters operating in limited streamer mode for the central and endcap detector.
- 160,000 output channels in total
- used for the measurement of the energy leaking not covered by the inner calorimeters,
- and for the muon tracking.

The electron tagger and the luminosity counter ET

- TlCl/TlBr cristal calorimeter with an energy resolution of $\sigma(E)/E = 0.1/\sqrt{E}$ and a space resolution of $\sigma(x,y) = 0.2cm$.
- The electron tagger is sensitive in the energy range between $0.2E^{in}_{electron}$ and $0.8E^{in}_{electron}$ and an angle below 5 mrad.
- The luminosity counter consists of one e^- and one γ detector near the beam pipe, to catch the e^- and the γ of the bremsstrahlung reaction $e^- p \rightarrow e^- p \gamma$ for luminosity monitoring.

All the subdetector data is condensed to 11 so called branches, the normalized interface to the Central DAC.

1	Central Trigger
2	Calorimeter Trigger
3	Calorimeter Analog Digital Converter
4	Central Tracker
5	Forward Tracker
6	Forward Muon
7	MWPC
8	Luminosity
9	Forward Muon Trigger
10	$DC\ r - \phi$ Trigger
11	A Test Branch

2.3 The Trigger Scenario

The HERA collider is designed to store 210 electron and 210 proton bunches. With a total ringlength of 6.4 Km , this results in the already mentioned $96ns$ time interval between the bunch crossings. Together with the other parameters of HERA, like total current and the density of the bunches , one can expect a luminosity of about

$$L = n_e * n_p * f/A \approx 10^{31} cm^{-2} s^{-1}$$

which leads to a nominal luminosity of some $\int L\,dt = 100\,pb^{-1}$ per year.

Number of electrons per bunch	n_e	=	$3.5 * 10^{10}$
Number of protons per bunch	n_p	=	10^{11}
Effective Area of the bunches	A	=	$0.3mm * 0.017mm$
Bunch Collisions per timeinterval	f	=	$1/96ns$

On the one hand this is a great opportunity for the physicists to obtain a good statistics for various branches of interest. On the other hand this short bunch crossing interval means a challenge for the decision logic and the readout electronics. The interaction between these two parts is described in the following.

2.3.1 The different Trigger Levels.

One important aim of the trigger logic is to avoid deadtimes as much as possible[ELS 87]. To obtain this, at least for the first trigger level, the idea of frontend pipelines was introduced. These pipelines sample the frontend analog signals with the HERA clock. The next table gives an overview of the different types of pipelining.

Subdetector	Sensor type	Pipetype	Sample frequency
LAr Calorimeter	liquid Argon	analog capacitor pipeline	HERA clock
jet-chambers,z-chambers planar and radial forward tracker forward muon	drift chambers	flash ADC sampling	10 * HERA clock
MWPCs, VETO, TOF	proportional chambers	shift registers	HERA clock
instrumented IRON	limited streamer tubes	shift registers	HERA clock

In addition to these sampled quantities the subdetectors provide fast signals with each HERA clock pulse. The main sources are :

Big Towers : Energy sums of the liquid Argon Calorimeter.

Big Rays : Mask of the central proportional chambers.

z-vertex : combination of z-chambers and MWPC.

Muon : hit pattern in the muon chambers.

and much more. These informations are compared with given thresholds, and the boolean results build the about 200 **Trigger Elements**. A progammable logic combines them to 64 **Subtriggers**. These subtriggers can be downscaled or totally disabled, and the result is ORed to build the **Level 1 Trigger Signal**. This game has two time constraints to avoid deadtimes :

- The time interval between a bunchcrossing and the corresponding level 1 trigger signal must not exceed the storage capacity time of the shortest frontend pipeline, which is about two microseconds, and

- the trigger must accept new input informations every 96 *nsec*, which means, that this trigger step has also to be a pipelinelike device.

Picture 4 gives an impression of the total trigger scenario up to level 3, as it will hopefully be in action from summer '93.

On the receipt of the level 1 signal, all subdetector pipelines are stopped, or proceed at most until the corresponding pipeline entry reaches the end of the pipe. In addition, the L2 trigger elements start to be calculated. As long, as there is no **L1 Keep**, the events drop out of the data pipe after the 2 μsec.

Starting with a **L1 Keep** the detector deadtime begins, and the subdetectors await the decision of the level 2 trigger. This level makes use of more sophisticated informations and algorithms and needs approximately 20 μsec. If the positive level 1 decision can be confirmed, a L2keep together with a unique number is sent to the subdetector components. The readout electronics now starts to assemble the event data, assigns it with the L2keep number and copies it into a buffer (MEB[1]) which is accessible by the central data aquisition. After some 800μsec the readout should be finished and the pipelines can be enabled again. To any time during this interval a planed level 3 trigger will be able to stop the readout and to reset the whole electronics.

2.3.2 Bottlenecks and Deadtimes

It is evident, that any type of detection device has a maximum rate, with which it can detect and store data. That is also true for our detector, and this rate is by no means 10 MHz, which would be necessary to store the detector informations of each bunch crossing. That means, that whenever the informationstream reaches a bottleneck, we have to decide which of the data is of interest and which can be thrown away. For us, this can only be decided on the event level.

The very first rate reduction appeared with the 800μsec of the readout. Without trigger, this would yield to an event rate of 1.2 KHz and a deadtime of 100 percent. If we look forward in the aquisition chain, we will see, that the next bottleneck is the assembling rate of the eventbuilder. It can only build about 50 events per second, assuming an average eventsize of 120Kbytes. Because of buffering, the peakrates can be nearly 1 KHz until all buffers a filled. Let us again assume no triggering. That would mean we would fill the buffer with maximum speed and 100 percent deadtime. After that, they would be read out by the eventbuilder. During this time, the detector also appears dead, because there is no memory available to store new data. As a result, the detector would of course produce events again with 50Hz, but the total deadtime would be in the order of 100 percent. As a large deadtime always indicates, that we didn't have the chance to see the good events inbetween, we have to tune the level 1 to level 3 trigger in a way which reduces the average deadtime to a minimum, but gives the optimal event rate of 50Hz. The result would be a deadtime of about 5 percent. Unfortunately it is not true, that we made the optimal trigger choices, if the central DAQ display shows 5 % deadtime, because possibly we cut into tails of distributions or throw away special exotic processes. So, detailed simulations are necessary to be on the safe side.

The same problem will appear again, when we discuss the next bottleneck, the IBM link.

---

[1]Multi Event Buffer

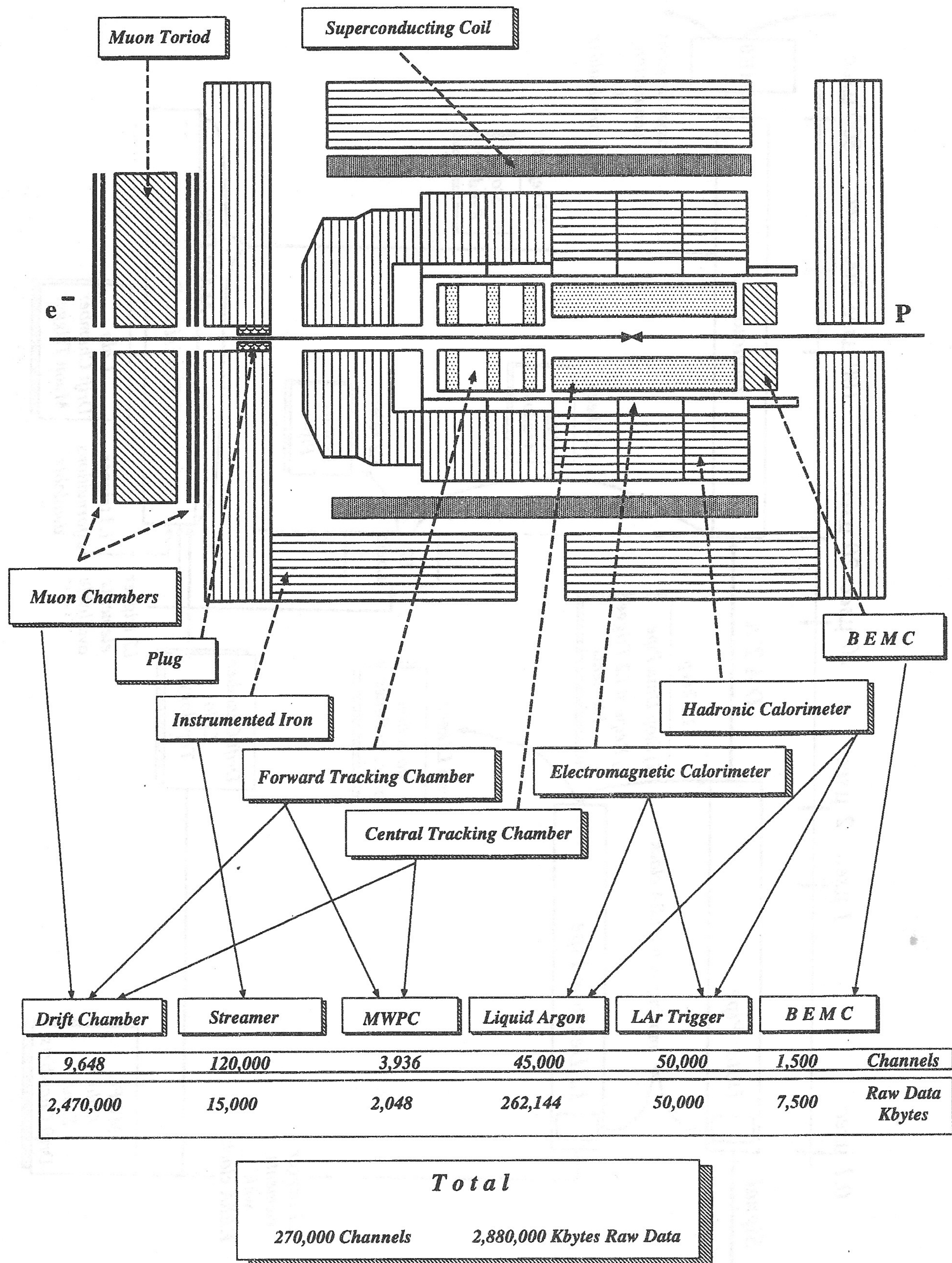


Figure 3: The Data Flow Channels

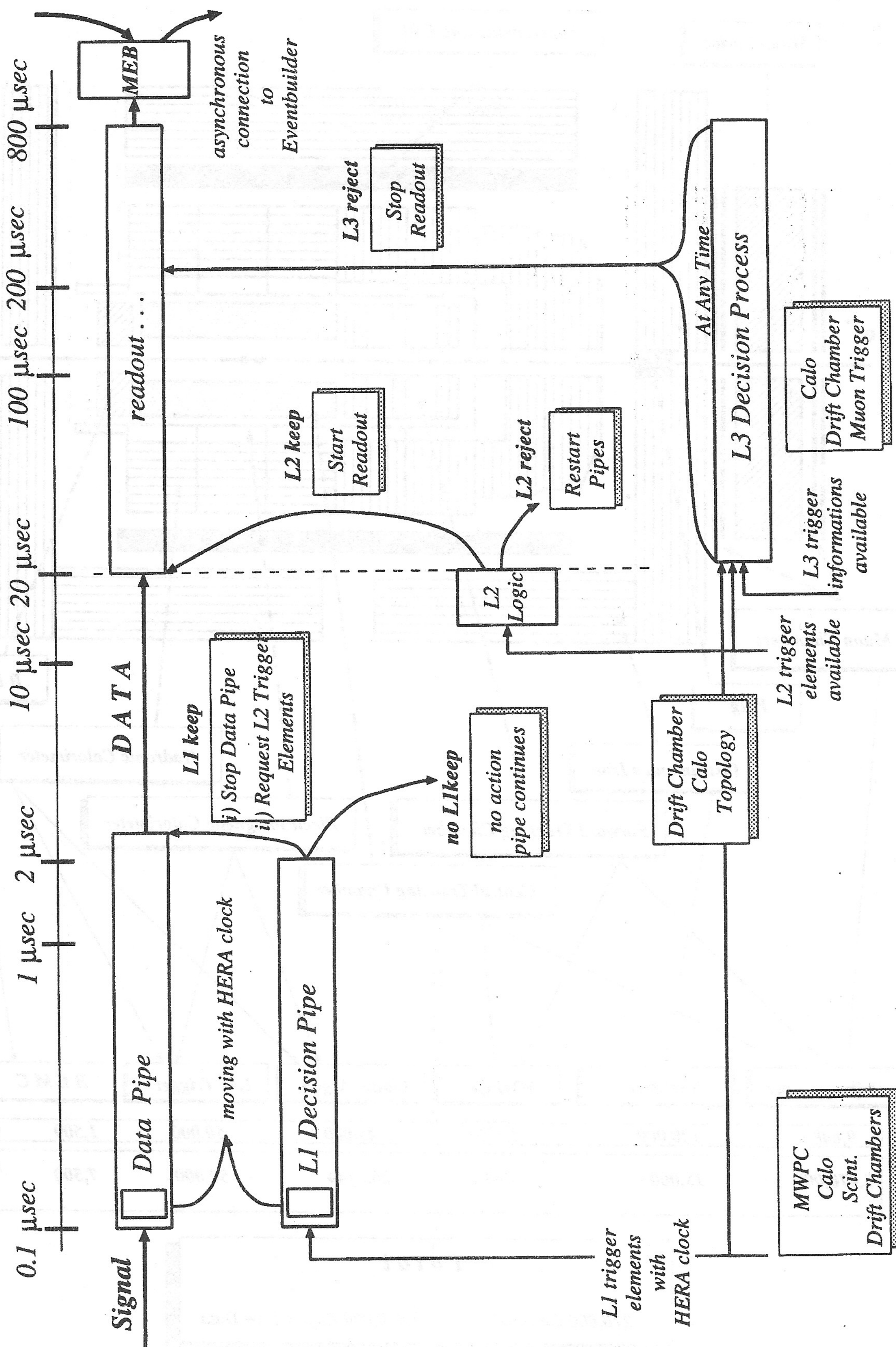


Figure 4: Level 1 to level 3 Trigger Timing

3 What to do with all this data

Until now, we followed the data stream from the production through the frontend pipeline and via the digitization to the Multi Event Buffers, MEBs. The central data aquisition is responsible for the collection of the MEBs, the merging and the transport to the central IBM via a set of *Full Event Tasks.*

3.1 The Central Data Aquisition, CDAQ

The H1 data aquisition system supports a general concept for the connection of the various subdetectors with the event builder(see Figure 5 on page 15). Each of the 11 branches ends up in an

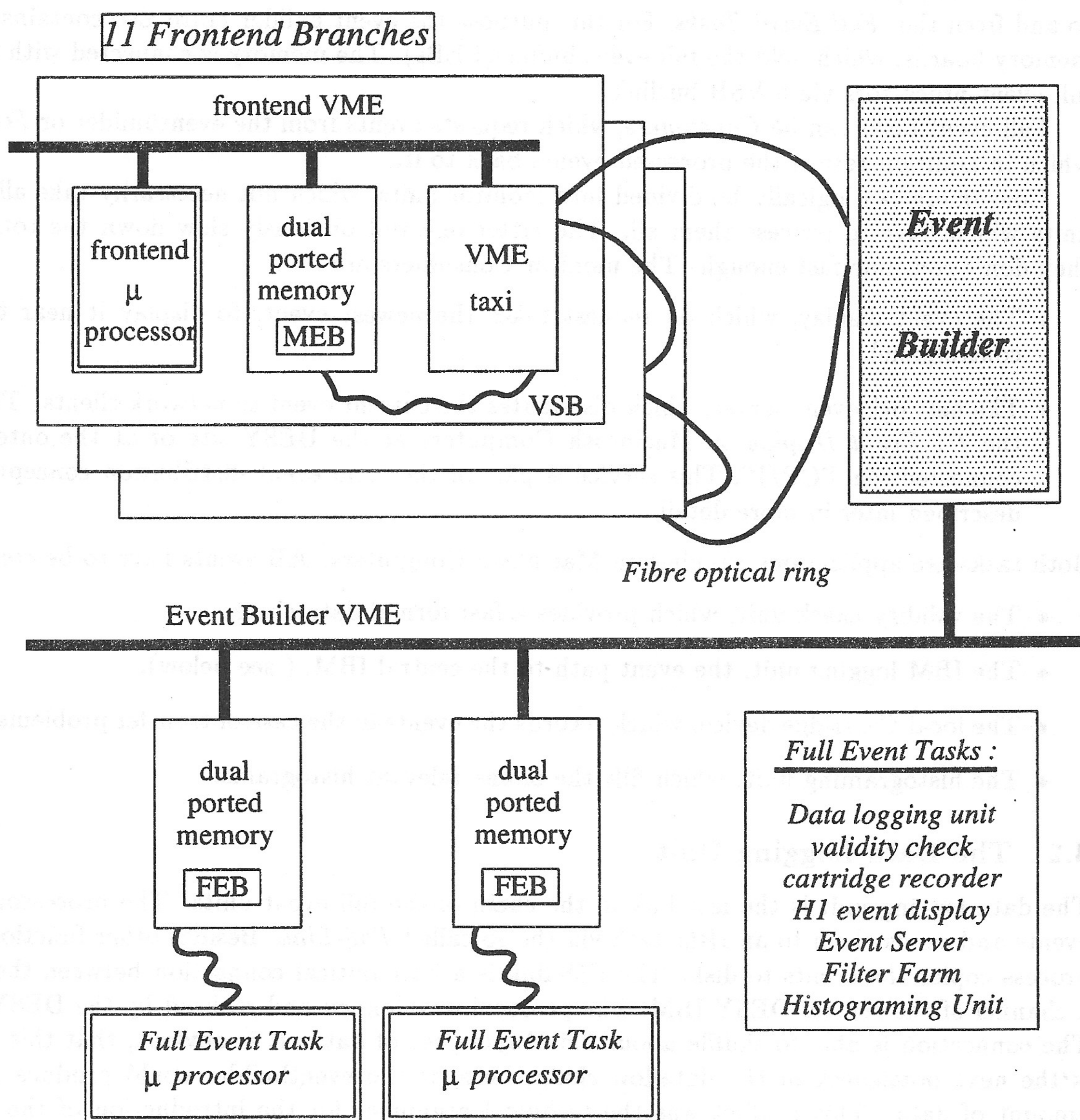


Figure 5: Event Builder Layout

μ-processor embedded in a VME[2] environment together with a 2 Mbyte dual ported memory and a so called *VMEtaxi* board, an interface to a fibre optical ring, which connects all branches with

[2]VME[VME 14] and VSB[VSB 96] are heavily used bus systems in the H1 environment

the central data aquisition VMEcrate. In addition, the VMEtaxi[3] has a VSB connection to the dual ported memory[DPM 89], which enables the simultaneous access of μ-processor and VMEtaxi to the memory board, without time delay caused by VME bus interferences. The subbranch μ-processors assemble their eventdata and copy it into the dual ported memory. To achieve an asynchronous operation, more than one event can be handled within one buffer (MEB). The eventbuilder is now able to collect these *Multi Event Buffers* via the optical ring and the additional VSB connection. With the help of the L2keep identification number, the full events can be built. The maximum transfer rate of the ring together with the recently used VMEtaxi boards is about 12.5 Mbytes/sec. Mainly caused by protocol overheads, the effective event transfer speed amounts to 6 Mbytes/sec, which leads to an event rate of 50 Hz assuming an event size of about 120 Kbytes.

The second part of the event builder task is responsible for the distribution and collection of events to and from the *Full Event Tasks.* For this purpose the event builder crate also contains dual ported memory boards, which hold the full event buffers(FEBs). The memory is connected with the different full event processors via a VSB buslink.

Full event tasks can be *Consumers*, which requests events from the eventbuilder or *Feedback units*, which in addition return the processed events back to it.

Consumers can logically be divided into monitor units, which not necessarily take all events, and units which have to process them all. The latter one will obviously slow down the total system, if they don't consume fast enough. The monitor Consumers are

- The event display, which on request takes the newest event, to display it near the main H1 controlpanel.
- The network event server, which distributes the current event to network clients. This is mainly the *H1 Event Display* on Macintosh Computers at the DESY site or at the outer Institutes, connected via TCP/IP. This service is part of the farm event distribution concept and will be described later in more detail.

Both tasks are applications running on Macintosh Computers. **All** events have to be consumed by

- The validity check unit, which provides a fast formal data check.
- The IBM logging unit, the event path to the central IBM. (see below).
- The local Cartridge device, which records the events in the case of transfer problems to the IBM.
- The histograming unit, which fills the online relevant histograms.

3.2 The Data Logging Unit

The data logging unit is the last link in the chain of the full event units. The processor requests all events and sends them to an IBM task via the so called *F58-Link.* Besides other functions, this IBM process copies the events to disk. The F58-link is a fibre optical connection between the CDAQ and a channel of the central DESY IBM. The protocol was defined and realized by the DESY F58 group. The connection is able to shuffle about 0.5 Mbytes/sec of data. It is obvious, that this transfer rate is the next bottleneck in the dataflow chain, because the eventbuilder could produce 10 times the amount of data. This conflict was the technical argument for the introduction of the level 4 filter farm.

The IBM part of the data logging system uses two different disks to achieve a maximum recording speed. While one disk is filled with the incoming data, the other is copied to the cartridge system.

A more serious argument for a filter was the fact, that simulations showed, that 95% of the data, reaching the Event Builder are beamgas and beamwall events, which means background. And there is no sense in copying them all to disk, wasting cartridge space and CPU power.

