GOOSY
Id.: VME-SYS
Version: 1.2
Date: September 6, 1988
Revised: Januar 22, 1990

GSI **O**nline **O**ffline **S**Y**stem**

VME System Description

H.G. Essel, J. Hoffmann, W. Ott, M. Richter, D. Schall, H. Sohlbach, W. Spreng

Januar 22, 1990

GSI, Gesellschaft für Schwerionenforschung mbH
Postfach 11 05 52, Planckstraße 1, D-64220 Darmstadt
Tel. (0 6159) 71–0

List of Figures

Chapter 1

Input/Output Channels (IO)

1.1 Channel Overview

NET-CEMES: VMEbus/DECnet Command/Error-Message Handling, bidirectional
NET-INP: VMEbus/DECnet General purpose output, unidirectional
NET-OUT: VMEbus/DECnet General purpose input, unidirectional

EB-EVOUT: VMEbus/Qbus/BI Full event buffer output, unidirectional
EB-SEIN: VMEbus/VMVbus Subevent input, unidirectional
EB-MES: VMEbus Message handling, bidirectional

FEP-SEIN: VSB Subevent data input, unidirectional
FEP-SEOUT: VMEbus Subevent buffer output, unidirectional
FEP-MES: VMEbus/VSB Message handling, bidirectional

ROC-SEOUT: VSB/CAMAC/Fastbus Subevent data output, unidirectional
ROC-MES: VSB/CAMAC/Fastbus Message handling, bidirectional

STR-EIN: VMEbus Event buffer input, unidirectional
STR-EOUT: VMEbus/Qbus/BI Event buffer output, unidirectional
STR-MES: VMEbus Message handling, bidirectional

1.2 Logical Connections Between the Channels

```
Level 1         Level 2         Level 3         Level 4         Level 5
-------------------------------------------------------------------------------
List mode buffer path with and without STR level

ROC-SEOUT ----> FEP-SEIN
                FEP-SEOUT |--->EB-SEIN
                               EB-EVOUT |----------------------> HOST
                                    |-->|--->  NET-INP
                                    |   |      NET-OUT     |---> DECnet
                                    |   |--->  STR-EIN
                                    |<------|  STR-EOUT    |---> HOST
--------------------------------------------------------------------------------
Test data buffer path for FEP/EB/STR - test

NET-OUT ----->  FEP-SEIN
                FEP-SEOUT |---------------->   NET-INP    |----> DECnet
                          |--->EB-SEIN
                               EB-EVOUT |----------------------> HOST
                                    |-->|--->  NET-INP
                                    |   |      NET-OUT     |---> DECnet
                                    |   |--->  STR-EIN
                                    |<------|  STR-EOUT    |---> HOST
--------------------------------------------------------------------------------
Test data buffer path for EB test

NET-INP   |----------------->  EB-SEIN
                               EB-EVOUT |-------------------->   HOST
                                        |--->  NET-OUT   |--->   DECnet
--------------------------------------------------------------------------------
Message buffer path to FEP / ROC

ROC-MES    <---> FEP-MES    <----------------> NET-CEMES  <--->  DECnet
--------------------------------------------------------------------------------
Message buffer path to EB

                               EB-MES    <---> NET-CEMES  <--->  DECnet
--------------------------------------------------------------------------------
Message buffer path to STR

                               STR-MES   <---> NET-CEMES  <--->  DECnet
```

1.3 Protocol Description

1.3.1 Message Path Protocol

The basic protocol on the VME side, between the NET and the FEPs, the EB and the STR works as follows:

From VAX host to FEP/EB/STR:

Step 1 NET receives an MCB via DECnet

Step 2 NET scans the MCB and determines the destination processor (FEP#1..FEP#13 in VME Crate #1-15, EB, STR)

Step 4 NET writes the message buffer to the destination processor's message input buffer

Step 5 NET writes the process-number of the process to which the message is directed into its FIFO and generates a local interrupt in the destination processor

Step 6 The destination processor receives the FIFO interrupt and scans the FIFO contents to determine the kind of request

Step 7 The destination processor reads the message and executes the desired functions.

Step 8 Depending on the 'synchronous/asynchronous' bits the destitation processor optionally signals 'end of execution' by sending an interrupt (IRQ1) to NET with its processor ID as interrupt status/ID-byte.

Step 9 Depending on the 'synchronous/asynchronous' bits the NET optionally waits for the interrupt from the destination processor.

From FEP/EB/STR to VAX host:

Step 1 The source processor sends an interrupt (IRQ1) with its processor ID as interrupt status/ID byte to NET.

Step 2 NET receives the interrupt and determines the source processor from the interrupt status/ID byte.

Step 3 NET maps onto the source processor's control space and scans the information to determine the kind of request and whether it is a synchronous or an asynchronous transfer.

Step 4 NET transfers the local message buffer of the source processor to its own local message buffer.

Step 5 NET transfers the message buffer to the host via DECnet.

Step 6 NET optionally (depending on synchronous/asynchronous) finishes the message transfer by writing the acknowledge message to the source processor's FIFO and generates a local interrupt in the source processor.

1.3.2 The List Mode Output Channel

The list mode buffer path is logically unidirectional. The physical buffer transfer is always executed by the logically higher level processor. The lower level processor only triggers the higher one. The EB-EVOUT channel is the main output channel of the event builder. There must be at least one ROC and one FEP in the whole VME–system.

The transfer of list mode data from the Front End system into the host CPU is performed in two steps:

1. The transfer of the subevents from the ROC into the FEP memory and the announcement of the subevents to the EB. These are the FEP-SEIN and FEP-SEOUT channels.
2. The transfer of all subevents into the EB's event buffer and the initiation of the transfer to the host CPU. These are the EB-SEIN and EB-EVOUT channels.

In this section only the basic concepts of the transfer protocol are discussed. For details of the internal structure of the EB and FEP list mode task you are referred to the chapter **??** on page **??**.

The FEP–SEIN and FEP–SEOUT channels:

Step 1 ROC#n sends a VSB interrupt to the FEP#n.

Step 2 The FEP locks a new subevent buffer in its buffer queue.

Step 3 The FEP initiates its DMA data transfers and waits for the completion.

Step 4 The FEP pre-scans the data and formats the subevent buffers

Step 5 The FEP announces the new subevent in its map segment of the EB's global memory by writing the subevent status, subevent length and subevent address. For each subevent index in the FEPs subevent queue one control region exists in the FEPs map segment.

Step 6 Continue at Step 1.

The EB–SEIN input channel

Step 1 The EB polls on the status of one subevent index in its FEP map region.

Step 2 If the status of one FEP–subevent changes the EB waits until all FEPs have announced their subevent or depending on the mode, starts with the data transfer of the first FEP which is ready.

Step 3 The EB scans the subevent status and decides whether or not the event processing should be continued.

Step 4.1 The EB initiates the DMA transfer for all subevents with "VALID DATA" status into the final event buffer. If the event buffer is filled the transfer to the host CPU will be initiated.