[3]The VMEtaxi is an Intelligent interface between the VME bus and an optical fibre ring

3.3 The Reconstruction

The files which are produced by the datalogging job only contains raw detector data, like hits in the track chambers or energy deposits in the LAr calorimeter. The overall aim of the analysis chain was to get the observable 4-Vectors of the detected event. The next step in this direction is to combine the subdetector specific data to more physical oriented objects. The type of object depends on the different subdetectors. For the tracking devices, tracks of maximum length are produced, and for the calorimeters, cells are merged to clusters. Then an attempt is made to combine these objects to calculate the real interactionpoint, to trace the particle through the whole detector and to obtain energy and momentum of the detected particles or jets. With all these informations some soft cuts are applied to make a first classification of the events. In H1 notation all these things are done in the *Reconstruction Program.* This program runs in several identical copies on silicon graphics machines. A master job requests the raw data from the IBM, distributes the events to the different reconstruction jobs, assembles the events after they have been processeds, and writes the result back to IBM cartridges. The different steps are shown in picture 6. It is obvious, that the current solution is not the most suitable one. So it is planned to store the raw and the reconstructed data on fast Video Tapes, direct connected to the SGI. An *Ampex DST800* robot system is currently under test. The new configuration would make it unnecessary to send the data via the IBM. One possibly (Picture 7) would be, to use the already tested *UltraNet*[4] connection from the *Farm Sparc Sun* to the SGI.

Because of the complexity of the reconstruction program, it doesn't only need additional run informations, like cracks in the detector, HV data from the tracking devices and more, but it also needs a lot of experience with the detector and the interpretation of its output. This makes it necessary to rerun the reconstruction even more than once, at least in the beginning.

3.4 The Analysis

With the production of the reconstructed eventfiles the official part of the datachain ends. From now on each physicist uses his own analysis programs to investigate the different parts of interest. For this purpose, a group of the H1 collaboration provided a nice tool[SCH 91], which tries to relieve the individual physicist the trouble of inventing the wheel. It is the first step in the direction of object oriented handling of data in the offline section, unfortunately using a language which doesn't support this feature at all.

[4] *UltraNet* is a fast Network device from the Ultra Company. It handles the TCP/IP protocol on an intelligent interface, reducing the protocol load of the CPU

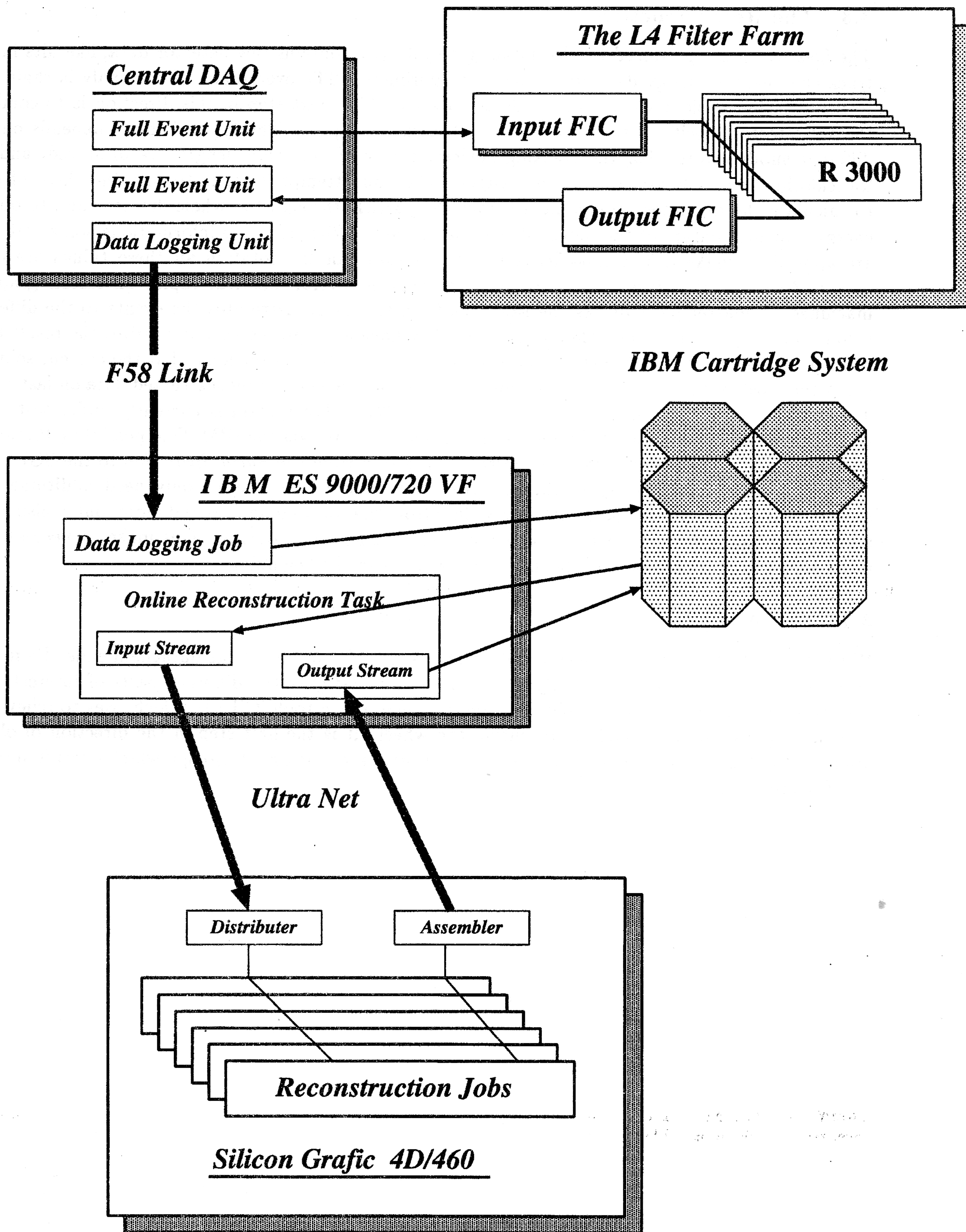


Figure 6: Current Dataflow of the Reconstruction

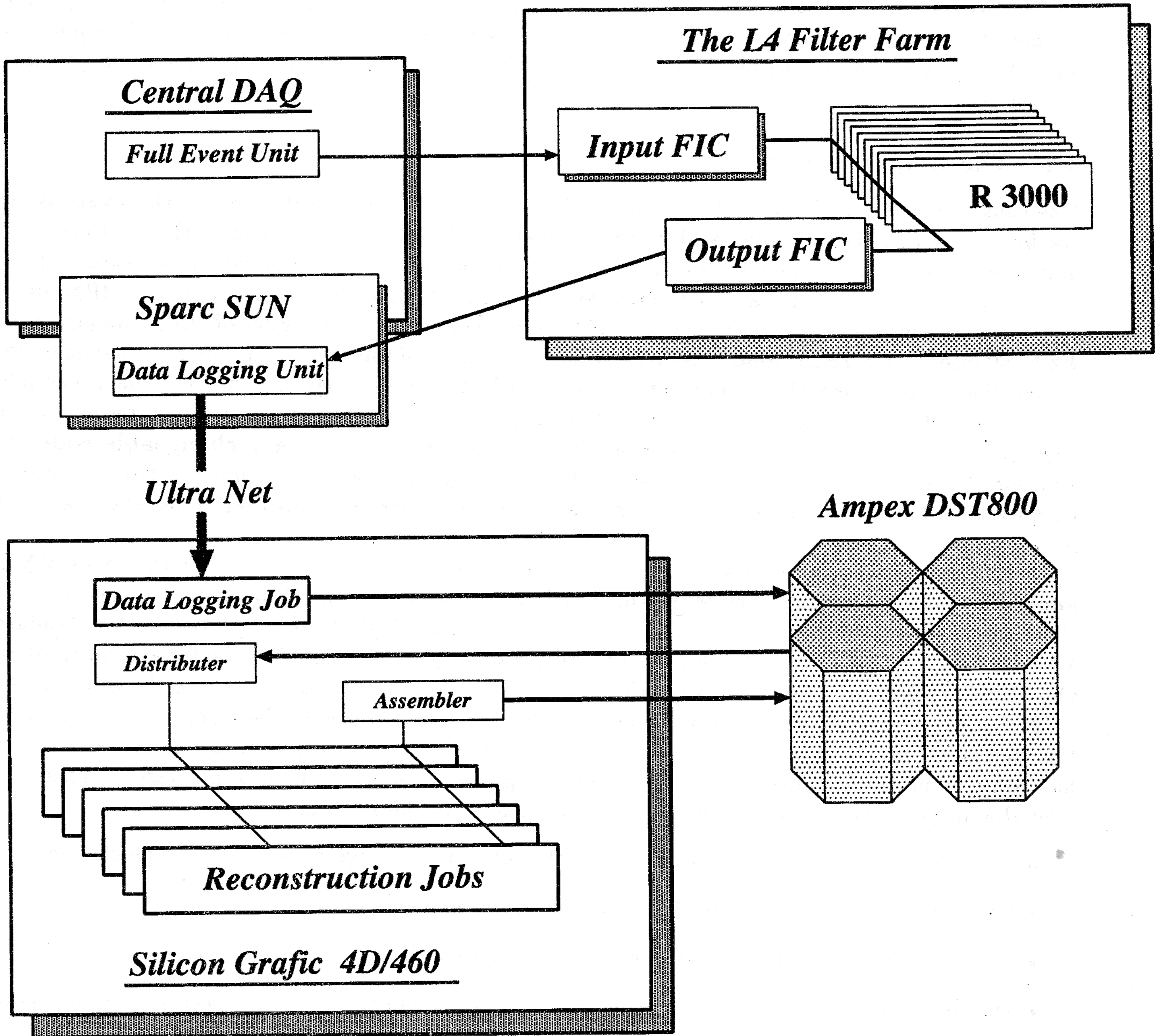


Figure 7: Possible future Dataflow of the Reconstruction

4 The Problem and the L4 Filter Farm

The previous sections briefly described the main data flow from the sensitive parts, up to the top secret private components of the analysis, which should end in the presentation of an H1 paper. During this overview, it was already mentioned, that a datarate discrepancy of a factor 10 occured between the eventbuilder and the F58-IBM-Link, where the L4 filter farm should be the way out. But this is not the only motivation of the introduction of an additional filter step. Even if we could shuffle enough data to disk, without having a glance at it, simulations showed, that this data would contain beamgas and beamwall events in the order of 95 %. This would waste luminosity and disk space in an irresponsible way. After having decided to build such a filter device, we have to find out which requirements have to be fulfilled.

4.1 The Requirements

One constraint, which of course always affects those decisions, is the **total price of the system.** So the first question is **How much MIPS do we need ?** We started with the idea, that fast decisions and rejection could be done with the knowledge about the tracks in the central and forward regions. Assuming the optimistic time approximation for a fast track finding algorithm of 150 msec (IBM 3090) in total, and remembering, that the event builder produces events in the order of 50Hz , we obtain 50 Hz * 150 msec = **7.5 IBM 3090** . Because of the uncertainty of the actual requirements it should also be possible to **increase the performance in small steps** even beyond numbers which possibly are needed after several upgrades of the frontend and trigger electronics. All these considerations presuppose, that the configuration will run flexible, that mainly means **easy changeable code.** In addition, it would be desirable to take parts of the already existing reconstruction software. So it is no question, that an Operatingsystem with a **well proven Fortran compiler** is necessary, at least for the code development step.

From the CDAQ side of view, the system must fit into the CDAQ concept, which means **VME access** is a must, and a **support from the hardware designer** wouldn't be bad. It is clear that all these arguments point to a multiprocessor environment with single board μ-processors for standard VME crates. The decision for the R3000 μ-processor was taken because of the **fast development in the RISC technology**, and the hope, that the R3000 successors can easily replace the recent ones.

In multiprocessor systems saturation effects often come into account. They are mainly caused by the increasing coordination overhead and access conflicts on the common data busses. So a setup had to be found, which shows an almost **linear response** with the number of processors. There are several additional requirements concerning the utilization of the L4 trigger.

- So it shouldn't be necessary to recompile the software, if triggercriteria have to be changed.
- These changes have to be possible in the short time inbetween runs,
- and non experts should also be able to learn this procedure in an acceptable time.
- During the runs, all relevant quantities have to be available, to change selection parameters in the case of strange behaviours.
- A full simulation of the filter program, with real and simulated data, should be possible during a real run is active.
- And the filter decisions must be offline reproducible

4.2 The Solution

The realization of the Filter Farm turned out to become a rather complex exercise, because a lot of informations had to be make available to the farm processors from outside and vice versa. In addition,

a set of features had to be provided, which not directly touch the main datastream, but were essential for the steering and monitoring of the system. In this section I would like to briefly summarize the components of the L4 filter farm, concerning soft and hardware. The following chapter then will go into detail.

4.2.1 The Hardware

The kernel part of the solution is the decision for the RISC R3000 and successors as farm processor. It provides about half the CPU power of one IBM 3090 element, which means, that in the beginning of the experiment 15 of them should be sufficient to prove the functionality of the farm concept. Using this chip, the company CES[5] offered a single VME board μ-computer, the RAID 8235 (see Figure 8 on page 21), which exactly fulfills the specifications we needed. The most important points are

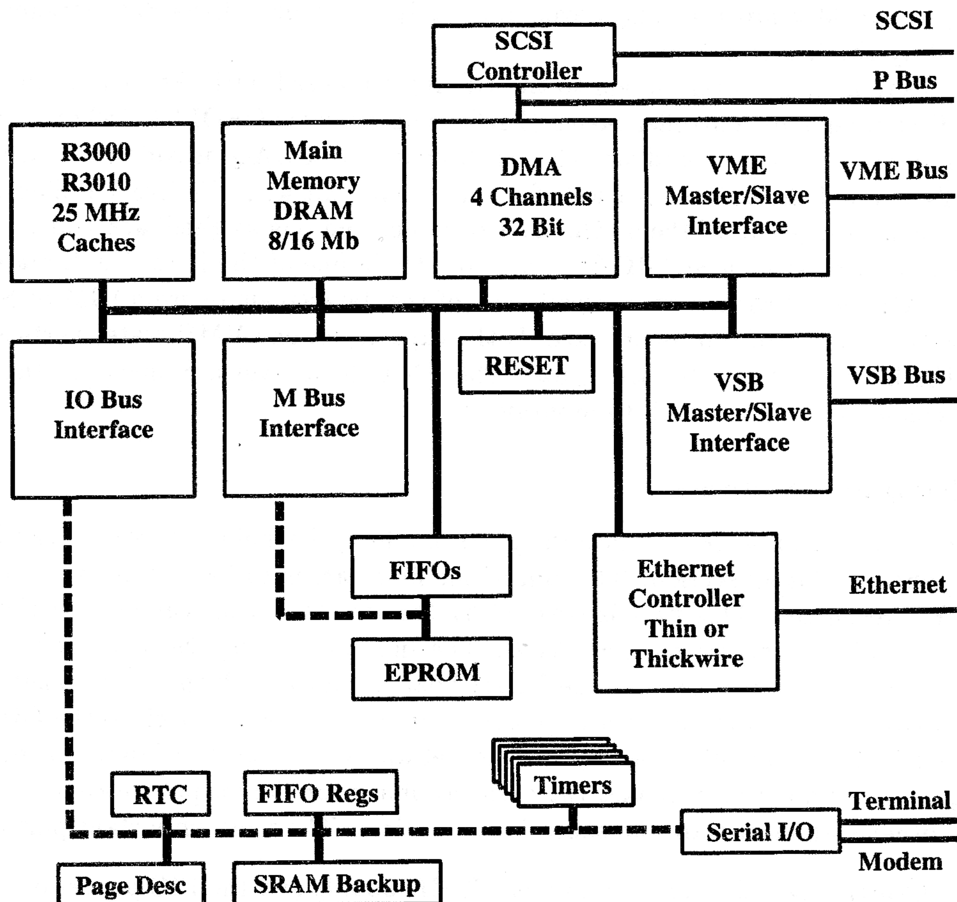


Figure 8: The RAID 8235 Block Diagram

- R3000/R3010 processor with 25 MHz
- up to 128 Kbyte instruction and data cache
- special write buffer, to speed up the write access
- up to 16 Mbyte Main Memory
- VME and VSB Master/Slave Interface

[5] Creative Electronic Systems, Geneva

- large EPROM for the monitor program
- and of course Serial I/O ports

For the use with an Operatingsystem a *SCSI Controller* and an *Ethernet Controller* are also on board. The decision was to run these boards without Operating System. This became possible because an other fully equiped $\mu-$computer was available with the R2000 processor, an predecessor of the R3000. This Computer, a MIPS/120, is running an Unix Operating System with an optimizing Fortran Compiler, and a standalone library with enables the execution of programs compiled and linked under M/120 Unix to be run on the CES RAID 8235 processor boards. The executable code is loaded into the RAID boards with a Macintosh Computer, which on the one hand can read the disk of the M/120 and on the other hand is able to access the main memory of the different RAID boards. This Mac also monitores the operation of all involved tasks and in addition distributes events to other computers on request. A second Macintosh displays events, directly taken out of the data stream and histograms accumulated by the RAID boards.

The connection of the total system to the CDAQ is realized with the help of an other microcomputer, the Fast Interface Controller board (FIC) using an MOTOROLLA 68020 processor chip (see Figure 9 on page 23). This board was chosen, because the CDAQ provides a library for that model which totally covered the communication of a Full Event Unit with the Event Builder. To provide the full speed, we are using two of them. One as Consumer and one as Feedback Unit.

For simulation reasons, a SPARC SUN also resides in one of the Farm VME crates. It can feed a simulation RAID board with events directly out of the datastream or from IBM or SGI disks.

Because the amount of processors exceeds the capacity of one VME crate, Vertical Interface Controllers, VICs between the crates are in use. The VICs themselves are linked by the VMV cablebus, which also connects the Macintosh Computers.

4.2.2 The System Software

The system software can be divided into the parts handling the main data stream, and the controlling, monitoring and simulating section. The aim of the first part is to be as fast as possible, to avoid the waste of CPU resources. This part is covered by the two I/O tasks running on the FIC processors and their protocol counterpart on the RAIDs. The Input FIC program requests the events from the eventbuilder and distributes them to the RAID boards, and the Output FIC takes the event out of the RAID board again, if the filter decision was positive, and feeds it back to the CDAQ. The system software on the RAIDs handles the protocol to get an event from the Input process, calls the Real Filter Program, and afterwards makes the event available for the output process.

For the other systemsoftware the speed is not critical. The most important parts are :

The Farm Control which loads the different RAID boards and the FIC processors and starts them. In addition, it permanently displays important quantities like the total number of processed events and the rejection ratio.

The Database Connection which sends the Farm steering file to the central H1 database on the IBM and obtains new runinformations from there.

The H1 Mac Event Display which displays the current event out of the datastream or which shows farm specific histograms with data directly taken from the farm processors.

The Event Server which sends all these informations to H1 Mac Event Displays, not directly connected to the farm VME crate.

The SUN SPARC simulation program which simulates the entire world for one or more RAID boards, running filter test versions, before they are used in the real world.

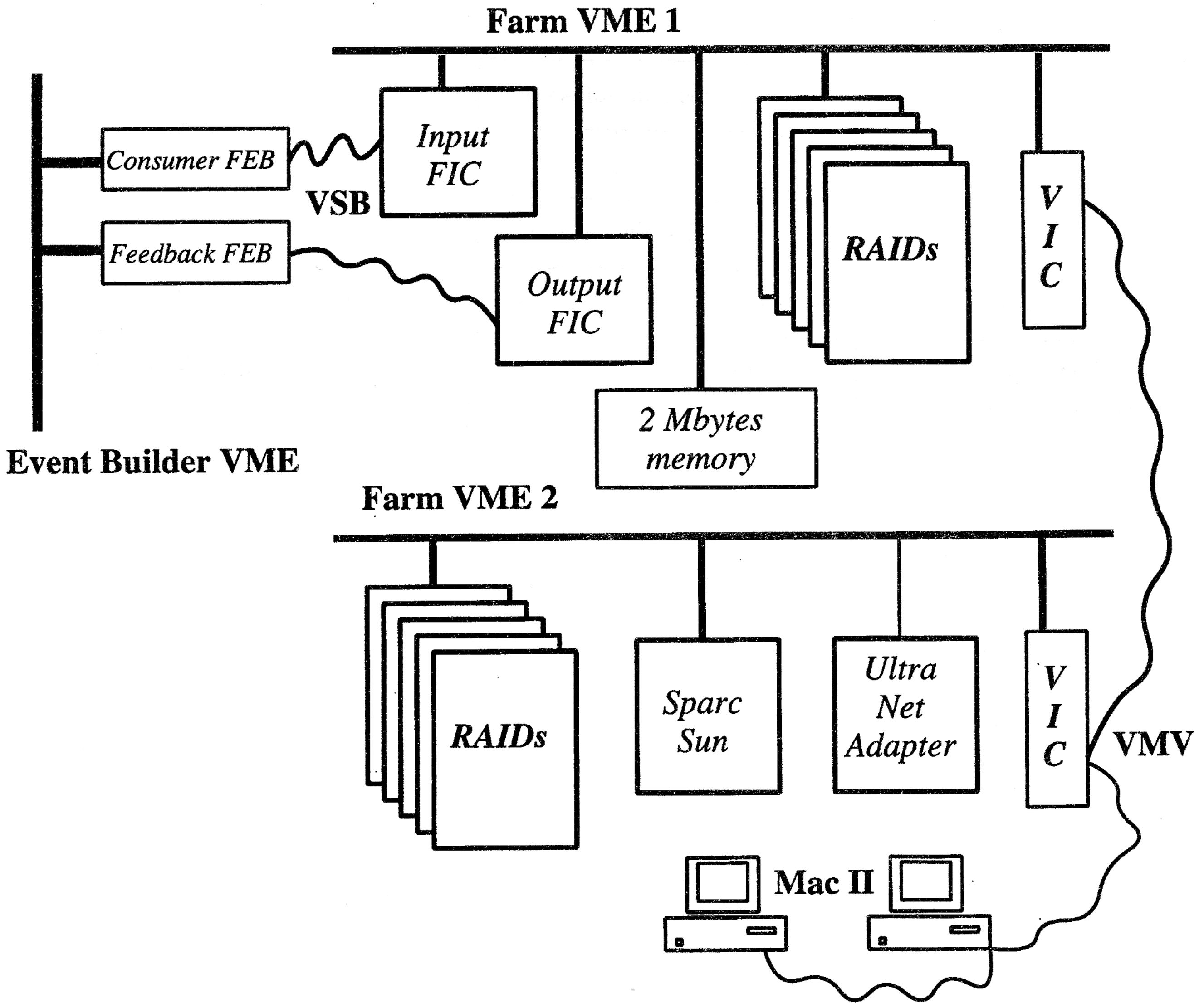
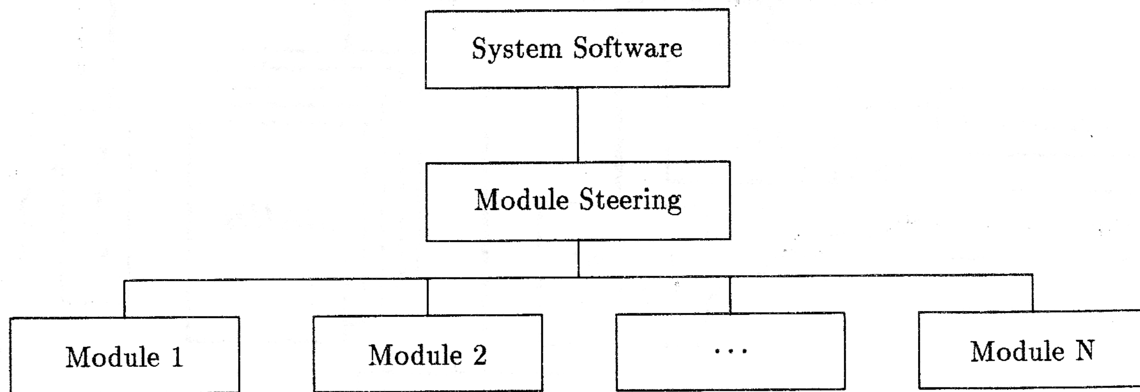


Figure 9: Physical Farm layout

4.2.3 The Filter Software

The Filter Program which is called by the RAID system software for each event, again consists of two major parts. The steering module, and modules calculating decision relevant quantities. The steering part determines the sequence in which the modules are called, depending on the output of the previously called ones and the remaindent time for the current event. The decisions are based on informations provided by the farm steering file.


System Software
Module Steering
Module 1
Module 2
...
Module N

5 The Details

5.1 The Uniform VME Addressing

5.1.1 The Hardware Connections

One of the most important aspects of a device, which is sayed to be a processor farm, is the data exchange of the individual farm elements. In our case this is mainly done via the VME bus, realized as backplain in the different farm crates. Each crate can hold up to 15 single slot boards, and contains at least one Vertical Interface Controller which are mutually connected with a VMV cablebus. In addition this bus also links the two farm Mac II with the rest of the system, because they are standalone Computers without a direct VME interface. Instead, they contain an interfacecard to the VMV bus.

5.1.2 The VME Addressing Concept

In the VME concept, each element can have active or passive interfaces to the bus, or both. All our processors have active access, that means they can read and write from/to the full 32Bit VME addressspace. Some of them also have a passive interface which enables others to access parts or all of their main memory. In the setup phase, each passive component has to learn which address the first byte of its own memory will have, seen from the VME point of view. So all passive participants occupy a different part of the total address space(see Figure 10 on page 26). The Macintosh Computers behave slightly different. Though they have active access to the full address space, they don't share parts of their main memory. Instead, they can use the internal memory of the VICs, which for its part is commonly accessible.

If a processor needs to access a part of the VME address space, it can't do this directly in specifying the appropriate address, but it has to use an internal mapping mechanism which is different for the various computer types. One of the memory boards for example is set up to have the global VMEaddress of 600000Hex, but none of the processors using this card can access the memory by specifying that number. The Macintosh points to it with F0600000Hex and the FIC Board with F0E00000Hex. The *Sparc Sun* takes the address out of its virtual memory pool, which means that the pointer will be known not before the application starts.

Most common is the usage of a programmable page descriptor, which has to be set up prior to the access. This page descriptor is a table which maps not locally existing memory space to an arbitrary VME space. Operating systems with Virtual Memory make use of an additional mapping between the virtual address and the physical one.

$$\text{virtual Address} \xrightarrow{\text{Map1}} \text{physical Address} \xrightarrow{\text{Map2}} \text{VME Address}$$

All this is hidden to the application programming level. One only has to request a local pointer for a part of the VME address space, which one has to specify. From that moment on, the returned address can be used to read and write from/to this memory portion via the VME bus. These different behaviors obviously make it difficult to use the same source code on the various machines, which are

- The Farm RAID boards (no OS)
- A RAID board running Unix (TC/IX)
- The I/O FIC boards (no OS)
- The SPARC SUN running SUN OS
- The two Macintosh Computers

To come around this problem, I have written a library, available for all of the above listed machines, which considers their specific handling of the active and passive VME bus access. The basic functions are :

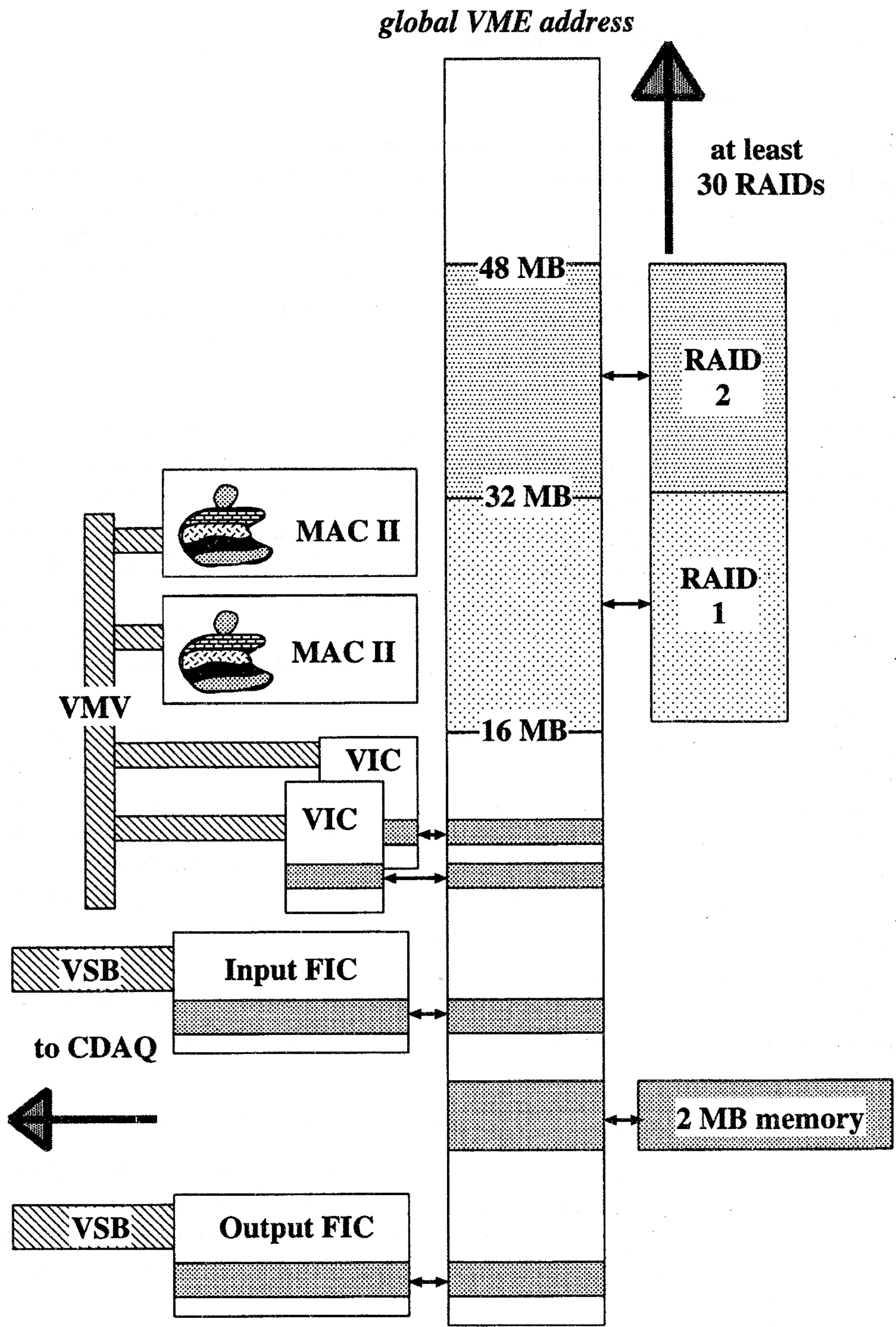


Figure 10: Logical Farm VME Configuration

- Return of the local address under which the specified VME address is accessible.
- Return of the VME address with which an other process has access to the local memory, if the local computer support passive VME access to this portion of memory.
- Transfer of data between VME and local memory, using the global VME address as source or destination.

The latter function can also be used on machines, not directly connected to our farm VME system. They only need to be tied up to the TCP network [6]. The advantage is, that source code is portable between the different computers, without changes and more flexible data exchange is possible. Imagine, that a task wants to make a local portion of memory available to others. Where this data resides is normally not know during the design phase of the system, which means, that a hard wiring of addresses is not possible. With the global mapping functions, the service task only has to call the LOCALtoVME mapping to get the appropriate VME address, and to send it to a commonly accessible mailbox. All other processes obtain the local pointer of the remote data with the VMEtoLOCAL function. This scheme is heavily used in our system.

5.2 A Relay Race

To avoid a waste of time with protocol overheads, the main eventflow is designed as simple as possible. An event is requested from the eventbuilder by the input FIC, which copies it to an idle RAID board, which starts the reconstruction and decision process. In the case of a negative decision, the event is discarded. Otherwise the output FIC is requested to feed the event back to the CDAQ. We should focus the attention to three points, which mainly determines the performance of the eventhandling :

- From the I/O FICs point of view, the eventsource and destination are connected via different bussystems. In detail, the Input FIC takes the event from the corresponding FEB via the VSB cable and copies it into the RAID memory over the VME backplain. The Output FIC acts the other way round. Both FICs use their DMA controller to do the transfer, that means, that the data is not stored intermediately in the FICs local memory. Under these conditions, the system makes optimal use of the different busses, because after some cycles, the two processes seem to be synchronized. During the time, the Input FIC, for example, reads a dataportion via the VSB from the Consumer FEB, the Output FIC can use the VME bus to access the RAID board. As a result, two different copytasks can act with nearly the full transferspeed.
- Assuming an eventsize of about 120Kbytes and a maximum bus transferspeed of 6 Mbytes/sec a total copycycle (Input and Output) lasts in the order of 80 ms. This time can't be neglected against the total processetime of the event. To profit from this time, two buffers are used alternatively inside the RAID boards. While the decisiontask cares about the event in one of the buffers, the other one can be emptied by the Output FIC and filled again by the Input FIC.
- To keep track of the different states in this multiprocessor scenario, we decided, not to use interrupts for the interprocessor communication, but instead, to use mailboxes. Because of the heavily used VME bus, it is essential, that polling those mailboxes has to be done only in local memory, and never over the VME bus.

Keeping these points in mind, the following scenario was implemented. (see diagram below and Figure 11 on page 28).

Each of the I/O FICs holds a table, the **RAID Buffer Table, RBT**, in its remote accessible memory, which provides one slot for each RAID event buffer, that means two per RAID. Immediately after the FIC task starts, the address of this table is send to a mailbox. The RAID programs can

[6] The way this is possible, will be discussed later.

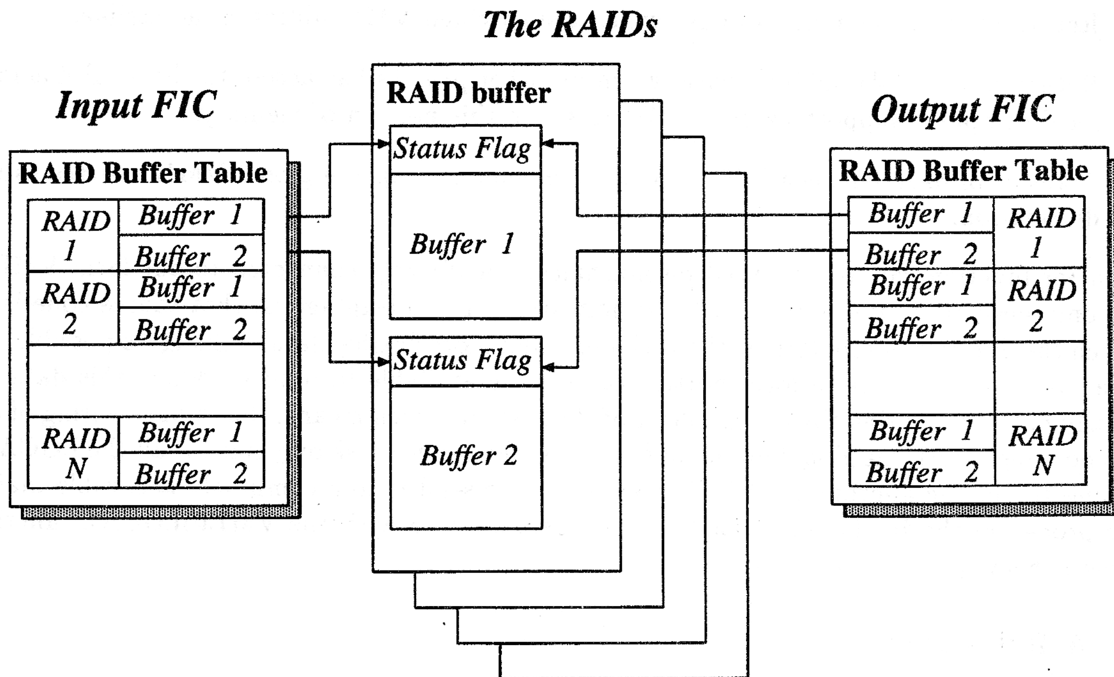


Figure 11: Fast Dataflow Protocol

read this information, and with the help of its own identification number, the RAID calculates the entrypoint of the two eventbuffers inside this table. This is done by both FICs. After that, the RAID boards determine the VME address of the two local eventbuffers, and write these numbers into the previously calculated slots inside the *RBT*.

The Input FIC permanently scans this table, which is no problem, because its only local memory. If it finds an entry, it maps the found VME address to a local one, requests an event from the eventbuilder, and initializes the DMA transfer between the FEB and the RAID memory. After the transfer is finished, a flag inside the RAID board is set to a predefined value. Inbetween, the RAID observes the two flags of the event buffers, which again doesn't charge the bus, because its done locally. With the change of the flag value of one of the buffers, the RAID knows that the event fully arrived, and that it can be processed. The actual Filter program is started on this event, and according to the decision, the flag is set to *accept* or *reject*. In addition the bufferpointer of the relevant event, mapped into VME notation, is written into the buffertable of the output FIC. This FIC behaves similar to the Input FIC. Depending on the flagvalue, it resets the RAID eventbuffer or copies the event back to the event builder. In any case, it takes the bufferaddress out of its own buffertable and writes it into the corresponding slot inside the Input FIC, because it can be sure, that the buffer is ready again. From now on the VME address is handled like a token. It wraps around together with the event requests.

Step	example Slot in RBT of Input Fic		status flag inside RAID		example Slot in RBT of Output FIC
Step 1	0		IDLE		0

RAID writes address of buffer into RBT of Input FIC

Step 2	VMEaddress	⟹	IDLE		0

Input FIC scans RBT and finds VMEaddress of a RAID buffer. Input FIC requests next event from Event Builder and copies if from FEB directly into RAID buffer with the specified address. Then it sets the status flag to EVENT_COPIED, and clears the slot in the RBT.

Step 3	0		EVENT_COPIED		0

RAID program was waiting for EVENT_COPIED message and processes the event. After that, the status flag is set to ACCEPT or REJECT, depending on the decision of the *Filter Program.* In addition the VMEaddress is written into the corresponding slot inside the Output FIC.

Step 4	0		ACCEPT	⟸	VMEaddress

Like the Input FIC, the Output FIC scans the RBT, finds the entry, and depending on the status value, deletes the event inside the RAID board, or it requests a free feedback buffer from the Event Builder and copies it. After that, the Output FIC sets the status flag to IDLE again, and copies the VMEaddress of its own RBT into the corresponding slot of the Input FIC. Then we continue with step 2.

5.3 The RAID Program

The behavior of the RAID program concerning the *Run States* can be expressed in form of a state transition diagram(see Picture 12). The RAID program only exists in one of the three states, **Wait-State**, **RunActive** and **RunGoingDown**, represented as bubbles in the diagram. State transitions are stimulated by events like **prepareRunEvent**, **bufferReadyEvent** a.s.o.. During the transition, actions have to be performed, like a refresh of the constants or the execution of the main *Filter Program* described in section 7. Actions are marked as boxes.

After the start of the RAID program, it first sends a request for new constants to one of the FIC processors, and awaits their arrival. If the run is already active at that point, it sends an *Input Request* for each of its buffers to the Input FIC and goes into the *RunActive* State. Otherwise, if the run is still stopped, it directly jumps into the *Wait* State. In this state it only waits for the *prepareRunEvent*, which is distributed by the Input FIC, to all RAIDs on *Run Start.* The RAID board than compares its own version of the loaded constants with the current one residing on the memory card and requests a new set if necessary. After that it sends the *Input Requests* to the Input Processor and goes into the *RunActive* State. In this state, the *BufferReadyEvent* is of course the dominating one. This event also contains the information, which buffer contains a valid input event and has to be processed now. The actual *Filter Program* is then called with the appropriate buffer number. With a *prepareRunEvent* during the *RunActive* State, the RAID board can be asked to copy new constants from the memory card to its own memory. A *nothingEvent* indicates, that there is no new event in one of the I/O buffers. The behaviour in such a case is the difference between the *RunActive* and the *RunGoingDown* state.

Because the *RunGoingDown* state can only be reached after the run stopped, a *nothingEvent* not only indicates that there is no new event, but that there will be no new one until the run starts again. This is the condition to enter the *Wait* state. During the transition to the *Wait* state all RAIDs except one, only clear their *Input Requests* inside the *Input Processor*. One RAID, which can be any of them, got the so called *End Of Run Record*[7]. This RAID waits until all other one reached the *Wait* state and until the *Output Processor* cleared one of its I/O buffers. Then it assembles the histograms of the other RAIDs and adds them to one set of histograms. Finally it send them to the output stream encapsulated as an ordinary event, but with a special event number. Now, also this RAID switches to the *Wait* state, and the cycle can start again.

With this scenario, it is possible to activate a new RAID processor at any time, even if a run is active.

Although each RAID is able to copy the constants into the own memory itself, it will request them in most of the cases. The reason is to avoid an overload of the VME bus caused by 30 RAID boards trying to read from the same memory card. On request, the *Input Processor* uses the DMA[8] controller to copy the constants sequentially into the requesting boards.

All events except the *bufferReadyEvent* are generated by the I/O processors writing the appropriate event number into a mailbox of the RAID board. The *bufferReadyEvent* is produced internally, if the RAID finds out, that a new event has been totally copied into an I/O buffer.