Step 4.2 When too many FEPs with a wrong subevent status have been detected the EB reports a message to the host CPU.

Step 5 After the completion of all transfers from the FEPs tto the EB the EB unlocks the transferred subevents by switching off the lock–bit of the corresponding subevent buffer in the FEP memories.

The EB–OUT output channel

If an event buffer is full or if the EB–SEIN channels are waiting for any recources this channel will be activated by a subroutine call.

Step 1 Check the transfer status longword. If a transfer is still active return.

Step 2 Unlock the last transferred event buffer by switching off its lock bit.

Step 3 Check if the next buffer is ready for the transfer into the list mode output channel. If not return.

Step 4 Initiate the transfer into the specified output channel and set the "TRANSFER ACTIVE" bit in the transfer status longword. There are three different output channels:

1. The event transfer to NET.
2. The event transfer to a DMA interface.
3. the event transfer to a STR.

Step 4.1.1 The EB sets the "LISTMODE TRANSFER" bit in the remote status longword in its control space.

Step 4.1.2 The EB writes the address of the event buffer to be transferred into the list mode output address field in the EB control space.

Step 4.1.3 The EB triggers NET by generating a VME interrupt with its processor ID as the interrupt status/ID byte.

Step 4.1.4 NET scans the remote status longword to determine the kind of request.

Step 4.1.5 NET transfers the event buffer in its own memory.

Step 4.1.6 NET puts the transfer status word into the EB–FIFO and the FIFO–ISP copies the status into the transfer status longword in the EB–control space. The transfer status is 0 if the transfer was successfull or negative if any error occured.

Step 4.2.1 Not yet specified.

Step 4.3.1 Not yet specified.

1.3.3 Test Data Buffer Path for FEPs, STR and EB

For testing purposes the list mode data input channels of the FEPs can be switched by a command not to wait for a VSB interrupt but for a local FIFO interrupt indicating the arrival of a test data buffer. For testing the EB and STR the trigger is the same as for the real list mode data buffers, only the NET has to be aware that the buffer in question has to be sent to the EB or STR. The buffer transfer to the FEP is similiar to the message path protocol but a seperate channel of the NET is used. For the output of the pre-scanned data there are two possibilities:

- the list mode data path to the EB,
- the network output channel NET-OUT.

For the latter the protocol is similiar to the message protocol:

Step 1 NET receives a message buffer containing the command code to switch the input mode of the FEPs/EB/STR to 'TEST DATA BUFFER INPUT'.

Step 2 NET writes the command to the destination processor's FIFO.

Step 3 The FEP/EB receives an interrupt set by NET and determines the interrupt reason from its FIFO data.

Step 4 The FEP/EB ackowledges the command.

Step 5 NET writes the first test data buffer to the destination processor's data input buffer and triggers it by a FIFO interrupt.

Step 6 The destination processor determines the desired output:

6.1 destination is NET-OUT: send interrupt (IRQ1) with the processor ID as interrupt status/ID-byte to NET.

6.2 destination is EB: protocol outlined in chapter **??** on page **??**.

Step 7 NET receives the interrupt status/ID - byte and determines the processor from its processor mapping table.

Step 8 NET maps onto the processor's control space and reads the control information to determine the kind of request.

Step 9 NET transfers the data buffer from the processor to its message buffer and initiates the network transfer to the VAX host.

Chapter 2

Processes/Tasks (PROC)

2.1 NET Tasks

2.1.1 Task Synchronization and Communication

The NET processor will have three tasks which are synchronized by RSX-11S event flags and communicate internally using an RSX-11S shared region. The FEPs, EB and STR trigger NET by VME interrupts (IRQ1). When NET transfers a message to one of the other processors, it will write the message type/subtype into the destination processors FIFO. On the other side the FEPs, EB and STR write their processor number as interrupt status/ID byte when interrupting NET. The local addresses in the destination processor can be calculated from the unique processor number according to Appendix A.

2.1.2 CMD/ERR Task (CEMES)

- one dedicated DECnet channel for receiving commands and sending error messages
- communicates with the VME I/O task and internal control status region in the shared region

2.1.3 NET I/O Task

- handles three DECnet channels which are serialized internally to one channel connected to the VME-I/O task
- limited access to internal control status region
- message transfer in both directions (VME - DECnet)

2.1.4 VME I/O Task

- can be triggered from CMD / ERR, NET-I/O and VMEbus
- transfers message buffers from/to VMEbus to/from message area in a shared region
- handles VME interrupts

2.1.5 Event Flag Usage

(To be done.)

2.2 FEP Processes

The tasks handled by the FEP are executed in separated processes:

1. The Subevent Readout Process is responsible for the readout of the Front End system and the organisation of the subevent output channel.
2. The Command and Message handler receives the commands and messages from the Host (VAX), which are dispatched by the NET processor. The commands are executed and the completion status or the command execution results are announced to the NET processor.
3. The Local Control Process allows to get detail informations about the status of all processes.
4. The Exception Handler will be activated if an exception occurs. It sends a message to the NET processor to inform the Host (VAX) about the reason of the exception.
5. The User Process is an arbitrary user written procedure which has access to the subevents via a mailbox mechanism. It can be loaded and activated at any time.

The system and process configuration parameters are kept in the Global Control and Status Region (CSR) which is accessible by all internal and external processes. The CSR is used like a global database, one process changes a system parameter which influences the functionality of an other process. Generally all system relevant configuration parameters are set only by **one** process; by the Command and Message handler. All other processes only have read access to these parameters!

2.2.1 Subevent Readout Process

Process Functionality Description

The listmode process has to handle the following tasks:

1. The readout of the selected input channel.

2. The execution of an optional user written subevent scanner or subevent analysis.
3. The initiation of the transfer of the subevent buffer to the selected output channel.

The listmode process has one input and one output channel for the listmode data, futhermore one mailbox like output is implemented which allows the transfer of subevents to a user written process.

The FEP can handle the following listmode input channels, which can be activated dynamically by GOOSY commands:

- The CAMAC Readout Controller (CAV 3400).
- The CAMAC Readout Processor (CVI 1000).
- The Aleph Event Builder.
- The Heidelberg Flash ADC System (not yet specified).
- The TH Darmstadt Transputer System (not yet specified).
- A Test input channel

The input channels are implemented by subroutine calls. To prevent a system overhead due to a scan necessary to find the actual active channel, the subroutine is called via an external function pointer, which is set by the Command and Message Handler task.

The subevent readout process polls on the "LISTMODE DATA RECEIVE" flag **b_fic_lm_receive** in the CSR (see page **??**). If the flag is set a subevent request is sent to the buffer manager, which returns the index of the next free buffer. Then the input handler subroutine is called with the pointer to the available subevent buffer data area. It has to return the size of the total subevent data. After a the completion of the subevent readout an optional user subevent pre-processing routine is called. If it has been finished successfully the status, length and address of the subevent is written into the buffer manager slots for that subevent index. The subevent status is used to signal the output channel how to process the subevent or the whole event. The following status are implemented (see include file $STATUS):

FIC__VALID_DATA The subevent contains valid data only.