[7]The EndOfRunRecord is a pseudo event with a special event number. It contains run specific informations available only at the end of a run

[8]Direct Memory Access

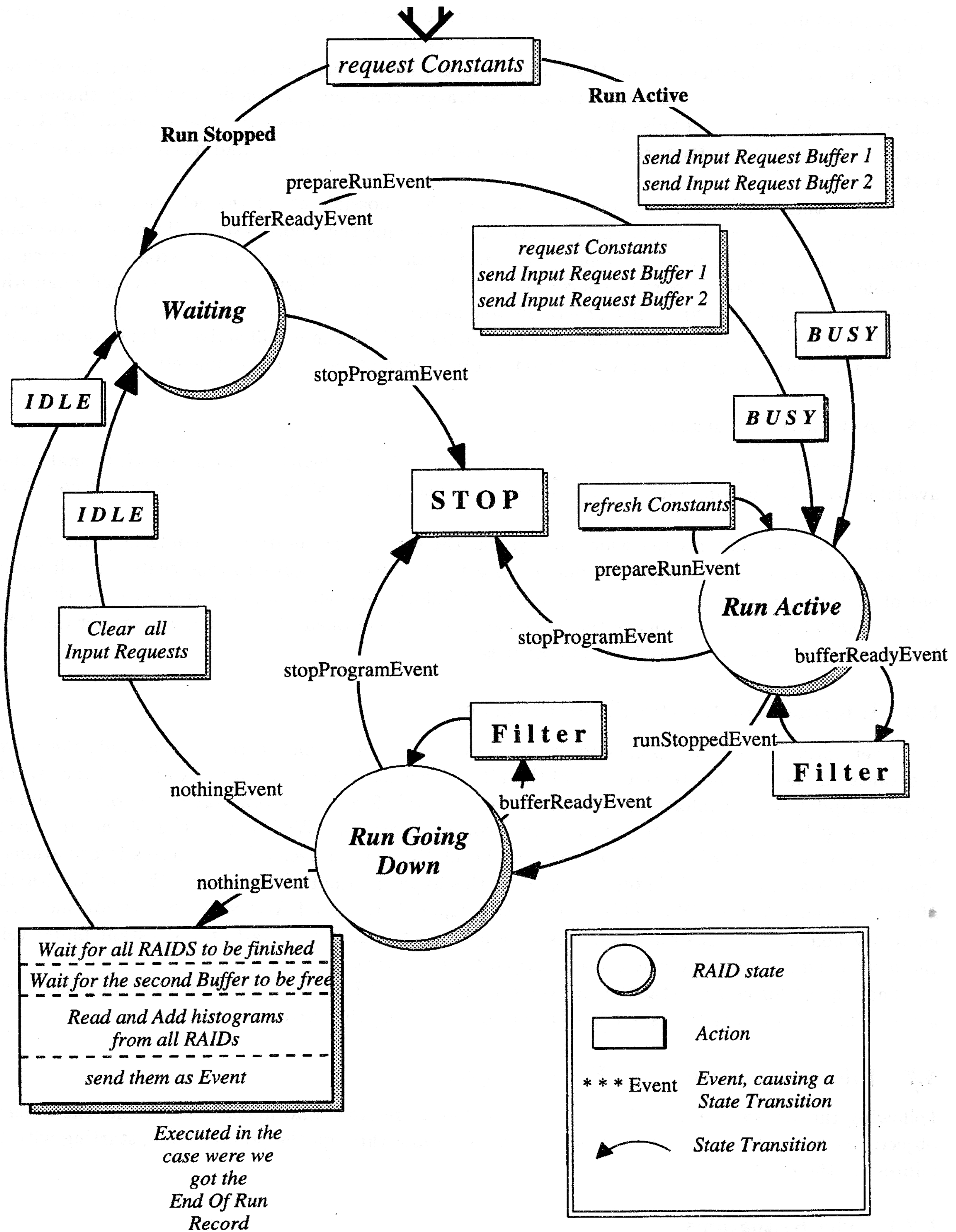


Figure 12: State Transition Diagram for the RAID boards

5.4 Information Exchange between inner System and Rest of VME

The previous discussed data exchanges only concerned the eventflow, but a lot of more informations had to be made available to and from the Filter Processes.

The individual Modules inside the Filter Program need special informations about the different detector components, like geometry data and calibration constants. These data will only change from run to run. The source of this information can be the **H1 database on the central IBM**, or a special **Begin Of Run Record** which is produced by the eventbuilder and distributed prior to the first real event.

On the other hand it's important to know as much as possible about the behaviour of the system during a run. These informations can be statistics concerning the whole system, like active processors, processing times, rejection ratios and so on. But even more important are histograms, which are assembled by the Filter Program, concerning the distribution of basic and reconstructed quantities. Because the Farm is the first fully programmable device in the data chain handling the full event, it gives the chance to monitor critical detector parameters while a run is still active. This is an additional help for the central DAC shift crew to detect malfunctions of detector components.

5.5 A Pointers Paradise

All these informations reside at different addresses in different devices. Our aim was, to make them available by only knowing one magic number, the base address of the *Communication Vector Table, CVT.*

The CVT is a pointer array which always has the same VME address. It contains pointers to all relevant structures, describing the different objects of interest. Figure 13 represents a small section out of the pointer construct, which is sufficient, to follow the discussion. The references to the *RAID Buffer Tables* inside the memory of the FICs was already mentioned. They are necessary for the RAID boards, to place their Input and Output Requests.

5.6 Informations for the Filter Program

An other entry in the CVT points to an array on one of the memorycards, which holds the newest constants for the Module steering part of the Filter Program and for the various Filter Modules. This buffer is updated with informations from the database before a new run starts, but only if some of them changed. But this is still not the final constant version. With the arrival of the first event, which is always the *Begin of Run Record*, the Input FIC has to decide if it contains informations to update the constants or to be added to them. If this happens, the FIC increments the Version Number of the constants in the CVT. This number is compared by each RAID with the version number it currently holds. If it finds out, that if is no longer up to date, it requests the constant buffer using the mechanism, it normally would use to request a new event.

The request for new constants is answered by the Input FIC, not only at run start, but also during a run. This is necessary to restart RAIDs without stopping the current run.

5.7 Informations from the Filter Program

Following the arrows in Figure 13, it it obvious that all relevant informations about the the different objects inside the Farm can by obtained by only climbing through the pointer trees, starting with the address of the CVT.

5.7.1 The Histograms

Whenever a RAID program starts, it converts the entrypoints of its histogram buffers into VME addresses, and assembles them in the *RAID Histogram List*, preceded by the number of entries following.

The address of this list is written into an entry inside the *RAID Descriptor Block* . All these memory portions reside inside the main RAID memory, which is fully accessible via the VME bus. To be linkable from outside, the converted address of the RAID descriptor block is written into a RAID specific slots of two tables inside the memory card, the *static RAID table* and the *dynamic RAID table.* The CVT points to both of them. So, at least any VME linked task is able to look into the histogram buffers of all RAID boards. The way this is done in practice, and from whom, is show in section *The Passive Farm OS.*

5.7.2 The Monitorinformations

Informations concerning the more technical aspects of the farm, can be obtained in an active and a passive way.

The latter one are the already described jumps through the wood of pointers, producing informations like the number and length of the currently processed event per RAID, the number of already seen events and so on.

For the active interaction with a RAID task, each RAID descriptor Block contains an **External Command** field. If a none zero number is written into this word, the corresponding RAID process interpretes the number as a command and tries to execute it. Most of the supported requests serve as debugging tools. The only frequently used command is the *Alive Request.* On this event, the RAID task clears the External Command field and continues. This scheme is permanently used by the Output FIC to determine if the RAID processes are still alive. If the request is not answered within a fixed time interval, the corresponding RAID is declared to be dead. Its entry is taken out of the *Dynamic RAID Table*, a warning message is written to the main Filter Farm Console and the information is forwarded to the CDAQ display. An automatic restart of the died task is possible and foreseen, but not yet implemented. The reason is, that it turned out, that these events mainly occured because of bugs in the imported reconstruction software. And it seemed to us to be a good habit, at least in the beginning, to find these problems and not to ignore them. At this point, the sense of two *RAID Tables* becomes obvious. With the cleared entry in the dynamic one, we are informed about the RAIDs making problems, and with the static one, we don't loose track of the internal RAID tables.

5.7.3 The Events

To obtain events out of the data stream similar to the histogram scheme is problematic.

- The events don't reside long enough in the different buffers to copy them during the normal dataexchange.
- Changing the FIC-RAID protocol to lock a desired event, would slow down the throughput, and would be potentially unsafe, because an aborted request task could block one buffer forever.
- In addition, allowing an unknown number of different tasks to lock and unlock memory portions makes it necessary to introduce a semaphore scheme.
- For some investigations, it is necessary to specify properties of the requested event. One or more external tasks, permanently comparing the desired selections with the actual properties would surely overload. the VME bus.

The protocol, which I decided to use, is similar to the communication between the IBM cpu and its channelprocessors(see Figure 14).

Each client, which intends to request an event has to provide two buffers. The first is the container for the expected event, and the second is a parameterblock describing its properties, the *Request Area.* Both parts have of course to reside in VME accessible memory. The client has to enter valid values in the following field of the Request Area.

Command	in our case 'GIME_EVENT'
BufferAddress	VME address of the container buffer
BufferSize	Size of the container buffer
RAID number	Number of the RAID board, the event should come from, or ANY.
decision mode	The event should have been accepted or rejected
decision flag	32 bit mask, compared with a value produced by the different filter modules

The VME address of this Request Area has to be written into the mailbox of the processor, which is responsible for the appropriate request. Events are delivered by the Output FIC. As expected it registers the address of its own mailbox into a field of the CVT, on Farm Startup. After receipt of the Request Area Pointer, the Output FIC clears the mailbox, copies the requestarea into a local queue, and changes the answer field in the request area to 'REQUEST_ACCEPTED'. The Output FIC permanently compares the properties of the processed events with the requested properties of the whole queue. If they match for a queue entry, the event is copied into the container area of this request. The request area is completed with the actual values, and the answer field inside the request area is set to 'REQUEST_FINISHED' or to 'REQUEST_REJECTED', if the FIC queue is full or the specified parameters don't make sense. The client task has to observe this field, until one of these values appears. Because of the polling, it is recommended, that the Request Area should be local for the requesting task, or in the case of the Macintoshes, resides in one of the VIC memories, which can be accessed by the Mac via the VMV bus, without interferences with the VME bus system.

Perhaps an other point should be stressed. The FIC mailbox can be accessed by several computers simultaneously, which yields to the result, that one of the addresses will be overwritten by another, if the FIC didn't look into the mailbox inbetween. To avoid this conflict, each participant first reads the mailbox to be sure, that it is empty. But even inbetween the read and the write, another task could have used the mailbox which wouldn't be detected. The most elegant way out, is the use of a special VME command, the 'read modify write'. During this cycle no other access is permitted, which ensures the conflictfree handling of the mailbox scheme. Up to now, we didn't use this special command, because in practice the problem never occured, and the worst result would be, that a request isn't answered. This malfunction would be detected by the clienttask, because it would observe, that the answerfield doesn't change to 'REQUEST_ACCEPTED' during a fixed timeout period. The task then is free to send the request again.

5.8 The Environment Simulation

The system software described in the previous sections, has been tested and used during the last half of 1992. The time, the system can act without a total breakdown, is in the order of days. So there is no need to concentrate on the debugging of that part. This is not true for the reconstruction modules of the RAID program. This software grows and changes permanently, which makes it necessary to have a tool with which the new versions can be tested in a nearly identical environment. The simulation should be possible with simulated and real events from the IBM and SGI disks and of course directly out the main data stream.

For this purpose a third farm VME crate holds a Sparc SUN, a single RAIDboard, some memory and as usual a VIC to link the crate to the rest of the farm environment. Because of the fact, that this crate doesn't contain any real run relevant objects, the bus can be used without disturbing the main system. But on the other hand, the address space is totally integrated in the Main Farm address space. That enables us to use the previously discussed protocol to get events from a real run, with which a new Filter Versions can be tested. For the simulation the software on the RAIDboard is

absolutely identical to that one used in the real system. All tables, including the CVT and the tables normally residing in the I/O FICs are now on a special memory card. The actions which are done by different processors in the real world, are totally simulated by tasks running on the Sparc Sun. The actions are

- Loading the object code from the local SUN disk or via NFS into the RAID memory.
- Loading the constants from Disk into the memory card.
- Consuming events from various sources and feeding them to the RAID process. (Input FIC)
- Reading the processed events out off the RAID board and writing them to Disk. (Output FIC)
- Extracting the histograms out of the RAID card and making them available for a Display Program.

This simulation system is an event consumer and an event and histogram source. Because of the identical pointer structure, applications which are able to get information out of the real system also will get the data out of the simulation, by only changing the CVT base address. Actually, the simulation is not restricted to the use with one raid board. In addition the Sun can also simulate the Event Builder task only, and serve the real I/O FICs with the total system.

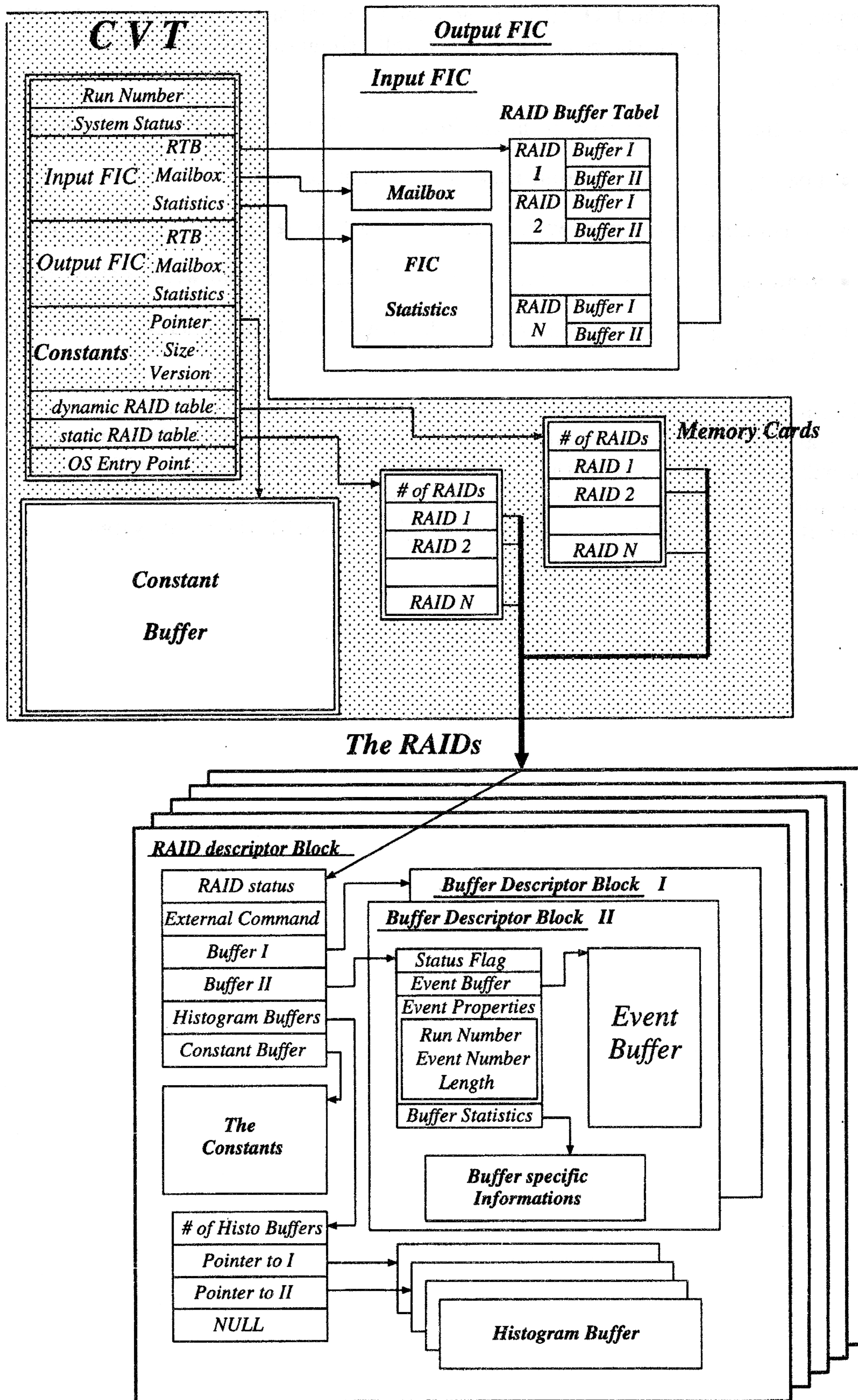


Figure 13: Pointer Layout

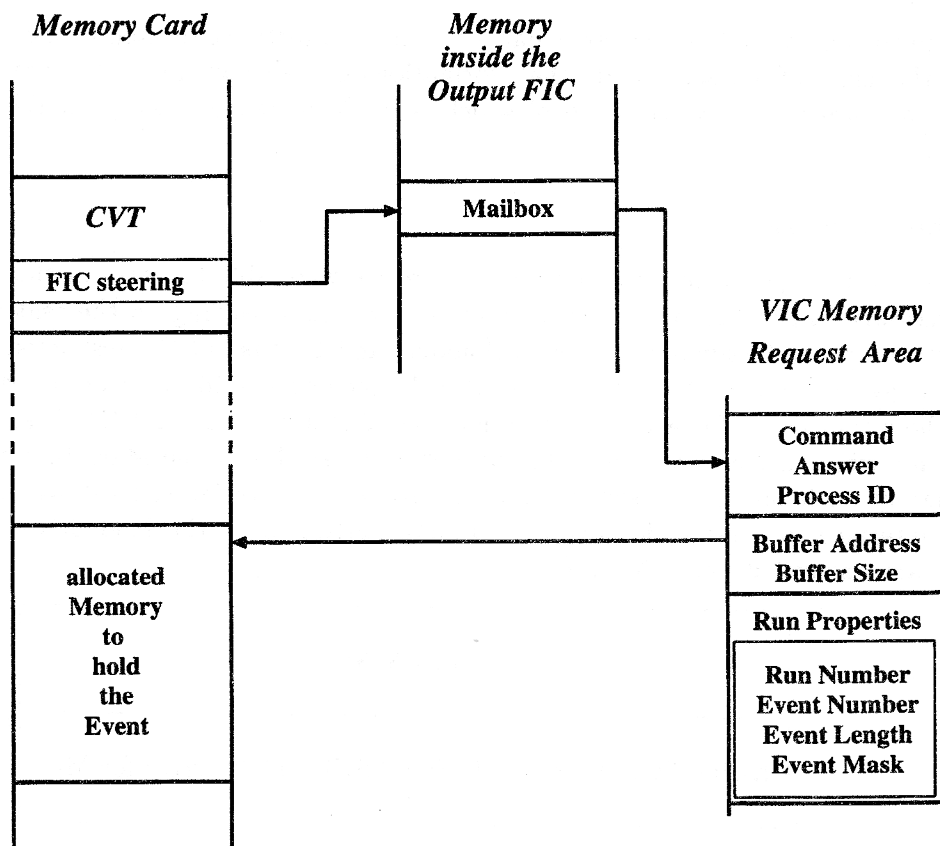


Figure 14: Farm Event Request Protocol

6 A general Data Distribution Concept

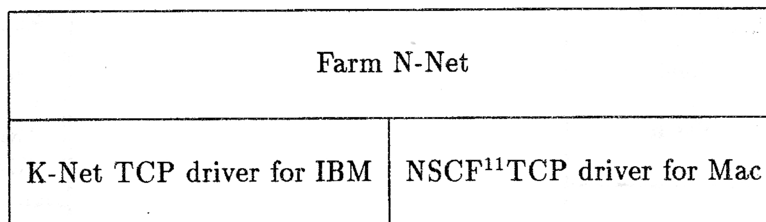
Up to now, we discussed how to obtain different types of data objects out of several steps in the data processing chain, using different protocols :

- Events from the CDAQ with CDAQ-FEB protocol.
- Histograms from the CDAQ Histograming unit, using direct VME memory access. Mainly used by H1 Online Event display.
- Events from the Farm I/O processors via the *Farm Request Protocol.*
- Histograms from the Farm, directly out of the RAID histogram buffers, or from the simulation processes.

All these protocols can only serve participants which are directly connected to the corresponding VME bus. But the aim is of course, to make the informations available wherever they are needed. To provide this, we make use of a network protocol which is implemented on nearly all DESY computers, the TCP/IP[9].

6.1 The Network

When the Farm project came into a phase, where we were able to perform the first realistic tests, it was necessary to run the system with simulated eventfiles residing on the central IBM. We used a Macintosh computer to simulate the Eventbuilder, and because of the restricted disk space on this device, a copy of the large Eventfiles to local disks seemed not to be recommended. So we decided to read these files event for event via the network, directly from the IBM disk. The only network protocol, for which drivers were available on both, the IBM and the Macintosh was the TCP/IP[10]. Unfortunately both machines used different programming interfaces, so it was necessary to define a unique interface to make code portable. Taking pattern from the IBM driver name *K-NET*, our software part was called *Farm N-NET.*

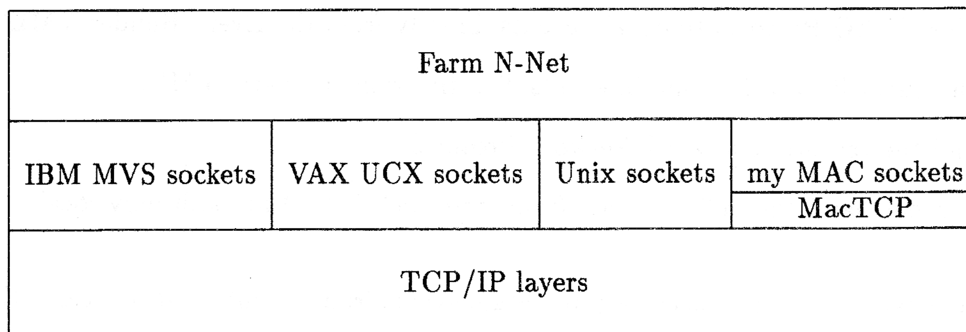


After a year, IBM released the socket interface for TCP, which is the Unix Standard for this protocol. So I also changed my interface to the sockets, with the advantage, that now all Unix Systems were also covered. Some time later, DEC followed with the UCX socket software for VAXes. In the meantime Apple released a new interface to TCP, but unfortunately not yet using sockets. The easiest response for me was to write my own socketinterface for the Apple software, which leads to the following final configuration[PAT 93].

[9] Transport Control Protocol/Internet Protocol

[10] For Details, see section *Network Details*

[11] National Supercomputer



This scheme had the additional advantage, that RPC[12] calls could be implemented on Macintosh, which was necessary to install the network part of FPack[13] on that computer type.

With the development to the sockets, the usage of the N-Net seemed to be unnecessary, but it provided some other useful features.

- N-Net is packet oriented. The number of bytes, received by one receive call exactly corresponds to the number sent within one send call, while TCP/IP is free to spit and merge the different packets.
- Additional informations about the exchanged data packet is available, like a packet identification number, and the data type.
- According to the data type specified, the packet is converted, in the case were different data representations come into account, like ASCII versus EBCDIC, big Endian versus Little Endian and so on.
- Automatic memory allocation is performed for incoming packets, which avoids the waste of memory space.

Based on N-Net I defined a simple protocol which was able to read and write events from/to remote computers[PAT 90]. It was realized in terms of the client server mechanism. On the service computer, lets say the central IBM, a *Global Server* is permanently listening for incoming requests. If the client, the computer which wants to read a file, needs to do this, it sends a user identification together with its own network address to this Global Server. The Server checks, if the identification data is valid for this computer, and starts the so called *Private Server* under the account of the requesting user[14]. This became necessary to ensure data security. The private server then connects to the client and awaits further commands, which are :

- Open, create and close a file.
- Read and write an event.
- Skip an event.
- Rewind the file.
- Database access, see 6.4.3.

Currently two of these servers exits, one on the IBM, and one on the SGI cluster, because these computers nearly hold the total H1 disk space.

Though we are now able to send events over the network, the possible number of protocols increases to three.

[12]Remote Procedure Calls, an upper layer of TCP/IP

[13]The offline filesystem

[14]Fpack adopted this scheme some time later

I. The CDAQ Protocol to obtain events directly from the Event Builder VME,

II. the Farm Protocol to obtain events directly from the Farm VME.

III. and the recently discussed Network Protocol.

On the other hand the online events from farm and CDAQ are still only locally available. What is missing ?

- A general interface is necessary to obtain events from the different sources with the same function calls.
- Two gateways are needed which take events from the farm and the CDAQ and distribute them via the network on request.

6.2 The Online Event Servers

The Eventservers are realized as Macintosh applications. One is connected to the CDAQ VME crates and the other to the farm VME system. Both have of course an Ethernet connection. From the network point of view they look like the previously described servers on the IBM and SGI, which means that the clients remain unchanged and only have to specify a different hostname. The Mac servers only differ in the interface to the VME protocol, but they are conceptually identical. Both are able to serve multiple clients simultaneously. Because of the restricted multitask capability they don't start a private server, but they wrap around and handle the next incoming request. On startup, both try to read an initialization file from the Farm MIPS/120 via TFTP[15]. This file contains the following types of information.

- Nodenames or domains from which access is permited or denied. So, one can for example enable the connection from the whole DESY except the central IBM. In addition, single nodes and whole domains can be declared as trusted hosts, which permits an access without user identification.
- Usernames, the corresponding passwords and their access priorities. This is only necessary for the farm server, because it has the possibility to execute all farm functions. A feature which is quite dangerous without password protection.
- Internal steering parameters, like maximum number of users, external and internal buffer requirements, address of the CVT or the CDAQ FEB and so on.

The CDAQ server is connected to a single CDAQ FEB and uses the first word (4 bytes) of the eventbuffer to communicate with the CDAQ. This word contains the number of bytes following this word, if the full event is available. This information is of course written by the CDAQ after copying the event into the buffer. If the server needs the next event, it only has to set this word to zero, which is the signal for the CDAQ to copy the next one. The eventserver distributes the same event until one client would get it again, then the next is taken from the CDAQ. On the one hand this reduces unnecessary event copying, on the other hand it makes sure that a client doesn't get the same event twice. Specifications concerning event properties are ignored by the CDAQ server.

The farm server has two servicemodes, depending on the priority of the engaged client. In low priority mode, a specification of event properties isn't possible, and all connected clients in this mode share the same event buffer. Each high priority connection manages its own VME buffer, and requests are send as individual requests to the mailbox of the Output FIC. All possible event selections are accepted. With this feature the online event display can monitor special classes of events, for example only those, which had been rejected, by the Filter Program.

[15]Trivial File Transport Protocol, an application level TCP protocol

6.3 The passive Farm VME OS

Although the Farm Request Protocol provides a flexible handling of requests it produces some new problems. Each participant of the communication needs some memory out of the pool of the different farm memory cards. The users mainly are

- One or two Online Event Displays directly connected to the Farm.
- The Farm Eventserver with one buffer for all low priority connections, and one buffer for each high priority one.
- The Farm simulation process running on the Sparc Sun.

The available memoryportions are

- Two memorycards à 2 Mbytes
- 128 Kbytes memory for each VIC card.

A preassignment of buffers to the different processes would waste memory because not all of the tasks are running permanently, and on the other hand it would restrict the system to a fixed number of assigned tasks.

6.3.1 The VME Memory Allocation

To solve this problem I have written a VME memory allocation system. On farm startup all available memory portions have to be made known to the system. To each portion the following attributes are assigned.

- The start address and the length.
- A unique name.
- If the portion should be divided into fixed blocks or not.
- A priority to reserve memory to special applications.

The *fixed* attribute means, that the total portion is divided into equally sized parts. They need less management overhead and are used as request areas for FIC requests. From the other type of memory, which is declared *variable*, one can request parts of any length up to the total buffersize. For a memory request, one has to specify the length and the priority, the name and the type of the memory are optional. The system then tries to find a part, with matches the specified requirements. This part is locked and belongs to the requesting process until it is freed again. It can be used as eventcontainer for a FIC event request. For all these applications again semaphores are required, because several different tasks could try to allocate memory at the same time.

6.3.2 The VME Task Monitor

Together with the allocation of memory, another problem arises. If an application aborts, or for some other reason is not able to free the allocated memory, this part would be wasted until the next full system start. To avoid this, a task, intending to get memory, has to register itself to the VME system. In this register call, a process name has to be specified together with the highest priority the task is willing to get memory for. The registerfunction returns a pointer, and only with this pointer an allocation of VME memory is possible. If the tasks finishes, it has to call only one close function, and all memories belonging to it will be returned.

If the process aborts, before it could call the close function, the entry remains in memory as zombi. To make it possible to detect such a zombi, a running task has to call an *alive routine* regularly, which updates an internal timestamp. With a shell program one can observe the different tasks and its last update and if the idletime increases, it is possible to kill the zombi by hand, without affecting the rest of the system. With this kill, all memory allocated to that process is release again.

If, for some reason, a process entry of an alive task is cleared, this task should be enabled to recognize the problem. So, a function is provided, which returns the unique process identification number of the task, belonging to that memory. The corresponding task only has to compare it with the ID returned by the registercall to know, if the memory does still belong to it. This precaution is also used by the I/O FICs before they copy something into an external buffer. For that reason, the requestarea must always contain the task ID of the process, sending the request.

6.3.3 Again the Event Server

All the above listed features, like registering a task, allocating VME memory, writing and reading of VME memory or using it to send a request to one of the I/O processors, are not only available for machines directly connected to the Farm VME system. Instead, the Farm Event Server also maps appropriate requests arriving as network commands to the corresponding VME activities. This is true not only for basic actions like reading a single word, but there are also complicated commands which are mapped to several VME accesses to reduce network traffic. The best example is the already discussed eventread via network from the farm. This single read triggers the following actions-:

- Allocation of a large memory portion, for the event.
- Allocation of the request area.
- Filling the request area with the chosen selections.
- Sending the address of the request area to the mailbox of the Output FIC.
- Observing the answer field of the request area to determine the arrival of the requested event.
- Sending back the event to the requesting task.
- Deallocation of all memory parts, if no longer needed.

6.3.4 The Unification

Now we have all tools available to construct a system, which provides us with events and other objects, without the need of the knowledge of the underlying protocols or activities. This part of software is the highest level of the *Passive Farm Operating System.* Concerning the reading of events, exactly the same calls are able to read events from IBM or SGI disk, out of the CDAQ datastream or the Farm datastream via network or directly connected to the Farm VME system. Reading and writing from/to Farm memory can also be done directly from the Farm computers or from other computers connected to the TCP network, without changing the code. Even the small Farm OS shell, which is able to monitor and control most of the farm functions, can run anywhere.

6.4 The Farm Startup

6.4.1 The Installation Language

There is still another open question. How does the Farm OS learn about the different parameters, which are necessary to install the initial tables.

The Farm OS shell provides a command to execute an initialization file containing the appropriate parameters. An example part is plotted below.

```
//
//   Farm Startup File
//
//          General Parameters
//
  etc = {   BuffersPerRAID   = 2  ;  // the number of io buffers per RAID
            maxProcessors    = 32 ;  // the maximum number of RAIDs
            maxProcesses     = 16 ;  // the maximum number of OS tasks

  } ;
//
//     Addresses of VME accessible FIC memory
//
 fics = {  input    = 0x600000 ;
           output   = 0x600010 ;    } ;
//
//     The available memory portions
//
 memory = {
//
//               FEB buffers ( information for FIC only )
//
         input = {    start    =    0xE00000 ;
                      type     =    vsb ;    // or VME for simulations
         } ;

         output = {   start    =    0xE00000 ;
                      type     =    vsb ;
         } ;

//
//              Memory areas, usable by all VME tasks
//
//               I)        buffers on the memory card
//
         buffer.var.1 = {
                              name     = mc.var1 ;
                              start    = 0x680000 ;
                              size     = 0x07FFF8 ;
                              type     =  variable ;
                              priority = 15 ;
         } ;
//
```

```
//            II)        buffers on the VIC local memory
//
        buffer.vic1.fix = {
                                 name     = vic1.fix;
                                 start    = 0x0e88000 ;
                                 size     =    0x8000 ;
                                 type     = fixed ;
                                 blksize  = 100 ;     // small blocks for
                                                      // the request areas
                                 priority = 15 ;
        } ;

  } ;  //    End of memory structure
```

Fortunately it is quite easy under Unix to construct compilers, which understand the above small language. The tools[SFR 85, KOP 88] are *lex* and *yacc* , an abbreviation for *Yet Another Compiler-Compiler.* *lex* is the lexical analyzer. It collects the single characters of the input file and builds tokens like *Numbers, Names and Operators.* These items are the so called *Terminals* for the next step, the grammatical analysis, done by *yacc.* To describe the new constructed language, the *Backus Naur Form, BNF* is used. With this input, *yacc* produces C code which can be extended to the final compiler.

The BNF for the Farm Description Language actually is quite simple :

```
     program  :  definitions

 definitions  :  definition
              |  definitions definition

  definition  :  Keyword '=' assignment ';'

  assignment  :  Constant
              |  HexConstant
              |  Name
              |  '{' definitions '}'
```

Where the terminal objects are :

Terminal	description	characterset
Constant	decimal constant	0 – 9
HexConstant	hexadecimal constant	0 –9 , a – f , A – F
Name	character string	digits letters '.'
Keyword	...	...

After the compilation of the Farm Startup File, the following directorylike structure is produced.

- **etc**
  - BuffersPerRAID = 2
  - maxProcesses = 16
  - maxProcessors = 32
- **memory**
  - **buffer.var.1**

    - * name = mc.var1
    - * start = 0x0e8000
    - * a.s.o
  - **buffer.vic1.fix**
    - * name = vic1.fix
    - * start = 0x680000
    - * a.s.o
  - a.s.o

Each installation module can obtain the values of the corresponding keyword out of this directory tree.

6.4.2 The Startup Procedure

After the Farm Startup file produced the already discussed tables inside the different memory cards the following steps have to be performed.

- The IO program has to be loaded into the Input and Output FIC.
- The Filter Program has to be loaded into the RAID boards.
- The run constants have to be obtained from the Database on the central IBM.

The IO program is written in C under the Macintosh MPW[16] environment. A modified c-library enables the I/O FICs to execute the linked programs. They are loaded directly from the Macintosh disk into the FIC memory via the VIC connection, and started.

The Filter Program consists of the C-steering and the FORTRAN Filter modules. All parts are assembled on the M/120 Computer and linked together with a standalone library, which enables the executable code to be run on the RAID boards. This code is also loaded with a Macintosh program into the different RAID memories, and started.

6.4.3 The Database Access

The connection of the Farm System to the central H1 Database has two reasons.

- The *Module Steering File*, which coordinates the actions of the different filter modules and its results and histograms, has to be made available to the offline analysis, to keep track of the decision chain. So, whenever parameters of this file change, the new file is assigned to the appropriate run number and send to the Central Database on the IBM.
- The same has to be done by the subdetectors, if detector specific constants changed. For the corresponding analysis modules, running in the Filter Software, this means, that they have to recognize these changes as fast as possible.

Both actions are combined to one remote database access. First the *Global Server* on the IBM is requested to start a special *Private Server* which enables the database interactions. Then the *Module Steering File* is sent to the server, which pushes the information into the database. This information includes a list of data objects needed by the different filter modules. These items are extracted out of the database and send back to the farm client.

The described job can be done by either the control Macintosh or the M/120 Minicomputer. This scenario makes sure, that one can always determine, which *Module Steering File* was used by the Farm, and which constants version was taken by the different filter modules.

[16]Macintosh Programmers Workshop

6.5 The H1 LAN

Beside the already discussed types of information, an other one had to be covered with a more general concept. It mainly consists of small messages or data to be displayed on special Computers far away from the datataking device. The expected datarates should be in the order of some hundred Kbytes per minute. Some other constraints had to be fulfilled. A breakdown of one of the participants should not influence the behaviour or its counterpart and the programmers interface should be easy to use and available in the most popular computer languages. Because the *Farm N-Net* was already available, I decided to construct a new layer above this level. It was called *H1 LAN*[PAT 91, PAT 92]. It uses the advantages of the underlying parts, like the data convertion of *N-Net* and the reliability of the TCP/IP. The main extentions are :

- *H1 LAN* opens a new connection for each message to be send, and closes it immediately after the delivery.
- The received message is copied into an internally allocated buffer and send to the application layer in form of an event.
- Automatic installation of a network listen port is performed. This portnumber, together with the network address is send with each packet, which makes it easy for the process to send an answer.

Unfortunately the features provided by the standard socket interface were not sufficient for Macintosh applications. The main reason is, that the Mac is no real multitasking system. Instead, control has to be given back explicitly to the Operation System, if the task is going to be idle. This makes it necessary to avoid all actions, which wait for the completion of tasks. So I have completely rewritten the *N-Net* and the *H1 LAN* for Mac, to meet all its specific requirements. The point is that none of the new routines wait until an action is completed, instead it returns immediately and sends failure or success in form of an event to a *nextEvent* Routine which has to be called frequently, a well known habit for Mac programmers. So, the final *H1 LAN* scenario looks like :

<table>
<tr><td colspan="4">H1 LAN</td><td>H1 MacLan</td></tr>
<tr><td colspan="4">N-Net</td><td rowspan="2">Mac N-Net</td></tr>
<tr><td rowspan="2">IBM MVS sockets</td><td rowspan="2">VAX UCX sockets</td><td rowspan="2">Unix sockets</td><td>my MAC sockets</td></tr>
<tr><td colspan="2">MacTCP</td></tr>
<tr><td colspan="5">TCP/IP layers</td></tr>
</table>

For binary data exchange this software is used between Macintoshs belonging to the same subdetector, for example the *Central Trigger Mac* permanently sends data to the *Trigger Display Mac* in the H1 Control Room. In the beginning of the experiment, the luminosity system sent Lumi Informations to the HERA Control Room. The slow control uses *H1 LAN* to connect VAXes with OS/9 stations.

Another type of message, mainly human readable one are exchanged in a standardized way, using character stings. They are automatically converted, so that also the IBM can participate. The request consists of digits and commands at fixed string positions[PAT 91]. A heavily touched server for this kind of data, is the *H1 Central DAQ Control Mac.* It answers two types of inquiries.

- The current status of the CDAQ, including informations about most important subdetector parts.

- The run summary for a specific run number, containing informations, like the number of taken events, the availability of detectorparts, the farm rejection ratio, the run comment of the CDAQ shift and much more.

The latter request is send by the IBM datalogging job after each run, to fill these informations into a human readable database on the IBM. The first request is send with the *H1CDAQ* command on the IBM and with the CDAQ desktop program for the Macintosh. Just for fun : This desktop obtained the H1 CADQ informations even from a Macintosh at the SSC laboratories in the US, as tests turned out.

On the IBM and on the central VAX cluster, servers distribute informations about the HERA machine status.

All these services can be tied together in the form of a directory tree. A central server, currently I'm using the Farm Sun, knows the address and the public request port of the subdetector servers. To climb to the next tree, one only has to send a directory request to one of these tasks, which responds with the possible inquiries for that section, or references to other directory servers.

6.6 The Alternative

Though I tried to cover most of the data producers and consumers with a more or less unified interface to the application layer, the work grew up to a patchwork. So, the aim should be to use the same data handling methods, at least between the different subdetectors and the outer world. The advantages are obvious. Today, a lot of time is wasted by each online group to write code for the readout the controlling and the display of data and tasks. Jobs, which are in principle very similar.

One reason for the missing unification could be, that at the time, the detector was designed, it was obvious to everybody, that the hardware of the subsystems only can be planed together with the global frame of the detector hardware. An understanding, which seemed not to be available for the software of the project.

Better concepts already exists from other experiments. Only one example is the *Glish*[PAX 92] software package from the *Lawrence Berkeley Laboratory.* This work describes a *software bus* connecting various computers. A loosely coupled distributed computer system. The convertion for the different data representations is done on the fly. The different computers provide several services like *data measurement*, *displays*, *fast calculations* and much more. The servers and clients describe their tasks with the help of a C-like language, the *Glish script*, supporting data exchange specific commands.

7 The Filter Program

The actual decision, if an event is worth being stored or not is done by the *Filter Program*, which runs in 'N' identical copies on the *RAID processors.* The events are delivered and assembled by the *FIC I/O Tasks.*

7.1 The Module Steering File

In principle, the *Filter Task* is a reduced *Reconstruction Program.* Like the offline reconstruction, it consists of different modules, calculating the subdetector specific quantities. The aim is to adopt the offline modules as much as possible without changes.

Each module takes one or more data structures[17] out of the stream, performs its calculations and returns the result in form of a structure into the data pool, were the new information is available for the next module. This implies, that a group of functions is able to run independently, while others depend on the output of previously executed ones.

In addition, a module can optionally provide decision relevant quantities, which are accessible from the module steering layer. With the help of these values, the steering can always decide to accept or reject the event. Then, the processing chain is immediately broken, and the decision is sent to the output processor.

To obtain a maximum of flexibility concerning the module execution sequence and the cut off parameters, Alan[CAM 92] has written the steering in a way, which makes it possible to predict the decision behaviour with the help of a description file without recompiling the Filter Code. This *Module Steering File* has to provide the following informations.

- In the first part, all modules have to be defined, together with the quantities they produce, and the modules they depend on.
- The second section describes the accept and reject conditions, using the above defined quantities. This part implicitly determines the order in which the modules are executed. A module is only called, if one of its output variables is used in the currently active trigger statement, or if it is declared to be an ancestor of a module, which is going to be executed. Even if an output variable is used several times, the corresponding module is never called twice.
- In two special parts, modules are listed, which should be called after a successful decision, depending on accept or reject.
- Keywords can force the system to execute all trigger statements even if a decision had been taken, or to call modules if there is still some time between now and the average execution time of the Filter Program.
- The last section defines histograms of the produced decision values and of informations concerning the time behaviour of the different modules.

7.2 The Modules

In October 1992 there were mainly 9 *Reconstruction Modules* active in the *Filter Program.* Only 6 of them provide variables, which influence the L4 trigger decision. A special module, the *QT Analysis* is necessary for the reconstruction of tracks inside the two *Central Jet Chambers* and the *Forward Trackers.* Charge division is used to calculate the particle position with the help of the accumulated charges at both ends of the sense wires. The other modules provide the following quantities for the decision process :

[17] This structures are called **BANKS** and are an assembly of the atomic data types *Integer, Real and Character,* in linear form or in the form of tables

- The **L1** Module unpacks the L1 trigger bits and assigns them to the variables. The most important are :

  L_L1_AllBunch indicates, if the trigger occured during a bunch crossing.

  L_L1_Diode is a trigger of the *Backward Electromagnetic Calorimeter.*

  L_L1_Muon_FEc indicates a muon track in the *Forward Endcaps.*

  L_L1_eTag states the detection of an electron in the *Electron Tagger.*

  L_L1_TrkMsk is a trigger of the *Central Tracker.*

- The **BEMC** Module only provides the **R_BMC_Diode** variable, which holds the energy fraction of the hottest Diode in relation to the total *BEMC* energy in the same stack.

- The **IRON** Module reconstructs muon tracks in the *Instrumented Iron* and provides the number of tracks found.

- The **FWD** doesn't yet reconstruct anything, but only assigns the number of hits found in one of the *Forward Radial Chambers* to **I_FWD_NhitFrr3**

- The **Z-Vertex** Module is used to decide if it is worthwhile to run the time consuming *CJC* code. The method is, to build combinations of hits in one segment of the *Central Jet Chambers* and to enter these pseudotracks into an histogram. The provided quantities are

  I_ZVT_nZVTXray : the number of combinations

  R_ZVT_ZVTXhoriz : the fraction of nearly horizontal pseodotracks

  R_ZVT_ZVTXfrac : and the fraction of entries which touch the z-axis at a distance greater than 75 cm of the nominal interaction point in the electron direction.

- The **CJC** Module provides a full track reconstruction in the *Central Jet Chambers.* It is the most time consuming part and needs about 125 % of the average execution time for one event, but it is only called for 70 % of them. It produces the following values :

  I_CJC_nTrackZdca : The number of tracks with more than 8 hits. The distance of closest approach according to the z-axis must be below 2 cm, and the point of contact of the dca and the z-axis must be greater than $-50cm$, where negative distances point in the electron direction.

  I_CJC_NegTrkZdca : Same as above, but only for negative tracks.

  I_CJC_nUpstrTrack : The number of tracks with more than 14 hits. The distance of closest approach according to the z-axis must be below 2 cm, and the point of contact of the dca and the z-axis must be greater than $-100cm$. In addition, the tracks must come from upstream.