FIC__ERROR_DATA An error occured during the subevent processing . The data in the subevent are invalid.

FIC__SKIP_SUBEVENT Due to a software or hardware trigger descision the subevent can be ignored.

FIC__SKIP_EVENT Due to a software or hardware trigger descission the whole event (all subevents) can be skipped.

The prepared subevent can be sent to the following output channels:

- to the Event Builder.
- to the NET processor.
- to a test output channel.
- to a local subevent debugger.

The output channels are implement like the input channels. They are activated by a subroutine call via an external function pointer.

In the following sections the FEP system components are described in detail.

The Subevent Buffer Manager

Each FEP has up to 32 subevent buffers supervised by a buffer manager (see figure 2.1). For each available buffer a 'BUFFER LOCK' flag exists in the buffer manager. If a new buffer should be locked the buffer manager checks if the **next** buffer in the queue is free. If this is the case the lock flag for this buffer will be set. The buffer manager returns the index of the locked buffer. Otherwise it returns an error. If no buffer is available the readout process polls on the buffer lock flag until the next buffer gets unlocked. After that the input channel handler is called.

Input from the CAMAC Readout Controller CAV 3400

In the standard version the CAMAC Readout Controller (ROC) is processed by a CAMAC list executer. The list is generated at the VAX and downloaded by the NET Processor. If the data in all CAMAC crates are ready for the readout a VSB–interrupt is generated. The VSB–interrupt service procedure sets the "LISTMODE DATA RECEIVE" flag in the CSR (see page **??**). The ROC-readout handler executes the CAMAC CNAF list located at the address kept in the CSR slot l_fic_readout_list (see page **??**).

Input from the CAMAC Readout Processor CVI 1000

If the CAMAC Readout Processor (ROP) has finished the processing of its internal CAMAC list or of the user program it generates a VSB–interrupt. The VSB–interrupt service procedure in the FEP sets the "LISTMODE DATA RECEIVE" flag in the CSR (see page **??**). The ROP-readout handler initiates a DMA transfer from the ROP memory via VSB into the subevent buffer. This will be done for each CAMAC subcrate. The information about the subcrates which should be included in the readout is kept in the CSR slot b_fep_crates_readout. In contrary to the simple CAV 3400 controller, where the readout is determined by a readoutlist, single crates can be included into the subeventreadout by switching on the corresponding bit in the b_fep_crates_readout slot.

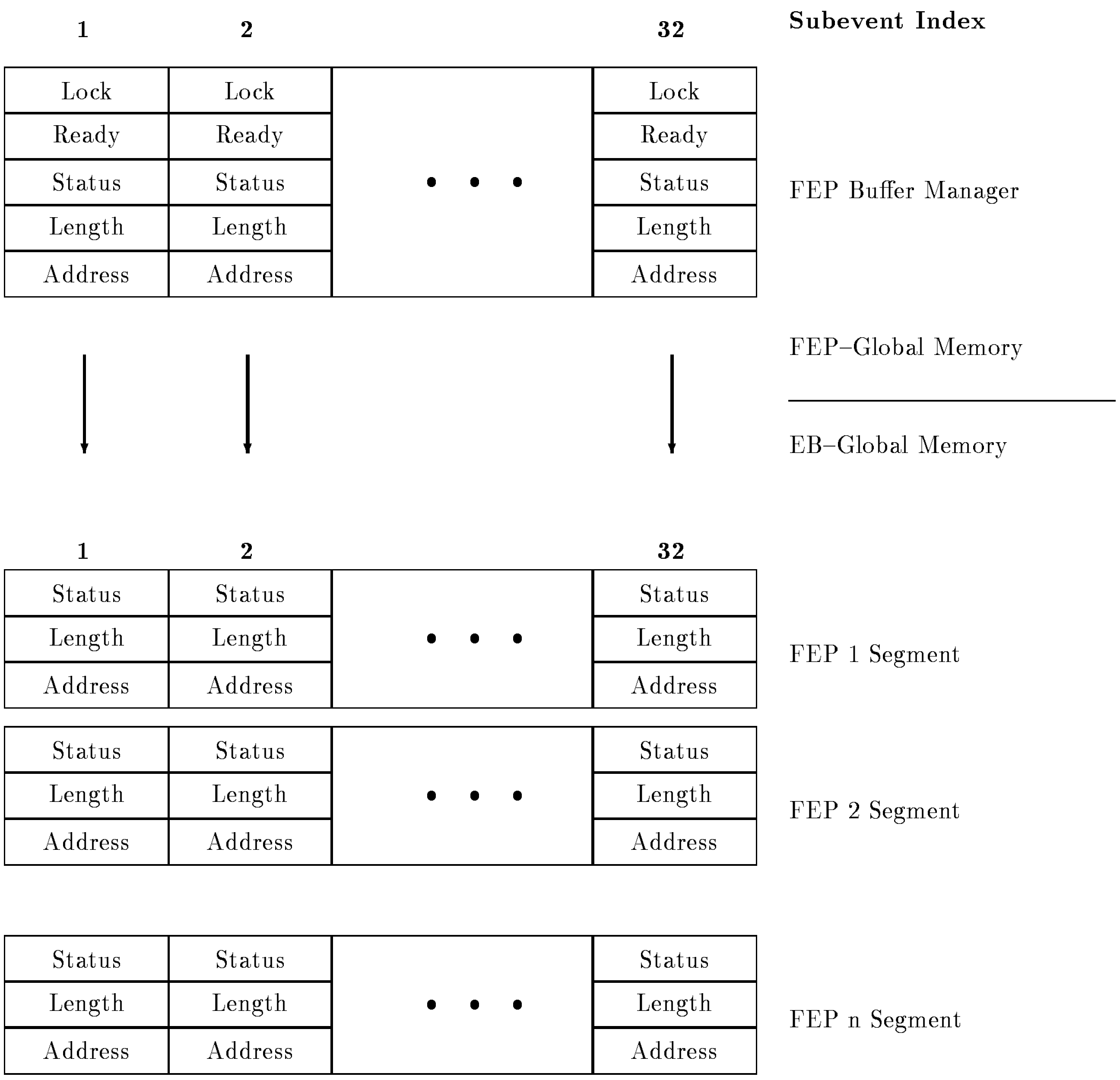

Figure 2.1: Announcement of Subevents in the FEP–map segment

Input from Aleph Event Builder

If the Aleph Event Builder (AEB) has finished the processing of its internal FASTBUS list or of the user program it generates a VSB–interrupt. The VSB–interrupt service procedure in the FEP sets the "LISTMODE DATA RECEIVE" flag in the CSR (see page **??**). The AEB-readout handler initiates a DMA transfer from the AEB memory via VSB into the subevent buffer. This will be done for each FASTBUS subcrate. The information about the subcrates which should be included in the readout is kept in the CSR.

Input from the Test Input Channel

This channel can be used for debugging and checking the correctness of the subevent transfer from the FEPs to the Host (VAX). The Command and Message task receives a buffer with test data from the NET Processor and sets the "LISTMODE DATA RECEIVE" flag in the CSR (see page **??**). The data are copied by DMA from the test input buffer into the subevent buffer.

Output to the Event Builder

The subevent data transfer into the Event Builder memory is not performed by the FEPs. The FEPs only announces to the EB that the subevent with the currently used index is ready for the transfer.

For the announcement of the subevents to the EB all FEPs have their own "database" in the EB memory, the so called FEP–map segment. For each FEP subevent buffer an entry in the map segment exists. The subevent index returned by the buffer manager and the entry index of the corresponding subevent buffer in the map segment are identical. (see figure 2.1 on page 15).
After a subevent is ready for transfer the FEP copies the contents of the buffer manager control block (status, length and address of the subevent) into the corresponding slot of the FEP–map segment. If the subevent status in this slot changes from 0 to any value the subevent is recognized by the EB (see page 20).

After the announcement of all FEP subevents with the same index the EB can start the transfer. If the subevent has been copied into the EB memory the EB switches off the "BUFFER LOCK" flag in the FEP buffer manager and resets the subevent status in the EB. Now the subevent buffer is available for the FEP again.

The subevent readout of all FEPs in the system is synchronized by a hardware trigger. To guarantee that subevents belonging to one event are packed together by the EB it has to be ensure that for each hardware trigger all FEPs fetch subevents with the same index from their buffer queue. That means the buffer queue length in all FEPs has to be the same! Futhermore the FEP has to announce its subevent to the EB in any case, independent on the quality of the subevent data and in any error case!

Output to the NET Processor

(To be done.)

Output into the Test Output Channel

(To be done.)

Output into the Subevent Debugger

The subevent debugger is an additional tool which allows to control the subevent structure and optionally the subevent data in the currently processed subevent. It can be activated and deactivated at any time. The subevent debugger is called before the output channel handler. The subevent debugging disturbes the timing of the whole system, therefore it is recommended to disable the timeout counter in all FEPs and in the EB to prevent stopping of the whole system due to any timeout condition! The timeout counter is set to infinite if the l_fic_timeout slot in the CSR is 0 (see page **??**).

2.2.2 Command and Message Handler

All FEPs and the EB can receive and process only one command at the same time. Therefore the Command and Message transfer from the NET processor to the FEP is synchronous. The Command and Message Handler is triggered by the NET processor due to a pSOS event flag. If this flag is received the command or message buffer is expected at the address specified in CRS slot l_fic_receive_msg. The command will be executed or dispatched to an other process. In any case the process which executes the command has to send an acknowledge with the execution status to the NET processor. If results have to be sent back, this has to be done via the Message path (see page 18) and by signaling that fact to the NET processor using the command execution status. If results from the command excecution should be send back, this is signaled to the NET processor by the FIC__CMD_RESULT status and the buffer with the results is written to the address saved into the l_fic_send_msg slot in the CSR.

2.2.3 User Defined Process

The inclusion of user subroutines or a user process in the VME–system is described on page 21. This section describes how a user process can receive data from the system.

1. Each user process can receive commands. Each special user command or message is signaled by the Command Handler by sending a pSOS Event Flag to the User Process, if it is loaded and activated! After the Command processing an acknowledge is expected by the NET Processor. Data can be sent back to the Host (VAX) via the Message path (see page 18). To receive a command the user has to call the system routine FIC_RECCMD, to send back the acknowledge FIC_CMDACK.

2. The user process has access to the subevent by a mail box mechanism to perform a user analysis. Subevents are signaled by the Subevent Readout Process using a pSOS Event Flag. After the subevent processing the user process has to inform the Subevent Readout Task that the next subevent is ready to be sent. To receive a command the user has to call the system routine FIC_RECSUB and to send back the acknowledge the user has to call FIC_SUBACK.
3. The User Process has a limited access to the Front End System, which should be handled by a subroutine to control if privileged action should be performed.

2.2.4 The Exception Handler

The Exception Handler is a process running at the highest priority in the system. It is started if one of the following exceptions occur:

1. Zero Divide check.
2. Address Error.
3. Bus Error.
4. Illegal Instruction check.

The Exception Handler sends a copy of the exception stack as an error message to the Host (VAX). In any case exceptions are unrecoverable errors, therefore the whole processor is set into the HALT state if any of these errors occur!

2.2.5 The Message Sender

At the same time FEPs and EB can send only one Message to the NET processor. Because of the fact that all FEP processes are allowed to send messages to the NET processor, the message synchronization is done internally by the pSOS Exchange facility. Each process requests a message buffer by the call of FIC_GTMSG which performs the locking and returns the pointer to the free message. After filling the buffer it has to be sent using FIC_SDMSG to the NET processor. The message buffer gets unlocked if the NET processor writes any value into FEP FIFO register 1. The message sender can be used by each process to send error messages to the host or for sending any data packages.

2.3 EB Processes

The tasks handled by the EB are executed in separated processes:

1. The Subevent Collector collects the subevents from the input channel and prepares them for the output channel handler.

2. The Command and Message handler receives the commands and messages from the Host (VAX), which are dispatched by the NET processor. The commands are executed and the completion status or the command execution results are announced to the NET processor.
3. The Local Control Process gets detailed informations about the status of all processes.
4. The Exception Handler will be activated if an exception occurs. It sends a message to the NET processor to inform the Host (VAX) about the reason of the exception.
5. The User Process is an arbitrary user written procedure which has access to the whole event via a mailbox mechanism. It can be loaded and activated at any time.

Beside the Subevent Collector all other Processes have the same functionality and are implement in the same way as described for the FEP. Furthermore the EB consists of the same system components as the FEP: The Buffer Manager and the Message sender are identical. Therefore only the Subevent Collector will be described in detail.

2.3.1 Subevent Collector

Process Functionality Description

The Subevent Collector has to manage the following tasks:

1. The collection of subevents from the input channel. The status of all subevents are scanned to decide how to process the whole event.
2. The execution of an optional user written event scanner or event analysis.
3. The packing of the FEP subevents into the final output buffer.
4. To initiate the transfer of the final event buffer into the selected output channel.

The Subevent Collector has one input and one output channel, futhermore one mailbox like the output is implemented which allows the transfer of events to a user written process.

The EB handles the following input channels:

- Synchronous Subevent input from the FEPs.
- Asynchronous Subevent input from the FEPs.
- Subevent input from a test input channel.

The input channels are implemented by subroutines, which are called via an external function pointer. The finally prepared subevents are sent into one of the following output channels:

- To the NET Processor.

- To the Host (VAX) via the CES-HRV 8217 DMA interface.
- To a Software Trigger System.
- To a Test Output Channel.
- To an Event Debugger.