Figure 15 shows the reduction fraction of the different *Farm Trigger Statements.* From top to bottom the reject trigger statements are :

1	L_L1_Diode	$\wedge$	R_BMC_Diode	$>$	0.95			
2	L_L1_eTag	$\wedge$	R_ZVT_ZVTXhoriz	$>$	0.5			
3	L_L1_eTag	$\wedge$	I_ZVT_ZVTXray	$=$	0	$\wedge$	I_FWD_nHitFrr3	$= 0$
4	L_L1_TrkMsk	$\wedge$	R_ZVT_ZVTXhoriz	$>$	0.5			
5	L_L1_TrkMsk	$\wedge$	I_CJC_nTrackZdca	$=$	0			
6	L_L1_TrkMsk	$\wedge$	I_CJC_NegTrkZdca	$=$	0			
7			I_CJC_nUpStrTrack	$\geq$	3			
8			0 $<$ I_CJC_nUpStrTrack	$<$	3	$\wedge$	I_CJC_NegTrkZdca	$= 0$

The greatest step is achieved by requiring at least one **nTrackZdca**. The numbers are calculated for a run at the 19 of October 1992. The integrated luminosity was $60.6\mu b^{-1}$ and about 37500 events were produced as input for the *Filter Farm.* Figure 16 points out the behaviour of the Farm as a total. Slightly more than a quarter of the input events were accepted. From the rejected events, about 1% was also kept for monitor purposes. Both, the monitor and the accepted events were processed by the full reconstruction task.

Two parts of the pie chart are of special interest and need further investigations.

About 50% of the *Farm Accepted* events were classified as beam gas events or not classified at all by the L5 Reconstruction Program. The aim is of course to filter these events at farm level.

A more serious problem are the 15% of the *Farm rejected* events, which would have been accepted by the L5 Trigger level. One has to show that none of them was a real physics event. This can actually be done. For example a third of them had been rejected because of the **R_BEC_DIODE** ≥ 0.95 , which means that one BEMC Diode holds more than 95% of the energy of the corresponding stack, which indicates a BEMC malfunction. Another part would have been accepted by L5 because the definition of the **NtrackZdca** is different for the reconstruction, where the $-50.0cm$ cut for the point of contact with the z-axis is not applied, which means, that for the Reconstruction, a good track can also start below $-50cm$ of the interaction point.

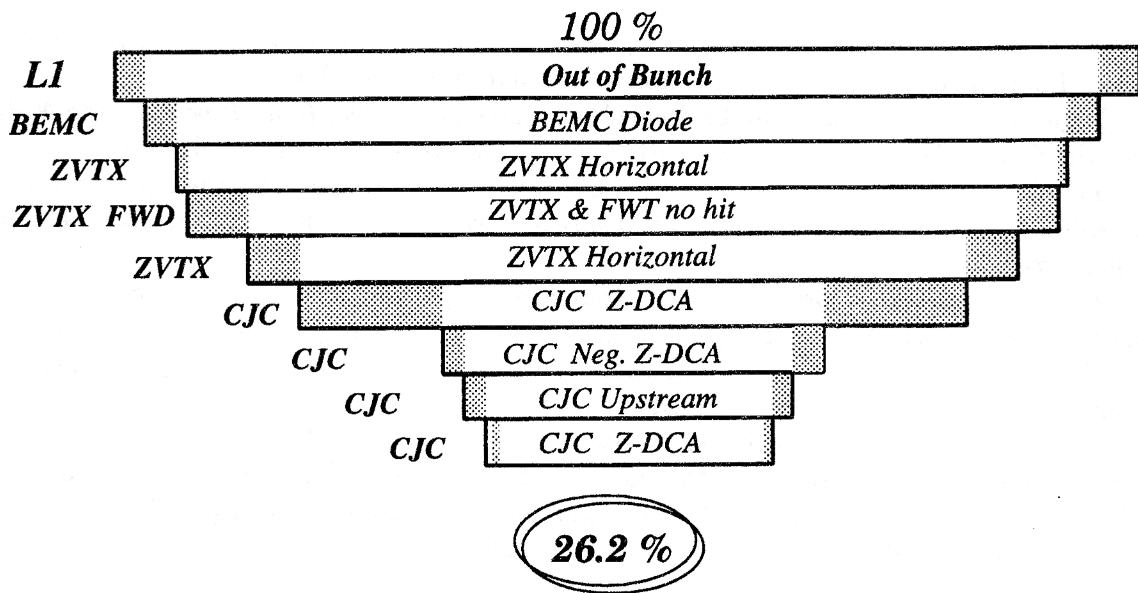


Figure 15: Event Reduction according to the Modules

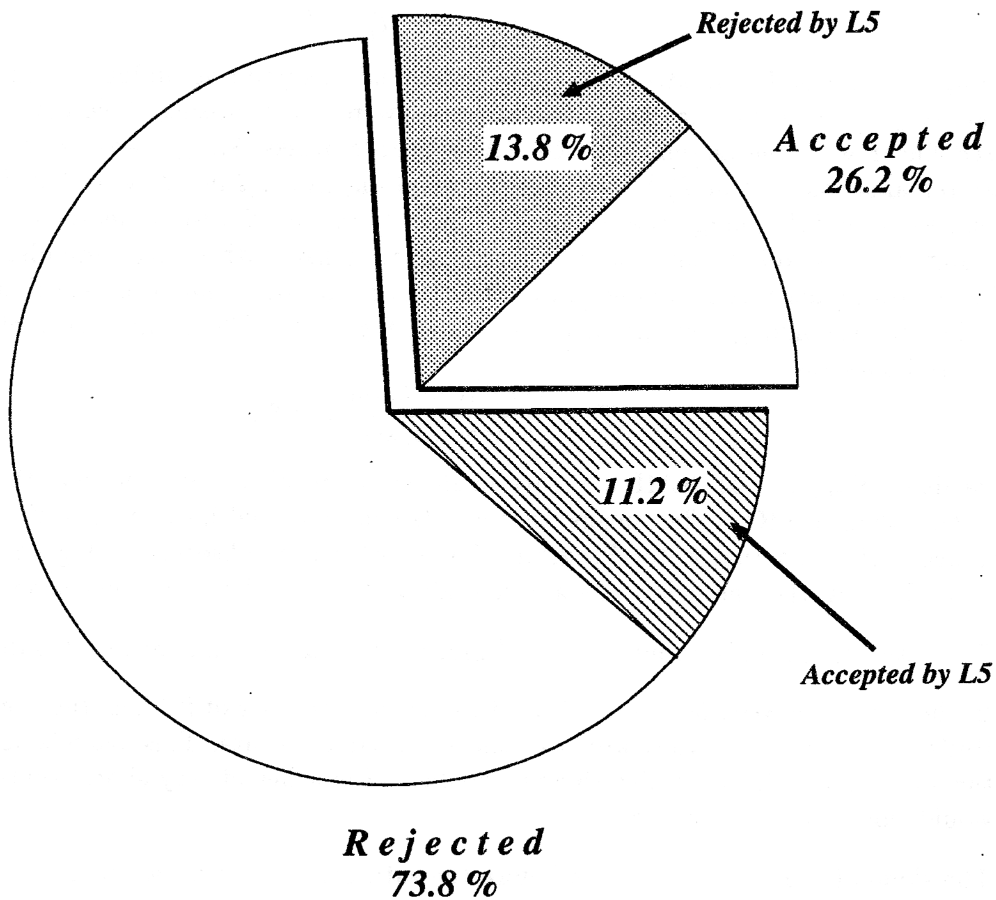


Figure 16: The Rejection Pie

8 The Performance

To get an impression of the performance behaviour of the *Farm* as a device in the event processing chain, I have written a tool, which is able to simulate the event flow.

Logically the whole *Farm* can be divided into three parts. The *Input Processor*, the *N RAID boards* and the *Output Processor*. For each part, a maximum event rate can be calculated.

The Output Processor doesn't need to be taken into account, because the rejection rate will always exceed 50%, which means, that the Output Task only has to shuffle less that half the amount of the *Input Processor*.

The maximum transport rate of the *Input Processor* T_c is solely dependent on the inverse ratio of the event size and the VME copy speed, which is about 6.5 Mbytes/sec within a crate and 6 Mbytes/sec via the crate interconnects. The constraint to reach this value is, that the processing device which follows, is able to consume all delivered events.

The rate of the actual processing device is $(T_p/N)^{-1}$, where T_p is the average time used for one event by one processor, and N is the number of RAID processors. Again the condition has to be fulfilled that the preceding *Input Processor* guarantees to produce the required event rate.

The total throughput is in the order of the smallest of the two values :

$$\text{Total Rate} = \min(\ (T_p/N)^{-1},\ T_c^{-1}\ )$$

The interesting point is the transition between these values, which means $T_p/N = T_c$. For a given event size (that means copytime T_c) and a given processtime T_p, the critical number of RAID boards would be

$$N_{critical} = T_p/T_c$$

As first approximation I simulated the behaviour around this point assuming T_p and T_c to be constant, which actually is extremely unrealistic. So the results were not surprising. Below $N_{critical}$, the eventrate increased linear with the amount of used RAID boards unless $N_{critical}$ was reached. From that on, the rate remains constant at T_c^{-1}. Even the assumption, that T_p and T_c were flat distributed around the mean value, didn't change the response of more than 2%. So, to be nearer to reality, I modified the simulation to be able to distribute T_p and T_c exactly like both values were distributed in a given run. The following discussion is based on the total execution time and event length distribution of run 33846. The corresponding histograms are shown in Figure 17. The mean values of the distributions are $T_p = 186.8ms$ and

$$T_c = \frac{8688\ Words * 4}{6\ Mbytes/sec} = 5.8ms$$

which leads to $N_{critical} \approx 31$ RAIDs. Figure 18 shows the result for one, two and three *IO Buffers* per RAID. More than one IO Buffer means, that events can be copied into and out of the RAID, without interference with the actual Filter Program. The effect can be observed in Figure 18. The solid line reflects two buffers, the line below one and line above three of them. Summarizing the results :

- The saturation rate is about $T_c{}^{-1}$ and independent of the number of IO Buffers.
- In the region between 50% and 100% of the critical number of RAIDs, the use of more than one Buffer can save up to two Boards to achieve the same IO rate. This again is no surprise, because the additional buffers are able to hold the data of bunches of very short events, which otherwise would slow down the system.
- The slope of the curves decreases before the saturation point is reached.

The last point is surprising. The left plot of Figure 19 shows the load per RAID (dotted line) in addition to the event rate for two IO buffers. The x-axis is labeled with the relative number of RAIDs

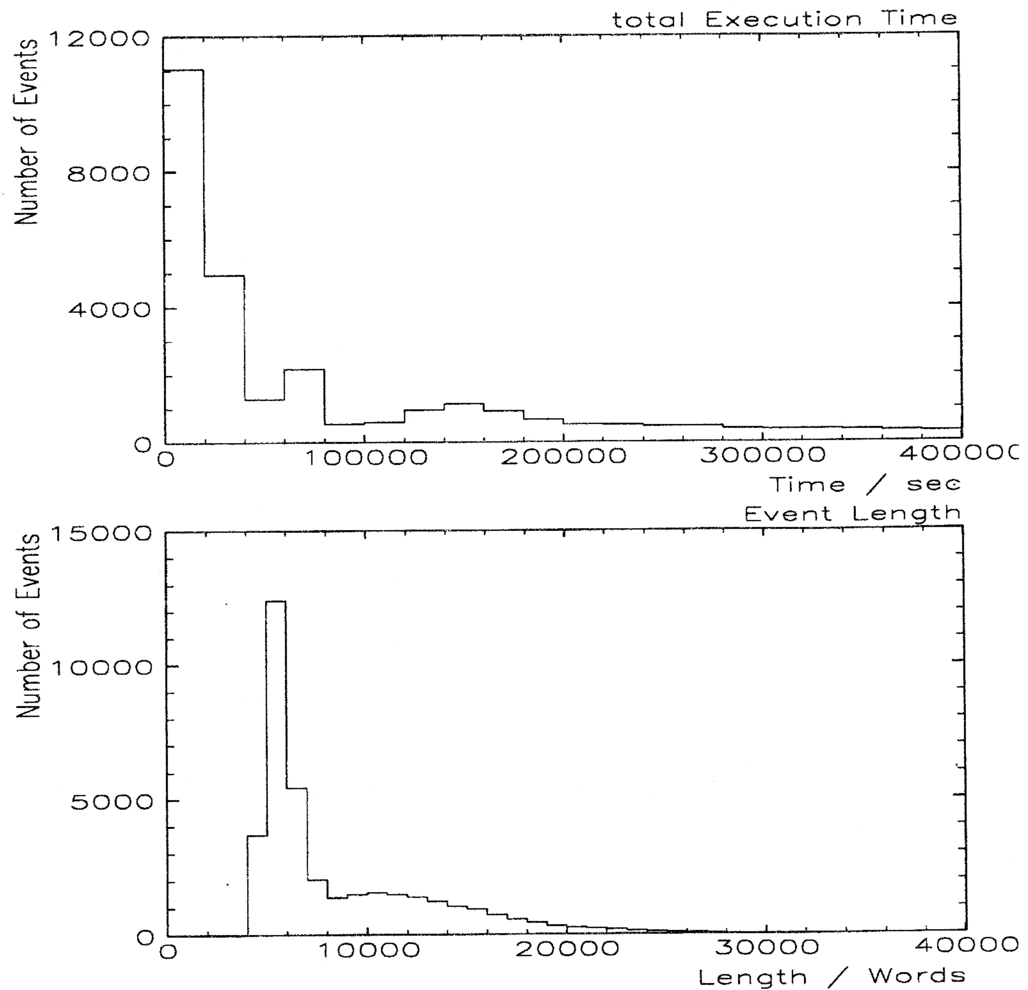


Figure 17: Input Event Length and total Execution Time

according to $N_{critical}$. Although the load per RAID decreases to about 96% for relRaid = 1.0, the throughput doesn't come up to it's final value.

Actually this effect is no longer of interest, because we have to go beyond the current saturation value anyway. This is achieved with the help of a second Input Unit. The results are superimposed on the left of Figure 18. It can be stated, that the load per RAID and the event rate again behaves linear under two constraints :

- Both Input Units have to use separate VSB links to the central DAQ.
- The Input Units have to serve disjunct RAID crates.

Otherwise bus interferences would reduce the data transfer speed.

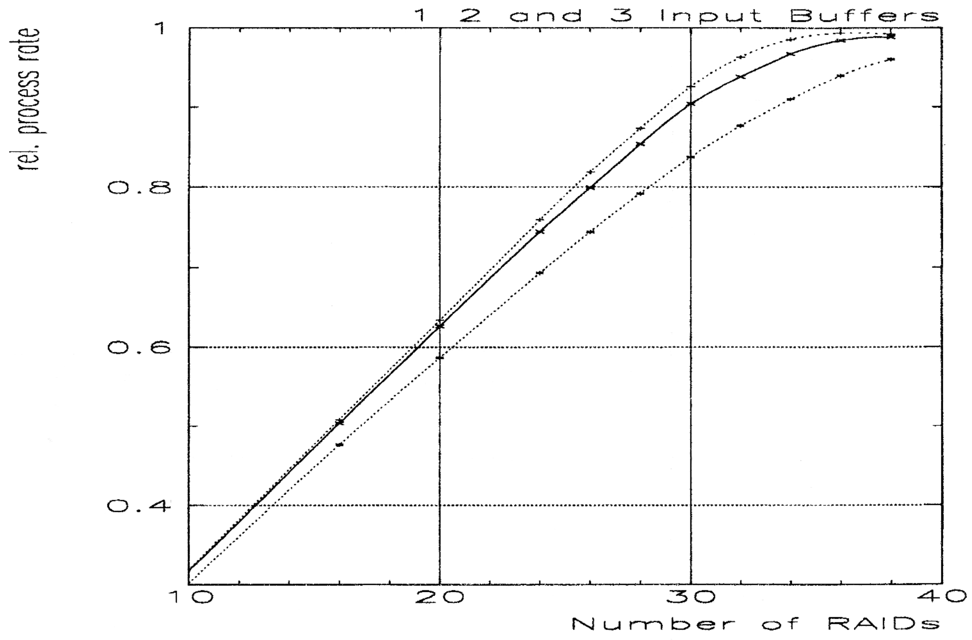


Figure 18: relative process rates for 1, 2 and 3 I/O Buffers

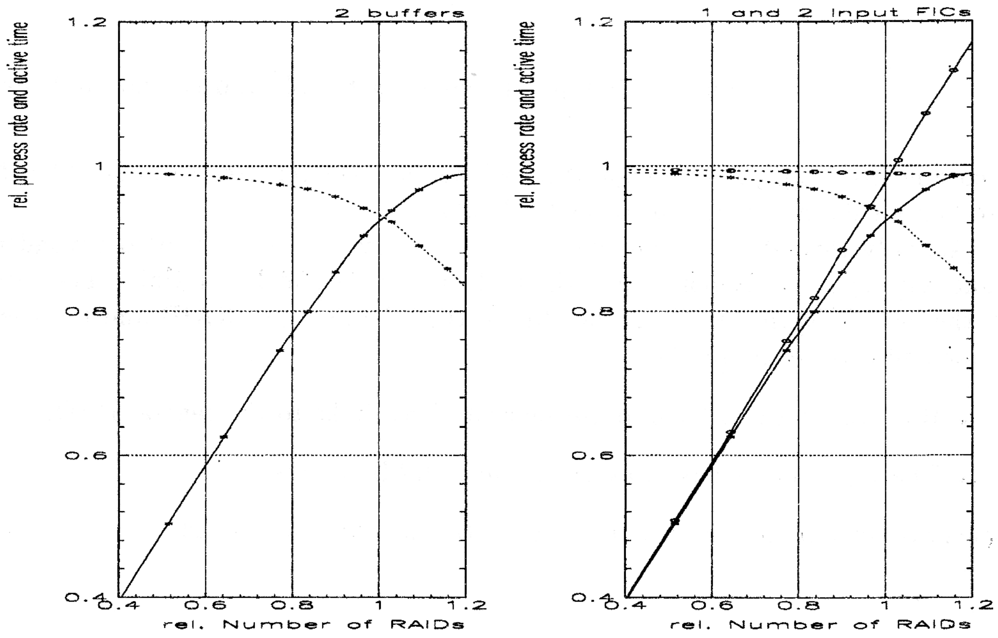


Figure 19: Performance for 2 Input Processors

A Network Details

A.1 TCP/IP and the OSI Reference Model

To understand the way the farm uses the network to exchange data, one should first have a brief look to the underlying transport protocols.

At DESY, several different protocols are in use, which mainly connects machines of the same kind, like

- **SNA** for IBM host computers.
- **DECnet** for all machines of the Digital Equipment Cooperation.
- **AppleTalk** for Machintosh Computers and Macintosh Disk Severs.
- **InterLink** for the connection of IBM mainframes and DEC computers.

The **TCP/IP** protocol is a more generic one. It is known to nearly all machines participating the network.

To have a unique description tool for all the different network languages, the *International Standardization Organization (OSI)* defined a 7 layer model, which covers most of the different cases. Each layer offers its service to the upper layer and uses the service of the lower one. It is not permitted, that a level makes use of informations provided by an upper layer.

Application
Presentation
Session
Transport
Network
Link
Physical

From bottom to top, the layers have the following responsibilities.

Physical This part describes the physical medium, which transports the information. It ranges from a simple telephone line using the *RS232 protocol* via coaxial cables like the *Ethernet* and optical fibres like the *FDDI Ring concept* up to Satellite Connections. Beside the hardware, it also has to define what a logical *One* and *Zero* means for that device.

Link The link layer knows about the connections of the transport medium and the host computers. The atomic item is no longer the bit, but a packet, whatever that means for the specific medium.

Network The network layer doesn't longer depend on the restrictions of the hardware. It can access nodes, which are not directly connected to its own section. *Bridges* and *Gateways* are necessary to interconnect different hardware sections. This level creates virtual pathes between nodes. It does not yet connect processes.

Transport This layer is not only able to address the remote host, but it also knowns about the remote communication partner, which usually is a task at the remote machine. The link can be connection oriented or not. If the first case, a so called virtual circuit is created, which is used to deliver the data packets. Using connectionless protocols, one has to specify the remote partner with each packet. The transport layer should also guarantee an errorfree delivery of data, a feature which includes the error detection and error recovery.

Session The Session layer controls the data exchange, and determines actions like the change of the communication direction and the relink after an abort.

Presentation In the presentation layer the representation of the different datatypes is defined, to enable a dataexchange between computers, not using the same data representation. This includes the conversion of ASCII EBCDIC characters, VAX IBM and IEEE floating point numbers and the big to little endian convertion.

Application The Application layer defines high level application protocols for services like database queries and file exchanges.

There is no unified opinion how to map the ISO layers to the TCP/IP levels. The next figure shows one possibility[COM 91].

<table>
<tr><td>Application
Session
Representation</td><td>Virtual
Terminal
TELNET</td><td>File Transport
Protocol
FTP</td><td>Electronic
Mail
SMTP</td><td>Name
Server</td><td>Network
File
System</td></tr>
<tr><td>Transport</td><td colspan="3">Transmission
Control Protocol TCP</td><td colspan="2">User Datagram
Protocol UDP</td></tr>
<tr><td>Network</td><td colspan="5">Internet Protocol IP &
Internet Control Message Protocol ICMP</td></tr>
<tr><td>Link
Physical</td><td>ARPANET</td><td>Satellite
Network</td><td>X.25</td><td>Ethernet</td><td>Token
Ring</td></tr>
</table>

Above the transmission protocol, the OSI model distinguishes between Session, Representation and Application, which is not done by TCP/IP. All these levels are combined to the application layer. To become more familiar to TCP/IP some additional expressions have to be explained.