The output channels are implement like the input channels, they are activated by a subroutine call via an external function pointer.

Subevent Input from the FEPs

Each FEP has its own "database" in the EB memory. These FEP–map segments are used by the FEPs to announce that the subevents are ready for the transfer to the EB memory. For each subevent out of the FEP subevent buffer queue one entry exists in the map segment which contains the status, length and address of the subevent corresponding to the entry index. In general the subevent index in the buffer manager control structure and the entry index have to be indentical. It is assumed that each subevent in the FEP–map segments having the same entry index belong to the same physical event!

Therefore all FEPs receiving a hardware trigger have to map subevents with the same index from their buffer queue and it has to be ensured that for each hardware trigger one subevent is announced to the EB.

The EB scans the entries in the map segments one after each other. The EB notices the FEP subevents if the subevent status changes from 0 to any value. If all FEPs have announced their subevent the EB decides how to continue depending on the subevent status:

- If all subevents have **FIC_VALID_DATA** status the subevents are copied from FEP memory into EB memory.
- If any **FIC_SKIP_SUBEVENT** status is found the corresponding subevents will be ignored.
- If a **FIC_SKIP_EVENT** status occurs the whole event is ignored, no subevent transfer from FEP to EB is initiated.
- If **FIC_ERROR_DATA** status is found in one or several events the EB starts an error analysis and sends a message to the Host (VAX). One mode is available to ignore subevents with invalid data. For such subevents a subevent header indicating the invalid data is written to the output event buffer.

If not all FEPs announce their subevent the timeout condition rises and the EB performs an error analysis to determine the reason for this error.

Additionally the Input Channel handlers of the FEPs have to copy the subevents into an intermediate buffer if an user routine is enabled which performs an event analysis or if an output channel is active which requires a total event. In any case the Input Handler generates a "copy–list" of subevents to be transfer. This list can be processed by the output channel handler.

If the subevents are copied to the intermediate buffer the EB clears the subevent status in the FEP-map segment, maps to the buffer manager structure of each FEP and unlocks the subevent buffer with the currently processed entry index. Now the subevent is available again. If the subevents are still in the FEP memory it is not allowed to unlock them.

Futhermore an asynchronous FEP input mode scan is implemented. In that mode the Subevent Collector does not wait until all FEPs have announced their subevent. But it includes all subevents in the "copy–list" which are ready for the transfer and which are found in all possible entry slots. The subevent readout is no longer synchronized. And subevents packed into the final event buffer do not necessarily belong to the same physical event! This mode can be used with a disabled hardware trigger to get a higher data rate for testing the single system components.

Event Output to the NET Processor

(To be done.)

Event Output with CES Parallel Interface

(To be done.)

Event Output into the Event Debugger

The subevent debugger is an additional tool which controls the Event structure and optionally the event data in the currently processed event buffer. It can be activated and deactivated at any time. The event debugger is called before the output channel handler. The event debugging disturbes the timing of the whole system, therefore it is recommended to disable the timeout counter in all FEPs and in the EB to prevent the stopping of the whole system due to any timeout condition! The timeout counter is set to infinite if the l_fic_timeout slot in the CSR is 0 (see page **??**).

2.4 How to Include User Routines into the System

It is possible to include the following user routines into the system:

- A Front End Readout Routine.
- A subevent scanner or subevent analysis routine.
- An event scanner or event analysis routine.
- An user process for analysis or control tasks.

All user routines are subroutines with a well defined parameter lists and with fixed subroutine names. They are linked together by the VAX–command CLIUSER and are downloaded by the "LOAD VME PROGRAM" GOOSY command.

The user subroutines are linked together with a system initilization routine, called when the routines are enabled. The initilization routine, placed at a fixed memory location, returns the addresses of the user routine entries which are stored in external function pointer. If the user routine does not exist a "NULL" pointer is returned.

The system initilization routine shows how that is done:

```
user_init(p_readout,p_anal,p_process)
long *p_readout;
long *p_anal;
long *p_process;
{
#define RTS 0x4e75          /* opcode for rts instruction    */
long l_statement;
extern long user_readout;   /* user readout subroutine  name */
extern long user_anal;      /* user analysis subroutine name */
extern long user_process;   /* user process subroutine  name */
/**/
p_readout = &user_readout;
l_statement = *p_readout;   /* first statement in routine    */
l_statement = l_statement >> 16;    /* get high order word   */
if (l_statement == RTS) p_readout = 0;
/**/
p_anal = &user_anal;
l_statement = *p_anal;      /* first statement in routine    */
l_statement = l_statement >> 16;    /* get high order word   */
if (l_statement == RTS) p_anal= 0;
/**/
p_process = &user_process;
l_statement = *p_process;   /* first statement in routine    */
l_statement = l_statement >> 16;    /* get high order word   */
if (l_statement == RTS) p_process = 0;
```

```
    }
```

For each non existing user routine the system links a system subroutine which consists of a "RTS" statement only. Therefore, if the first statement in the subroutine is "RTS" no user routine exists and a "NULL" pointer is returned.

Chapter 3

VME–System Initialization (STARTUP)

3.1 The system start–up

The total VME–system consists out of up to 4 different processors: the NET processor, the FEP and EB processors, the CAV 1000 controller and the ALEPH event builder. Each of these processors has an other hardware configuration and a different operating system. Therefore the startup of each system components look somewhat different:

1. The **NET** processor is the EUROCOM 5 VME–module with an ETHERNET interface. After the power-up a user defined startup procedure is activated which starts the NET processor tasks fetched from a disk server running at the VAX host.
2. The **FEP** and **EB** processor copies the operating system pSOS from an EPROM into the local memory. After that a user written ROOT process is activated. This ROOT process polls at the b_fic_load flags in the CSP space (see page **??**). This slot is set to 1 if the VME–system tasks have been downloaded from the VAX.
3. Similar to the FEP or EB the **CVI 1000** controller pools on a load bit. The controller software is loaded from the VAX via the NET and FEP processors.
4. The **Aleph Event Builder** gets it's system software from a disk server running at the VAX Host. The system is loaded during the power-up sequence.

3.2 VME–System initialization

After the power-up the NET processor and the Aleph Event Builder are ready, but the FEP-, EB-processor and CVI controller are waiting for the system download. This download is initiated by a GOOSY–command. Before this is possible the NET processor has to guarantee that all FEP's in the system can be reached. For that purpose VMV and the all FEP's in the system have to be initialized:

1. The NET processor gets a list from the host containing all FEP's expected in the VME–system.
2. The VMV–modules are initialized to work in the "window mode".
3. The **CTLVME** 1 register of the EB and all FEP's in the system are initialized by the NET processor containing their VMV–card cage number in the lowest four bits. This defines the VME–crate offset of each processor in the whole system and should be done with the A24/D32 address modifier. The CTLVME1 register is located at the VME–address 0*x*00000008. Now each FEP in the system can be accessed in an unique address range on the VME–bus:

VME–Address mapping

31 28	24	20	16 12 8 4 0	Offset
—	Crate	Proc	VME–Address	0

4. The NET processor initializes the VMV–modules to work in the "autorouting mode".
5. The system tasks for the EB and FEP's have to been downloaded by a VAX–command.
6. The NET processor copies the system software into the EB or FEP memory.
7. After the load completion the NET processor puts the start address of the system software into the l_fic_root_start slot of the CSR and sets the b_fic_load flags to 1.
8. The FEP or EB jumps to the start address of the system task and initializes all hardware components and the CSR–space. Futhermore the VME–master interface will be initialized to generate A32/D32 address modifiers.

3.3 EB and FEP configuration

After the system start of the EB and FEP the following hardware modules will be initialized:

1. The interrupt controller, to enable the local interrupt levels generated by the DMAC, FIFO and the system timer. Futhermore the expected VME- and VSB–interrupts are enabled.
2. The FIFO's are initialized:

   FIFO 0 This FIFO interrupt send a specified pSOS event flag to a specified process. The event flag number is expected in the lower 16–bit word of the FIFO register and the process index in the upper 16–bit word.

FIFO 1 Is used by the Message Sender. The FIFO contents is ignored. The interrupt service procedure puts a message into the message exchange.

FIFO 2 Is used to signal a command to the Command and Message handler.

FIFO 3 Not yet used.

3. The Zilog Z8036 i/o and Timer is initialized to generate the system clock ticks in a 10msec sequence. Futhermore the FIFO interrupts are generated by this hardware component.
4. The pSOS system timer is initialized and the timer interrupt service procedure is enabled.
5. The DMAC is initialized. The different DMAC channels are preset as the following:

   Channel 1 is used by the subevent collector of the EB. it generates no interrupts.

   Channel 2 not yet used.

   Channel 3 not yet used.

   Channel 4 not yet used.

After this hardware initialization phase the system CSR space (see page **??**), as well as the other system structures are initialized and the subroutines for the standard input and output channels are enabled. Then the system processes are created. The readout process in the FEP's will be suspended after the creation, because it has to wait until the readout list, respectively the CAMAC Readout processor CVI 1000 has been initialized. If this has been done, the FEP's are ready to perform the readout of their front–end system.

Chapter 4

Interrupts (INTER)

4.1 VMVbus Demands / VMEbus Interrupts

- IRQ1 in all VME crates
- crate #1 VMV demands 2-15 enabled
- crates 2-15 all VMV demands disabled for trigger by NET

4.2 Local Crate VME Interrupts

- IRQ3 in crate 1 for DMA completion of VME/Q-bus interface

4.3 Local Interrupts on the FIC8230

- Timer interrupt, all processors (LIRQ3). The counter/Timer and Parallel I/O–unit (ZILOG Z8036) is initialized as a timer. It produces clock ticks with a 10 ms period. Each clock tick is signaled by an interrupt at level LIRQ 3 connected to interrupt vector **65**. The timer ISP controls if the interrupt is valid and announces a clock tick to pSOS.
- DMA completion, all processors (LIRQ2). After the completion of a DMA an interrupt at level LIRQ 2 is generated. Each channel produces another interrupt vector if it has been finished with success or with an error. The ISP can signal an event flag to the process which generated the DMA.
- VSB interrupt for FEP's (LIRQ4) generated by ROCs.
- Local FIFO status flags (LIRQ3). If a FIFO is accessed an interrupt at LIRQ 3 level is produced and an interrupt vector **64** will be generated by the ZILOG Z 8036 parallel I/O–

unit. Each FIFO has two bits which are used as inputs for the Z 8036. Therefore the ISP can analyse this input register and decides which FIFO has been triggered the interrupt.

Chapter 5

Trigger and Synchronization (TRIG)

5.1 Trigger and Synchronization

5.1.1 Trigger Tasks

Introduction

The Trigger Modules connected via the Trigger Bus have to guarantee together with the FEPs and the ROCs a correct inhibition of ADC gates and TDC starts during conversion and read-out time as well as the collection of corresponding subevents to the complete event by the EB. The Trigger Modules are either integrated into the EBI, AEB-VSB-Interface in FASTBUS, or into a CAMAC or VME single module.

There is just one dedicated Trigger Module in the whole system, called Master Trigger Module. Only this module accepts external triggers and performs Fast Clear (FC). It controlls also the GO line of the Trigger Bus (TBUS) thus starting or stopping the data taking in the whole system.

There are four Master Trigger ECL-inputs available on the front panel to select a specific trigger type. Using these four coded trigger inputs 15 different trigger types (1 - 15) can be distinguished.

Event Synchronization

The first of the four Master Triggers starts a trigger acceptance time window (GTIME) in the Master Trigger Module. To be accepted other triggers must arrive during this time. The GTIME is adjustable within 50 - 500 ns limits by means of on-board switch H2. The duration of the trigger inputs must be longer than 30 ns. The four inputs set four flip-flops to keep this trigger information until the Total Dead Time (TDT) on the Trigger Bus is cleared. The flip-flop outputs of the Master Trigger Module are sent via four Trigger Bus lines to all Trigger Modules in the system from where they are readable through each Status Register. These four bits might be used by the ROCs to select an appropriate read-out procedure. The front panel ECL-signal Master Trigger (MT) with a duration of 50 - 500 ns (adjustable by on-board switch H1) is produced starting with the first Master Trigger input.

At the end of the GTIME that is calculated separetly in each Trigger Module the local Dead Time FF is set in each unit. The Total Dead Time (TDT) line of the Trigger Bus represents the wired OR of the local Dead Time FF of each Trigger Module. It is available as a front panel ECL-signal and as a bit in the Status Register of each Trigger Module.

With the first of the four Master Trigger signals a preprogrammable conversion time window (CTIME) (in 100 ns steps up to 6553.6 μs) is started in each trigger module individually. The length of the CTIME can be set by program individually in each Trigger Module.

At the end of its CTIME each Trigger Module sets the Start-of-Read-out (SOR) and interrupt signals to its ROC (only if no Fast Clear had stopped the CTIME). The interrupt is set in those Trigger Modules that have interrupt enable bit set in their Control Registers. In case of Fastbus an interrupt is sent via the EBI module to the AEB. In case of CAMAC this is done by asserting a LAM, in case of VME a VME interrupt is initiated. When working with CBV CAMAC module the LAM produces a VSB interrupt (if enabled) in the FEP. In addition, the interrupt bit is set in the Trigger Module's Status Register on which the ROC might poll. Now the ROC or the FEP in case of CBV module starts its read-out procedure.

The duration of the conversion time (CTIME) preprogrammmed in the trigger module must be longer by 100 ns than the gate time (GTIME) that was set by switch H2 in this unit. There is no hardware check on these settings! The restriction must be taken into account by the software initialization of the Trigger Modules.

Event Rejection

In case of necessary data flow reduction a fast and/or slow mechanisms of event rejection are foreseen.