The **ARPANET** *Advanced Research Project Agency* was the first TCP/IP network backbone. It was supported by the US Military. Today its a subnet of the worldwide Internet.

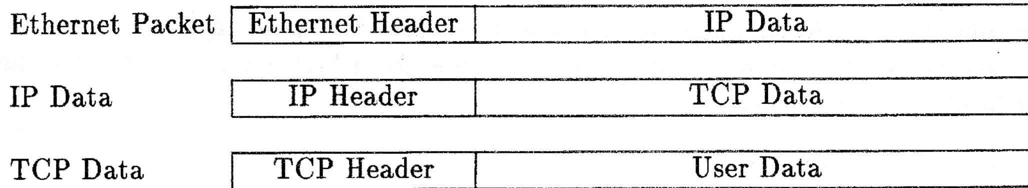
The **Internet Protocol** builds the Node to Node connection, using unique hostnumbers or hostnames. The IP number can always be represented in four bytes. One can mainly distinguish 3 different classes, A to C , depending on the size of the subnet. Large subnets, belonging to class A, can have up to 2^{24} hosts and the smallest subnet, class C can only hold 256 different machines. Because of the 32 Bit limit of the total address, it is obvious, that the worldwide Internet can only have a maximum of 64 class A subnets. The two most significant bits in the address are used to select the subnet type.

The **Transmission Control Protocol** uses an additional information, the port number, to reach the destination process inside a host. Portnumbers are chosen in the range from 1 to 2^{16}. Some of them are so called *Known Ports.* They provide standard services, e.g. 21 for the TELNET and 25 for the Mail protocol. The TCP is connection oriented. After a virtual connection is established the protocol takes care of data reliability, order of packets and data losses. The **User Datagram Protocol** is packet oriented. The data is not acknowledged on arrival, and the order of the packets is not conserved during the transmission. The advantage is an increase in performance, caused by the missing overhead of the datachecks and the acknowledgments.

The **IMCP** *Internet Control Message Protocol* is mainly used by the hosts operatingsystem to inform the connected participants of status changes concerning the network connections, like buffer overloads and the change in the average transmission times.

A.2 A Protocols Framework

Whenever a dataexchange has to follow a protocol, additional data, so called pseudodata has to be added to the user supplied informations. This pseudodata contains source and destination addresses, packet lengths and much more. In most cases the additional informations are sent prior to the the real data body. An example is a TCP packet send over the Ethernet. The user data is encapsulated in the following way :



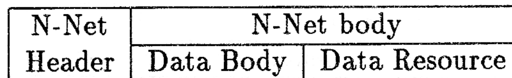
All these encapsulations are of course hidden to the TCP user. He can't even influence the fragmentation. Two packets of 100 bytes, send by one node can arrive as one packet of 200 bytes or 20 packets of 10 bytes. That means, that from the users point of view, TCP builds an unstructured connection stream.

A.3 The N-Net

Following the above introduced tradition of encapsulating data, *N-Net* also uses this method. The main additional features of N-Net are

- the switch to packet oriented behaviour and
- the knowledge about the datastructure, which leads to the appropriate convertion according to the machine dependent data representation.

Like TCP/IP, N-Net internally uses the big endian form of data, which means, that for integers the most significant byte is sent first. Again a data packet is preceded by a header



The *Data Resource* is optional. Currently only a *Data Format Description* can be used as resource. The *N-Net Header* is of fixed length, and consists of 16 Bytes, with the following contents :

Word	Byte 0	Byte 1	Byte 2	Byte 3
0	Subprotocol : Detail		0	Command
1	length of N-Net body in bytes			
2	Subprotocol : Type			N-Data Type
3	length of Data Resource in bytes			

The *Command* describes the content of the body, following the header. Only two values are of interest.

bit	7	6	5	4	3	2	1	0
Send Message	0	0	0	0	0	0	1	0
Send Data	0	0	0	0	0	1	1	1

Where the least significant bit always indiates if data will follow the header or not. In the case of a '0', no data is following, and beside word 0, the header doesn't have the above described form. If the packet consists of homogeneous data of the same type, the N-Data type field has one of the following values :

Data Type	Convertion	Num. of Bytes	Symbol	Value
Integer	Integer	4	Nint	0
Character	Character	1	Nchar	1
Word	Nothing	4	Nncint	2
Byte	Integer	1	Nbyte	3
Short	Integer	2	Nshort	4
Float	Float	4	Nfloat	5
Double	Float	8	Ndouble	6

The header fields, labeled *Subprotocol* make it possible for upper layer protocols to participate the header structure. This avoids another encapsulation for a higher level. Currently there are only two higher level protocols in use :

- the H1 LAN Protocol
- the Remote Server Request Protocol

In the ISO layer model, the N-Net builds the presentation layer.

A.4 The H1 LAN

The *H1 LAN* is totally based on the *N-Net* Protocol, using all Subprotocol fields in the *N-Net* Header part. In addition, it reduces the data transmission to a single packet per connection to decouple the network participants as much as possible. This concept only allows the exchange of messages with a relatively low frequency. The *Subprotocol : Detail* field of each message contains the port number where the answer has to be sent to, if an answer is required. The *Subprotocol : Type* field is split into a *Message ID* and the *H1 Data Type* part. The message ID allows the application to keep track of the requests and answers sent and received. The H1 Data type extends the possible data types mainly to cover the dialects of handling character strings in the different computer languages. The new types are mapped to a valid *N-Net Data Type*, and the actual conversion is still done by *N-Net*. The *H1 LAN* header has the following outfit :

Word	Byte 0	Byte 1	Byte 2	Byte 3
0	Answer Port		0	*Send Data*
1	Length of N-Net body in Bytes			
2	Message ID		H1 Data Type	N-Data Type
3	Length of Data Resource in Bytes			

using the *H1-Data Types* :

Data Type	N Convertion	Num. of Bytes	Symbol	Value
Byte	Nbyte	1	YHraw	1
Integer	Nint	4	YHint	2
Float	Nfloat	4	YHfloat	3
Pascal String	Nchar	1	YHpascalString	4
C String	Nchar	1	YHcString	5
Fortran String	Nchar	1	YHfortranString	6
Short	Nshort	2	YHshort	11
Double	Ndouble	8	YHdouble	12
Byte	Nbyte	1	YHbyte	13
F Pack	Nbyte	1	YHbyte	14

A.5 The Remote Server Request Protocol

To obtain a service from a remote computer, the client has to connect to a so called *Known Port.* The *Known Port* is a port number which permanently provides the same service. The corresponding task, listening on that port number mainly has three ways of reacting on a connection request.

I. The server can answer the requests sequentially. This is only possible for trivial services, not requiring too much CPU power. In addition it is assumed that the operating system queues the incoming connection requests. If it is necessary to keep a connection to such a server alive for a longer time, the task has to handle the communication to several connections simultaneously. This requires an absolute asynchronous behaviour of the Network routines, because a network problem to one of the clients should not influence the performance of the others.

II. Mainly Unix Systems provide the feature of *forking* a server task after a successful connection request. At a special point, which is the *OS fork Routine*, the program is duplicated, and both copies have access to the same IO channels. In our case both are connected to the same client. Only by checking the return value of the fork routine, the different tasks can determine if they are the original server or the daughter process. The mother process closes the connection and waits for the next one, and the daughter continues to talk with the client. The client doesn't notice this change in the identity of its counterpart.

III. Some machines, like the IBM mainframe running MVS, do not support the *Fork* feature. Instead, one has to submit a *Private Server Task.* This scheme is a bit more complicated to implement, because the private server needs the information how to reach the client. I introduced this mechanism the first time with the installation of the *H1 BOS Server*[PAT 90].

Two major checks have to be done by all of the above listed possibilities. The requesting client has to send access informations to ensure data security on the server hosts, because in most cases the *Main Servers* have no privilege restrictions. On the other hand each process has to make sure that no kind of incoming data is able to corrupt its own integrity, and causes unpredictable behaviours.

To make clients independent of the server type, implemented on the different machines, I have chosen an initialization handshake, which allows the clients to find out how to proceed. It uses the *N-Net Header* in the *Send Message* mode. The *Subprotocol : Detail* is set to zero, and the remaining three words are interpreted as integer informations.

Send Message	Information 1	Information 2	Information 3

An initial request to a *N-Net Server* jumps through the following steps :

1. The client connects to the *Known Port* at the server site.

2. The server sends back the *N-Net Message*

Send Message	Server Type	0	0

   where *Server Type* can have the values

   N_Prot_Fork which belongs to Server Type I and II.

   N_Prot_Submit belonging to Type III

3. Depending on the *Server Type* the client behaves in the following way.

   Problem If the client is not willing to accept the specified server it answers :

Send Message	NotAccepted	-1	-1

   N_Prot_Fork No special action on the client side is required. So it only sends back the acknowledge :

Send Message	Accepted	0	0

   N_Prot_Submit The client asks its own Operating System for a currently unused port number. This port number is opened for listening. Together with the host IP address, the port number is sent to the server :

Send Message	Accept	local Host IP	local Listen Port

4. From now on the usual *N-Net* protocol is used for the exchange of the access informations and the parameters. If the default service of the connected task is required, the client only has to send the access informations, in the form of a character string.

Nchar | User Password;PROGRAM=service;... |

If a special program is required, which resides in a different directory, the path has to be make known :

Nint | Dir=1 |

Nchar | Directory String |

Nchar | Access String |

5. The server replies with :

Nint | Request Accepted = 0 |

or

Nint | Request Rejected = -1 |

6. If the server supports the fork protocol, it forks and the already existing TCP connection is used for the following data transfer. In the case of the submit scheme, the server submits the private tasks together with the information about the clients listen port number. The private server tries to connect to the client and the new connection is used. The control connection is closed by both parties.

Now a bidirectional client server connection exists. The above steps are a good example for the *OSI Session layer*, because the protocol determines the direction of the data exchange.

A.6 The Net - Event Request Protocol

One possible service, which belongs to the Farm Event Distribution Concept is the *Event Server*. These servers can deliver events from Disks like SGI or IBM and out of the Main H1 Eventstream, directly from the *Event Builder* or from the *Farm System*. Sometimes a single client needs to access several files on the same disk. To avoid an overload with *Private Servers* a multiplexing of one private server would be desirable, handling all files belonging to the same client. To achive this, I created a kind of virtual channels using a single TCP connection. All requests concerning the channels consist of 4 Integer Words. This block is sent by the client, modified by the server and sent back to the client. First, the client has to request a channel number which it can use in subsequent requests :

Request :	createChannel	0	0	0
Reply :	createChannel	Channel #	0	return Code

If the return Code indicates success, the returned channel number can be used for Net I/O operations. Each operation has to be embedded into a channel select block :

Request :	selectChannel	Channel #	Argument	Function
Reply :	selectChannel	Channel #	Argument	return Code

Function and Argument specify the I/O operations, like open, read, write, close and rewind. Depending on the function, a variable number of packets may follow, and are interpreted as be sent via the currently selected channel. If no longer needed, a channel can be returned with :

Request :	freeChannel	Channel #	0	0
Reply :	freeChannel	Channel #	0	return Code

The whole server can be stopped by :

Request :	stopServer	0	0	0
Reply :	stopServer	0	0	return Code

This protocol part also belongs to the *OSI* **Session** *Layer.* For the file access, 7 actions can be sent via an I/O channel. The open, read, write, rewind, skip and select function. The select functions allows to select several event properties, which is only of use in conjunction with the *Farm Event Server.* All other functions are selfexplaining.

The *OPEN* channel function initiallizes the following handshake.

Client to Server	Nint :	1	1	0	Action	Filename Length	Format Length
Client to Server	Nchar :	Filename					
Client to Server	Nchar :	Format String					
Server to Client	Nint :	Return Code					

The *Action Field* can contain *openRead or openWrite.* The *Format String* is used to send machine specific informations, and the *Return Code* must be zero to indicate success.

The *READ* function causes the server to send the following packets, if the next event is available.

Nint :	# of Words in this Block
	length of Format Block or Zero
	length of Data Block 1
	length of Data Block 2
	⋆⋆⋆
	length of Data Block N
	Zero

For each nonzero length in the above descriptor block, a Data Packet in the Nint format is sent by the server. All these packets together build the full event.

After initiallizing the *WRITE* function, the client has to send the event exactly in the same manner. The server replies with a single integer, which indicates success if it is zero.

CLOSE, REWIND and SKIP are pure, unacknowledged channel functions.

The *SELECT* function has to be acknowledged :

Client to Server	Nchar :	Select String
Server to Client	Nint :	return Code

A.7 The Farm Event Server Protocol

The *Farm Server* is a Server of type I. It serves multiple clients with only one task. The server runs on a Macintosh computer and makes use of Mac specific system calls to access the TCP layer. They work fully asynchronous, which is a fundamental requirement for the simultaneous handling of several TCP connections. Because the farm as event producer can be compared with a single file, more than one I/O channel per connection doesn't make sense. So the server always returns the same channel number for different *createChannel* requests. For the additional service of the farm server, the remote VME memory access, the channel concept is superfluous anyway.

B The Passive Farm OS

B.1 The Task Monitor

The main job of the Farm Task Monitor is to keep track of all Processes currently using farm resources. These tasks can be directly linked to the Farm VME bus System or attached to it via a Network Server. The reasons for an access can be to obtain the recent status informations, to assemble the online farm histograms or to request an event out of the date stream. Whatever is intended, the corresponding process has to make itself known to the system. After that it occupies an entry in the Monitor table and a unique *Task ID* is assigned to the process. From the programming point of view, the connection is achieved with the **mXhereIam** call, which returns a pointer to the **mLproc** structure. This pointer is the reference to the farm in all subsequent calls of the task to it.

> *To get an entry in the Task List, the list has to be searched for an empty slot. Because several tasks accidentally could try this simultaneously, the corresponding code in the Library is called a* Critical Path. *This Path has to be protected against simultaneous access. In real time programming the problem is solved with the help of semaphores[RIP 90]. I implemented binary semaphores, which can hold the values* **I_am_in_Use** *or* **I_am_free** *. In our case, the introduction of semaphores creates two problems.*
>
> - *The* **Deadlock** *problem occures when two tasks try to get a semaphore, which is currently hold by the other task respectively. But both are only willing to free the already attached flag if they were successful in obtaining the other one. It is obvious that such a strategy will lead to a hopeless situation.*
> - *The* **Abort** *problem occures when a task dies, before it was able to return an attached semaphore. The object, which was protected by the corresponding flag is then lost forever.*
>
> *I'm using two types of semaphores, which show a different sensitivity for the above problems. The* Longlife *and the* Shortlife *ones. The latter are only occupied during a very short time interval, mainly to search through lists. They are used indirectly by system calls and they are returned before the system call finishes. Because in the same call, no second semaphore will be requested, they are safe of deadlocks. The extreme short lifetime and the fact, that they are only in use during the time, were system code is active, reduces the probability, that the job aborts while it holds the flag.*
> *For* Longtime Semaphores, *the Abort Problem is solved by the Task Monitor itself. A watchdog, for example could try to find out, if all registered tasks are still alive. If not, the corresponding semaphores and the allocated resources are returned. As long as systemcode is used to request histograms or events, which sometimes makes it necessary to allocate several memory portions, Deadlock is no problem, because whenever a semaphore is not available, the already attached one are freed again.*

In addition to **mXhereIam**, there are only very few calls directly managing the the *Task List.* The **mXwillGo** call which unlinks a connection to the *Farm OS*, the **mXalive** call which has to be called frequently to inform the system, that the task is still alive and the **mXmyID** call which returns the process ID of the calling task.

The structure of the organization tables is shown in Figure 20. A field in the *Communication Vector Table* points to the *Farm OS Entry Header* which contains global informations about the next free process Identifier, the maximum number of allowed processes, a semaphore for the task registration and a pointer to the *Memory Management Area.* This header is followed by an array of structures describing the properties of the registered processes.

Process ID A unique number identifying a process. System processes occupy numbers below 100.

Traceback Pointer back to the first word of *Farm OS Entry Header.* That allows processes to find the OS Entry by only knowing it own process descriptor.

Timestamp The Unix-Time of the process start.

Priority Number between 0 and 15, describing the priority of the process. This assignment is necessary to reserve special memory sections for tasks which can not effort to wait for memory.

Semaphore Flag which indicates, that this process entry is in use.

Alive Counter This field is set, whenever mXalive or any other system routine is called by the task. So, this word only has to be reseted and observed for some time to find out, if the process is still alive.

Name Character string, holding the name of the task.

B.2 The VME Memory Management Part

One entry inside the *Farm OS Header* points to the *VME Memory Management Area* which again consists of a header followed by one descriptor for each memory section. The sections may reside on different VME boards and need not be continues.

The header mainly contains the number of descriptors. Each descriptor holds a pointer to the corresponding memory section, its size, type and name. Two types of memory are supported, the variable and the fixed blocked one. Each section consists of a header and a body. The body is divided into blocks which themselves hold a header and the usable memory area (see Figure 20). The headers contain :

- The **Semaphore**, which declares the block to be in use or not.
- The **Process Identification number**, which protects the tasks of so called *Dangling References.* To visualize this problem, imagine a client, allocating memory and sending the pointer to that portion to a server task. The server possibly needs an unusual long time to handle the request, and copies some data into the remote allocated memory area. Because of the delay, the client could have decided in the meantime, that the server doesn't intend to answer the request. So, it deallocates the memory and continues. This action would be unrecognized by the server, which still copies the data into a memory location possibly occupied by the next task. To avoid this, the server task has to compare the **Process ID** of the remote memory with the ID of the task it is working for.
- The **Traceback** points back to the header of the section. This enables a task to free the memory again, without knowning more than the pointer to the memory block.

The **Fixed Blocked Memory Sections** consist of Blocks, all having the same size. Different sections may use different block sizes. They have the advantage, that they can be allocated without protecting the whole section with a semaphore, and their fragmentation can not increase. This memory type is mainly used as request area.

The **Variable Blocked Memory Sections** adjust their block sizes according to the allocation requests. The disadvantage is, that the whole section has to be locked as long as a task is looking for a free memory portion. The allocation strategy is outlines in figure 21. Each block header doesn't only hold the number of bytes used but also the number of bytes which are usable. The difference is the available free memory. The next allocation request places its memory header and the used memory inside this space and corrects the values in the preceding header. The deallocation process adds its own memory amount to the usable amount of the memory portion before. Figure 21 explains the method in four steps.

1. The *Usable* field of the initial header is set to the total amount of free memory and the *Used* value is set to Zero. At the end of the section, a Stopblock indicates that no blocks are following.

2. An allocation request, not requiring the full free memory, places its header next to the initial header and changes its *Usable* field to the number currently stored in *Used*, which is Zero. The own *Usable* value is calculated to *oldUsable minus sizeof(header)* . The *Used* variable is set to the size of the required portion.(See step II)

3. The following request will do exactly the same with block header I. The difference is, that the own *Usable* field is set to the *Used* amount, and not to zero. The initial header remains unchanged.(Step III)

4. Step IV shows the deallocation of the first block. The *Used* field of the preceding block is not changed, and the *Usable* value is incremented by the number of bytes stored in the *Usable* variable of the block which is going to be deallocated.

> *This method takes care, that successive free memory blocks are merged to avoid a too large fragmentation. Actually there are other possibilities to reduce fragmentation nearly completely. They all make use of so called* **Handles**, *which are pointers to pointers to the memory locations. The advantage is, that the allocated memory block can be moved to a more optimal place, without informing the application, using that memory. But for a multipocessor system this scheme contains a lot of disadvantages. To be sure to own the memory, the* Handle *has to be locked before each access, and unlocked afterwards. The lock prevents the* Garbage Collection Task *from moving the memory around. In addition, a lock needs not always to be successful, because the memory might be on the way to another place, which causes a waste of time. The* Garbage Collection Strategy *has to be well tuned, otherwise the total memory is permanently shifted around. Another point is, that in our case, there are some memory blocks with special purposes, which shouldn't reside in other places. The request area of a Mac, for example always has to be in the local memory of the corresponding VIC to avoid VME bus overloads, due to polling accesses. On the other hand a fair allocation and deallocation strategy of the connected task can also reduce fragmentation without the use of* Handles.

To allocate memory it is necessary to be a registered task, and the own priority must be greater or equal to the priority of a free memory section. The allocation routine allows to specify a Memory Section Name, which may contain wildcards and a hint, if Fixed or Variable memory is prefered.

B.3 The External Requests for the I/O Processors

Using the above VME tools, we are now able to obtain informations out of the I/O processors. The method is uniform to all requests. The structures are shown in picture 22 and the protocol in picture 14. A request area is allocated and filled with the request command, the process ID and request specific data. The pointer of the request area is send to the mailbox of the appropriate I/O processor. The Answer field of the area is observed until it changes from **statusWaiting** to **statusFinished** or **statusRejected**. The possible requests are :

- The **aliveRequest** only tests is the I/O processor is still alive. It consists of one word, to receive the return code.
- The **messageRequest** sends the specified message to the central DAQ console. The message can be contained in the request area itself.
- The **queryRequest** is used to obtain informations about all pending requests of the corresponding I/O processor. The answer will be copied into a separately allocated buffer. The pointer to that buffer and the size are expected in the *Buffer Description Structure*, a part of the request specific part. If the Answer field changes to **statusFinished**, the additional buffer contains an array of the **EXrequest** structure. One field of this structure is the unique request ID, assigned to each request. With the help of this ID a pending request can be taken out of the queue. This

becomes necessary in the case where a task needs to quit, but a request, launched by it has not been answered.