The Fast Clear method can be initiated in Master Trigger only. Together with the GTIME the Master Trigger Module starts the fast clear acceptance window (FCATIME) preprogrammed in 100 ns steps up to 6553.6 μs. During this time an eventuall Reject pulse obtained by this unit produces a Fast Clear signal distributed over specific Trigger Bus line, to all Trigger modules. This signal resets the local Dead Time FF as well as a started CTIME in all Trigger Modules without interrupting a read-out processor, i.e. the trigger is ignored as if it had never appeared. A Fast Clear ECL pulse of 50 - 500 ns duration (adjustable by on-board switch H4) is available on the front panel and can be fed into the Fast Clear inputs of the ADCs or TDCs. After a fixed settle down time (100 ns - 50 μs adjustable by on-board switch H3 in each module) the Total Dead Time line of the Trigger Bus is released and the system is now ready again to accept a new event trigger. These events never appear in the event data flow and are not counted by the event counters installed in each Trigger Module of the system.

The duration of fast clear acceptance time window (FCATIME) of the Master Trigger Module must be shorter than the shortest conversion time (CTIME) preprogrammed in the system. However, the FCATIME must be longer by 100 ns than the GTIME set in the module by switch H2. There is no hardware check on these settings! The restriction must be taken into account by the software initialization of the Trigger Modules.

The slow rejection of event can be initiated in any Trigger. During the CTIME or the read-out time each ROC or the FEP in case of a CBV module may detect invalid data and assert the Reject bit in its Trigger Module's Status Register. The Reject ECL-input from the front panel may also be used for this purpose. All Trigger Modules receive this Reject signal via the Subevent Invalid Trigger Bus line. Each ROC or the FEP in case of a CBV module can read this bit from its Trigger Module's Status Register. If the local CTIME is still in progress a Fast Clear signal is generated also as a front panel ECL pulse with 50 - 500 ns duration (adjustable by on-board switch H4) which can be fed into the Fast Clear inputs of the ADCs or TDCs. The SI signal stops the CTIME of all Trigger Modules where it is not yet timed out, to interrupt the ROC immediately.

Each Trigger Module can set or reset the Delay Interrupt line on the Trigger Bus by setting the corresponding bit in its Status Register. This bit is ORed on the Trigger Bus line 'Delay Interrupt'. The bit can be cleared by setting the Clear Delay Interrupt bit in the Status Register. The ECL Delay Interrupt Input level from the front panel of each Trigger Module is ORed to the Trigger Bus line 'Delay Interrupt'. The Delay Interrupt Trigger Bus line can be read from the Status Register. When the Delay Interrupt Trigger Bus line is set the Start of Read-out, and the Interrupt of the ROC or FEP, will be set if the local CTIME AND the Delay Interrupt disappeared. If the local CTIME disappeared already before the Delay Interrupt was set the interrupt is set 'normally' at the end of the CTIME and the later coming Delay Interrupt is ignored. The Delay Interrupt is cleared by setting the Subevent Invalid Trigger Bus line (Reject). With this mechanism one or more ROCs or FEPs can prevent others from the read-out of their modules by calculating Reject conditions during the Delay Interrupt time and setting the Reject Status Register bit in case of Reject.

At the end of the read-out time each ROC must sent a complete subevent with valid data or at least a subevent header with the Reject bit set. Each ROC may then set the Fast Clear bit in the Status Register for clearing of all ADCs or TDCs which are not read out before. This Fast Clear bit produces a pulse of 50 - 500 ns duration (adjustable by on-board switch H4) on the Fast Clear ECL-output at the front panel. This Fast Clear is not propagated through the Trigger Bus but is used for local clear only. After data handling each ROC must reset the Dead Time FF locally in its Trigger Module by setting the Clear Dead Time bit in its Status Register. With the disappearance of the ORed Total Dead Time Trigger Bus signal a new trigger can be accepted by the system.

Event Counter

To guarantee the synchronization of all Trigger Modules in the system a readable 5 bit event counter located in each Trigger Module is installed. The Master Trigger Module increments its event counter at the end of the FCATIME if no fast clear appears during this time. The four least significant bits of the counter of the Master Trigger Module is propagated via the Trigger Bus to all Trigger Modules. All other Trigger Modules increment their event counter at the end of their conversion time window (CTIME), but only if no FC appears via the Trigger Bus from the Master Trigger Module. Then they check their own counter against the four bits of the Trigger Bus. In any case of inequality a Trigger Module sets the readable bits 'Mismatch' and thereby 'Reject' in its Status Register by hardware. To take into account the propagation delays of the four counter bits on the Trigger Bus the CTIME should be set at least to the length of the FCATIME of the Master Trigger Module plus the propagation delay between the Master Trigger Module and the specific Trigger Module. The propagation delay is about 7 ns/m Trigger Bus cable.

Reset and Halt

A Reset signal is produced in each Trigger Module either on power-up or by software bit in the Control Register. It is propagated over the Trigger Bus clearing Interrupt Enable,Master Trigger select, GO and Trigger Enable bits in Control Registers of all Trigger Modules in the system. A Halt command (Go bit cleared) sets the Total Dead Time line of the Trigger Bus thus inhibiting the system. The event counters of all Trigger Modules are cleared by Reset also. (TDT-FF).

5.1.2 Trigger Signals and Time Windows

- Master Trigger 1 to 4 (MT1, MT2, MT3, MT4):
  Four Trigger Bus signals initiated by the experiment's fast trigger logic and produced by the Master Trigger. The leading edge of the first one is equivalent to the leading edge of Start signals for TDCs. The first Trigger initiates the Master Trigger (MT) output signal on the front panel and starts the CTIME in each Trigger Module.

- Master Trigger (MT):
  Master Trigger output on the front panel in ECL-logic. The duration of this signal is 50 - 500 ns (adjustable by on-board switch H1). It is set with the first of the Master Trigger (MT1-MT4). This is the pulse with which all converters in the system begin digitizing the analog signals from the experiment, i.e. the ADC gate or TDC start.

- Conversion Time Window (CTIME):
  It starts with the first Master Trigger (MT1-MT4) signal and it is active as long as the the slowest converter served by Trigger Module needs time for conversion. It is programmable in 100 ns steps up to 6553.6 μs. Its length must be at least equal to the length of the FCATIME of the Master Trigger Module <u>plus</u> the propagation delay between the Master Trigger Module and the specific Trigger Module. The propagation delay is about 7 ns/m Trigger Bus cable. It can be shortened by a Fast Clear (FC) signal.

- Start of Read-out (SOR):
  This pulse is triggered by the trailing edge of CTIME telling the ROCs to start the read-out of data from the converters. It is not produced if a FC signal was generated by the Master Trigger Module. It sets the flag in the Status Register of a Trigger Module. If the interrupt is enabled it produces an AEB interrupt in case of Fastbus, a VME interrupt in case of VME or a LAM in case of CBV and CVC.