- The **purgeRequest** is used to take requests out of the I/O processor queue. The request specific part has to contain only the request ID. This request is restricted to a system task or to the task which launched the request to be deleted.
- The most important application is the **gimeEventRequest**. It filters the next event with the specified properties out of the data stream and copies it into the buffer area specified by the *Buffer Description Structure*. The Output Processor expects the *Raid Number* and the *Event Mask* in the *Event Selection Structure* of the request specific part. The Raid Number may be zero to indicate **anyRaid**, and in addition to the Event Mask a Mask Validity Word has to be provided, which selects bits in the mask which have to be used or to be ignored. If an event is copied, the *Event Property Field* is filled with the actual event properties.

The different ways of obtaining informations from the farm processors are hidden by a unique programing interface, which chooses the appropriate way to access the data. For tasks using the network, the request is first transformed into the network protocol. The *Farm Event Server*, which produces a mirror process for each network task, translates this protocol into the VME requests. In the following, an example is given to obtain farm events and farm histograms.

```
mLproc          *task :
EventProperties  eventprops ;
char            *bigBuffer ;
long             endTime , histoStack ;
mLHdesc          histos[maxHistos] ;
/*
 *    get enough local memory to hold the event and histograms
 */
  bigBuffer  = malloc( 100000 ) ;
/*
 *     open the connection to the Farm
 */
  priority   = 15 ;      /*  highest priority, will be reduced     */
                         /*  according to the password             */
  strcpy( accessString , " see text " ) ;

  task       = mXhereIam( defaultCVT ,
                          accessString ,
                          priority ,
                          & returnCode   ) ;
/*
 *   test if 'task' is not zero, otherwise check 'returnCode'
 */
/*
 *  now, select the event type you are interested in.
 *     - any RAID board
 *     - maximum event size is 60000 bytes
 *     - only events, which have been rejected by the Filter program
 *     - the event mask should be  '1xx0'
 *                                  |||+-- must be zero
 *                                  ||+--- any value
```

```
 *                                       |+---- any value
 *                                       +----- must be one
 */
    mXselect( task , "event raid=*,size=60000,mask=1xx0" ) ;
/*
 *  check each second, if the event already arrived.
 */
  for( endTime = time() +  10 ; endTime > time() ; ){

      returnCode = mXreadRequest( task , bigBuffer , &eventProps ) ;
      if( returnCode )break ;
      sleep(1) ;
  }
/*
 *   if returnCode is zero or negative, there is no event
 *   otherwise, bigBuffer contains the event, and
 *   eventProps holds the properties :
 *      eventProps.run       run number
 *      eventProps.event     event number
 *      eventProps.length    length of the event in bytes
 *      eventProps.mask      the actual event mask
 */
/*
 *   Now try to get the current histograms of an active raid board.
 */
     histoStack = maxHistos ;
     mXgetHistDescPos( task , histos , &histoStack , 1 ) ;
/*
 * histos contains the pointers to the histogram buffers
 * of all active RAIDs.
 *    i = 0 ... histoStack-1
 *     histos[i].raidNumber  : number of the RAID
 *     histos[i].LOOKdesc    : pointer to start of buffer
 *
 *  with mXreadVME( task , histos[i].LOOKdesc, length, bigBuffer ) ;
 *  the contents of the histograms are copied into local memory.
 */
/*
 * at the end of the session the connection has to be cut.
 */
     mXwillGo( task ) ;
```

This code would run for example on our *Sparc Sun* directly connected to the VME bus, and on one of the SGI machines using Ethernet to access the eventserver which serves as gateway. In the first case the access string can be any name, which is used as processname for the *Farm OS*, and in the latter case, the string should have the form :

```
serverNodeName"username password"::gateway
```

The event request part of the example will also work to get events from the IBM or the SGI. In that case, the keyword *gateway* has to be replaced be the name of the file, which one intends to read.

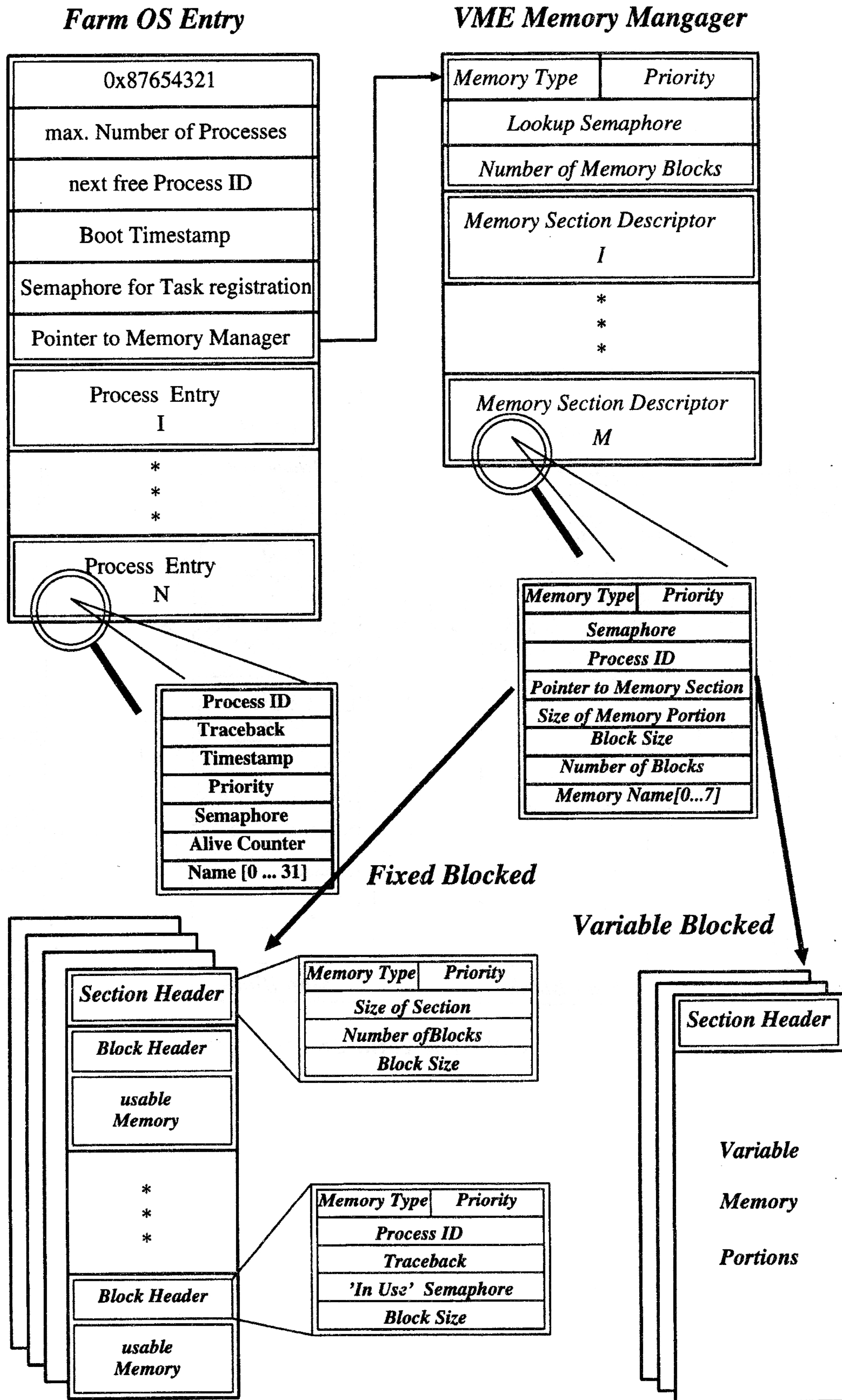


Figure 20: The Farm OS Kernel

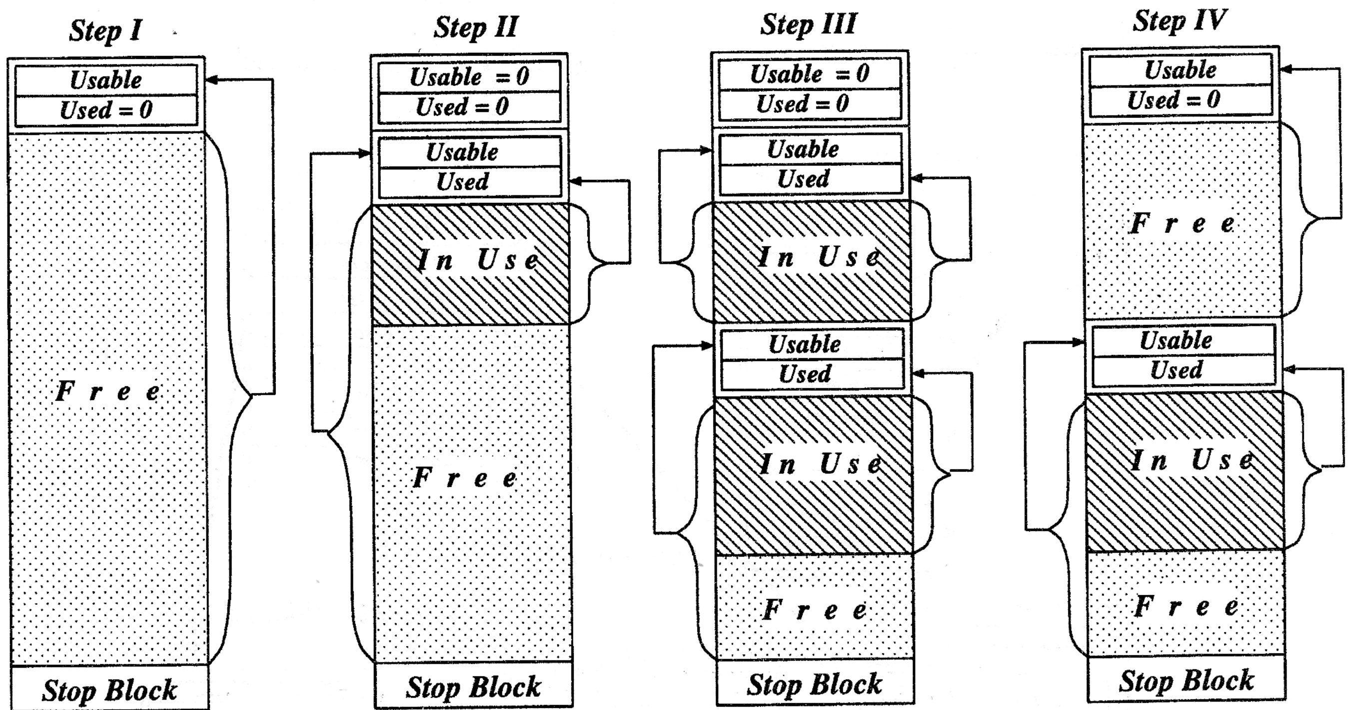


Figure 21: The Farm OS Structures

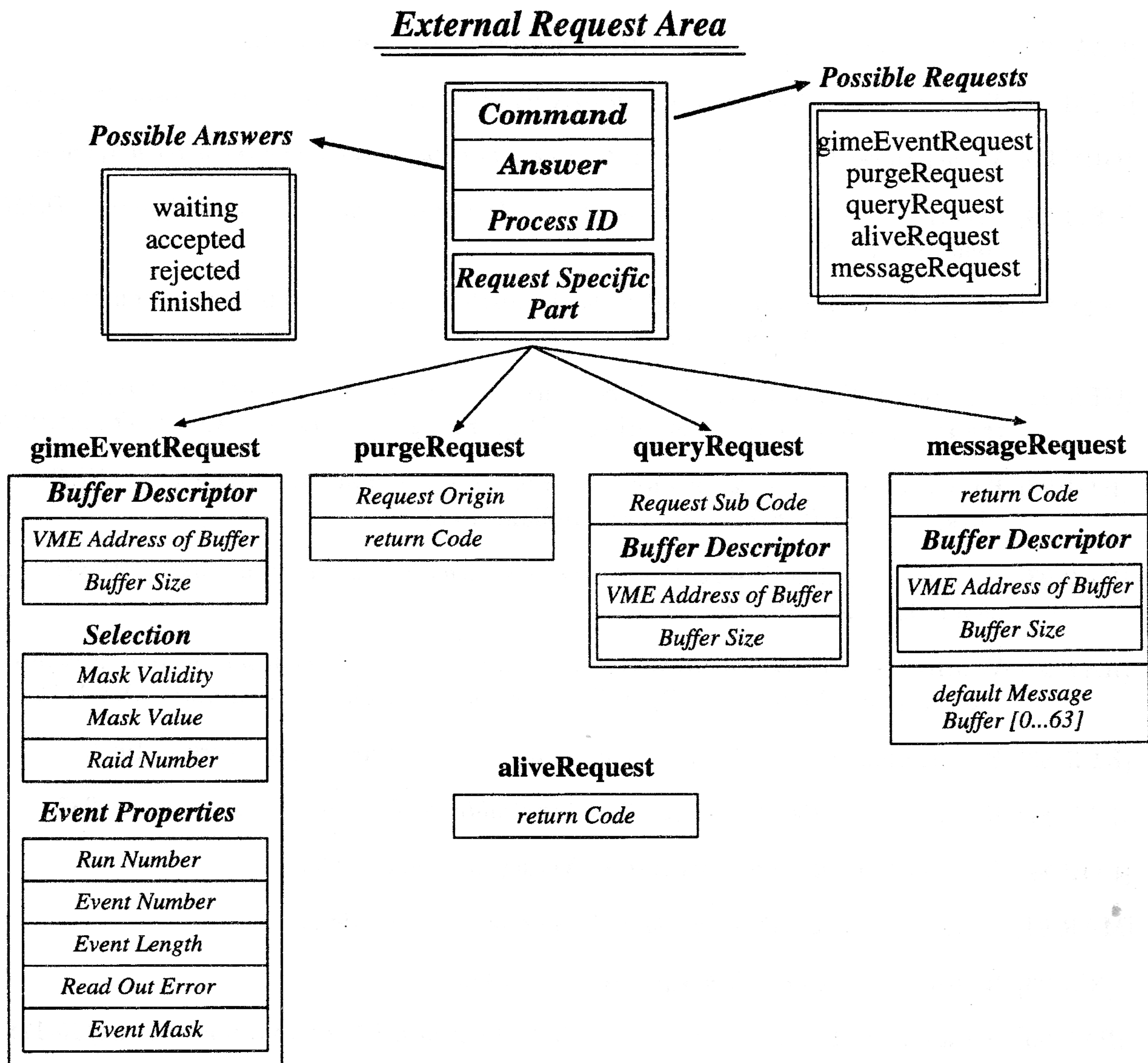


Figure 22: The External Request Union

References

[BER 92] C.Berger,*Teilchenphysik*,Springer-Lehrbuch, 1992.

[BER 93] C.Berger,*Physics at HERA*,private com. 1993.

[BRY 89] J.T.Berry,*The Waite Group's C++ Programing*, Howard W.Sams & Company, 1989.

[CAM 91] Alan J. Campbell,*A RISC multiprocessor event trigger for the data aquisition system of the H1 experiment at HERA*, Conference Record of IEEE,REAL TIME 91.

[CAM 92] Alan J. Campbell,*A Module Steering Program*,private com. 1992

[COM 91] D.E.Comer,D.L.Stevens,*Internetworking withTCP/IP*, Vol. I&II,Addison-Wesley, 1990.

[DEV 92] Michel Devel,*Etude de la production de jets dans le detecteur H1 a HERA*,Thesis.

[DPM 89] Creative Electronics Systems SA,*VME/VSB Dual Ported Memory DPM 8242*, Geneva,CH,1989

[ELS 87] E. Elsen,*The H1 Trigger and Data Acquisition System*, published in Conference Proceeding,Dubna, USSR,May 1987.

[GER 92] R.Gerhards,Z.Szkutnik,*First Experience with Online Reconstruction in H1*, Proceedings of the Int. Conference on Computing in High Energy Physics '92, Annecy,France,1992

[H1-1 93] H1 Collaboration,*Total photoproduction cross section measurement at HERA energies*, Physics LettersB 299(1993),North-Holland,1993

[H1-2 93] H1 Collaboration,*Observation of deep ineleastic scattering at low x*, Physics LettersB 299(1993),North-Holland,1993

[HAY 92] W.J.Haynes,*Bus-Based Architectures in the H1 Data Acquisition System*, presented to the International Conference "Open Bus Systems '92", Zürich,Switzerland,Oct 1992.

[KER 83] B.W.Kernighan,D.M.Richie,*Programmieren in C*, Carl Hanser Verlag, 1983.

[KLE 87] K.Kleinknecht,*Detektoren für Teilchenstrahlung*, Teubner Studienbücher, 1987.

[KOP 88] H.Kopp,*Compilerbau*, Carl Hanser Verlag München, 1988.

[MAR 91] J.Marks,*Level 4 Filtering Algorithms*,private comm. 1991.

[NIS 93] R.Nisius,*Physics at HERA*,private com. 1993.

[PAT 90] A.Campbell, P.Fuhrmann,*Network BOS Routines*, H1 internal note, H1 01/90-128

[PAT 91] E. Deffur, P. Fuhrmann, M. Zimmer,*Communication on the H1LAN*, H1LAN Users Manual Part 1, 11 Feb. 1991.

[PAT 92] P. Fuhrmann,*H1 LAN Interface for Macintosh and others*, H1LAN Users Manual Part 2, 17 June 1992.

[PAT 93] P. Fuhrmann,*N-Net Interface for TCP/IP*, N-Net Users Manual, 4 Nov 1992.

[PAX 92] V. Paxson, C. Saltmarsh,*Glish: A User-Level Software Bus for Loosely-Coupled Distributed Systems*, private comm. 1992

[PRO 93] R.Prosi,*Physics cuts on the L4 Filter Farm*,private com. 1993.

[RIP 90] D.L.Ripps,*An Implementation Guide to Real-Time Programing*, YOURDON PRESS, 1990.

[SAN 90] M.Santifaller,*TCP/IP and NFS in Theorie und Praxis*, Prentice-Hall, 1991.

[SCH 91] H.Albrecht, M. Erdmann, P. Schleper, *H1PHAN, a physics analysis program for H1*,H1 internal note.

[SCH 92] P. Schleper,*Concepts for Leptoquark Search at HERA*,Thesis 1992.

[SFR 85] A.T.Schreiner, G.Friedman,*Compiler bauen mit Unix*, Carl Hanser Verlag München, 1985.

[TAX 90] Micro-Research Oy,*VME taxi User Manual*,Helsinki,SF,July 1990

[TPR 87] H1 Collaboration,*Techical Progress Report*, Oct 4, 1987.

[TUT 92] J.Tutas,*Muonen im H1 Detector*, Phd Thesis,May 1991.

[VME 14] *The VMEbus specification*,IEEE standard 1014.

[VSB 96] *The VME Subsystem Bus(VSB) specification*,IEEE standard 1096.

Lebenslauf

Ich wurde am 18.9.60 als erstes von zwei Kindern des technischen Zeichners Heinrich Fuhrmann und der Verkäuferin Hubertine Fuhrmann, geb. Neuß, in Neuwied geboren.

1966	Einschulung an der Grundschule St. Georg in Irlich
1970	Fünftes Schuljahr an der Hauptschule Irlich
1971	Beginn an der staatl. Jungenrealschule Neuwied.
1977	Wechsel zum David Röngen Gymniasium in Neuwied
1980	Beginn des Physiktudiums an der RWTH Aachen
1983	Vordiplom
1988	Diplomprüfung und Beginn der Promotion beim I. Physikalischen Istitut in Aachen

Danksagung

Herrn Prof. K. Lübelsmeyer gilt mein Dank für die Ermöglichung der Arbeit an seinem Institut und der freundlichen Übernahme des Korreferates.

Ein Quell unerschöpflicher Motivation war die Begeisterung meines Doktorvaters, Prof. Ch. Berger, für die Geheimnisse der Physik. Bei Ihm möchte ich mich für seine rückhaltlose Unterstützung und konstruktive Kritik besonders bedanken.

Bedanken möchte ich mich auch bei den Leuten, die mir durch ihre Arbeit im DESY und in Aachen viel Mühe abgenommen haben, wie Helmut Bergstein, Prof. H.Genzel, Ralf Gerhards, Dirk Handschuh, Torsten Köhler, Frank Raupach, Rolf Steinberg und Engelbert Vogel.

Ein erstaunliches Verhalten zeigten meine Mitarbeiter Rainer Herma, Richard Nisius, Richard Kaschowitz, Peter Schleper und Jörg Tutas. Während der Einnahme von Koffein und Glucose waren sie zu fast jeder Hilfeleistung bereit. Diesem Umstand verdanke ich einen großen Teil meines Wissens über unseren Detektor.

Immer bemüht, die für die Seele so gefährliche Freizeit auf ein gesundes Maß zu reduzieren, waren die Herrschaften von der Online Crew, Erwin Deffur, William Haynes, Paul Hill, Rainer Prosi und Manfred Zimmer. Sie brachten es fertig trotz häufigen Kontakts mit meiner Software nur in Ausnahmefällen die Fassung völlig zu verlieren.

Für die notwendige menschliche Unterstützung in vielen Lebenslagen möchte ich mich bei Alan Campbell, Frederic Descamp, Jörg Marks, Gerhard Söhngen und nicht zu vergessen Plücki besonders bedanken.

Dem DESY Direktorium und den angeschlossenen DESY Abteilungen danke ich für die Bereitstellung der notwendigen Resourcen.

Nicht zuletzt möchte ich meinen Eltern und meiner Schwester für das unerschütterliche Vertrauen danken, das sie in mich gesetzt haben.