- Delay Interrupt (DI):
  Any Trigger Module may set its Status Register bit Delay Interrupt which is ORed on the Trigger Bus line 'Delay Interrupt'. The ECL Delay Interrupt Input level from the front panel of each Trigger Module can also be ORed to the Trigger Bus line 'Delay Interrupt'. This signal delays the Start of Read-out, i.e. the interrupt of each module not yet interrupted. With this signal a ROC or a FEP can postpone the interrupt of all other processors during the evaluation of a Reject condition, thereby saving the read-out time of the other processors. The line is reseted by explicit setting the Delay Interrupt Clear bit in the Status Register or by the Reset and the Subevent Invalid Trigger Bus signals. The ECL Delay Interrupt Input signal must be cleare externally.

- Read-out Time ROC (RTR):
  This is the time period the ROC needs to read its crate.

- Fast Clear output (FC output):
  This pulse is produced by each Trigger Module in case of Fast Clear, Reject state or if the

local Fast Clear bit in the Status Register was set. It is a pulse of 50 - 500 ns duration (adjustable by on-board switch H4).

- Fast Clear Bus (FC Bus):
  This Trigger Bus signal is sent by the Master Trigger Module to all Trigger Modules to 'forget' an event. It is generated by the Master Trigger Module only if a Reject input appears within the FCATIME.

- Fast Clear Acceptance Time (FCATIME):
  It starts with the first Master Trigger (MT1-4) input signal. It is programmable in the range in 100 ns steps up to 6553.6 μs. During this time a Reject input signal in the Master Trigger Module causes a fast clear signal on the Trigger Bus forcing all Trigger Modules to 'forget' the current event. Each Trigger Module produces therefore a local Fast Clear (FC) front panel pulse of 50 - 500 ns duration (adjustable by on-board switch H4).

- Reject (Rej):
  The Trigger Bus signal Subevent Invalid (SI) can be produced by any Trigger Module if the Reject bit in the Trigger Module's Status Register is set. This Status Register bit can be set by a front panel Reject ECL input signal (or together with the Mismatch bit in the Status Register or by software. It tells all other ROCs and FEPs that one ROC or FEP has detected an invalid subevent.

- Total Dead Time (TDT):
  It is the logical OR of MT1-4, CT and RTR. It is a Trigger Bus line (TDT) as well as a front panel ECL-Signal (TD) on each Trigger Module.

- RESET (RES):
  It resets all Trigger Modules on Trigger Bus, i.e. stops counters, clears the Status Register and sets the Halt state. It does not clear the preset counters (FCATIME and CTIME). This signal produces by either power up or by setting the corresponding bit in the Control Register of a Trigger Module.

- GO:
  This Trigger Bus line when disabled forces the whole system in the Halt state. The Halt state generates the TDT signal. The Halt state is set by either power up or software reset or by clearing the GO bit in the Control Register of a Trigger Module. The Go state is only set by setting the GO bit in a Master Trigger Module by program starting the data acquisition in the whole system.

5.1.3 Trigger Sequence

The signals assumed to be active low are indicated by |....|

Normal trigger and read-out sequence (not CBV):

```
MT 1-4      --|.|------------------------------------------------------------
MT 1-4 FF   --|........................................................|-----
GTIME       --|..|-----------------------------------------------------------
MTOutput    --|..|-----------------------------------------------------------
FC Accept   --|..........|---------------------------------------------------
Reject Input-----------------------------------------------------------------
FC Output   ---------------------------------------------|.|-----------------
FC Bus      -----------------------------------------------------------------
SI Bus      -----------------------------------------------------------------
DT local    -----|.........................................|-----------------
TDT Bus     -----|.....................................................|-----
CTIME       --|.....................|----------------------------------------
SOR Interr. ------------------------|.|--------------------------------------
RTR         --------------------------|....................|-----------------
End of RTR  -----------------------------------------------|.|---------------
VSB-IR      -----------------------------------------------|.|---------------
FEP read-out-------------------------------------------------|...............
```

Normal trigger and read-out sequence for CBV :

```
MT 1-4      --|.|------------------------------------------------------------
MT 1-4 FF   --|........................................................|-----
GTIME       --|..|-----------------------------------------------------------
MT output   --|..|-----------------------------------------------------------
FCATIME     --|..........|---------------------------------------------------
Reject Input-----------------------------------------------------------------
FC output   ---------------------------------------------|.|-----------------
FC Bus      -----------------------------------------------------------------
SI Bus      -----------------------------------------------------------------
DT local    -----|.........................................|-----------------
TDT Bus     -----|.....................................................|-----
CTIME       --|.....................|----------------------------------------
LAM->VSB-IR ------------------------|.|--------------------------------------
FEP read-out--------------------------|....................|-----------------
End of Read -----------------------------------------------|.|---------------
```

Trigger and read-out sequence with Fast Clear:

```
MT 1-4      --|.|------------------------------------------------------------
MT 1-4 FF   --|..........|---------------------------------------------------
GTIME       --|..|-----------------------------------------------------------
MT output   --|..|-----------------------------------------------------------
FCATIME     --|..........|---------------------------------------------------
Reject Input--------|.|------------------------------------------------------
FC output   --------|.|------------------------------------------------------
FC Bus      --------|....|---------------------------------------------------
SI Bus      -----------------------------------------------------------------
SettleDownT -------------|....|----------------------------------------------
TDT MTM loc -----|............|----------------------------------------------
DT local    -----|.......|---------------------------------------------------
TDT Bus     -----|............|----------------------------------------------
CTIME       --|..........|---------------------------------------------------
SOR Interr. -----------------------------------------------------------------
RTR         -----------------------------------------------------------------
VSB-IR      -----------------------------------------------------------------
FEP read-out-----------------------------------------------------------------
```

Trigger and read-out sequence with subevent invalid SI:

```
MT 1-4      --|.|------------------------------------------------------------
MT 1-4 FF   --|.............................................|----------------
GTIME       --|..|-----------------------------------------------------------
MT output   --|..|-----------------------------------------------------------
FCATIME     -----|.......|---------------------------------------------------
Reject Input----------------|.|----------------------------------------------
FC output   ----------------|.|----------------------------------------------
FC Bus      -----------------------------------------------------------------
SI Bus      ----------------|...............................|----------------
DT local    -----|.................................|-------------------------
TDT Bus     -----|..........................................|----------------
CTIME       --|.............|------------------------------------------------
SOR Interr. ----------------|.|----------------------------------------------
RTR         ----------------|......................|-------------------------
LAM->VSB-IR ---------------------------------------|.|-----------------------
FEP read-out-----------------------------------------|.......................
```

5.1.4 Trigger Module Registers

Register Bit Layout

Status Register:

```
  16  15  14  13  12  11  10 9   8   7   6   5   4   3   2   1
----------------------------------------------------------------
|EON|    |DI |EC5|EC4|EC3|EC2|EC1|SI |MIS|TDT|    |MT4|MT3|MT2|MT1|  Read
|               |               |               |               |
|               |               |               |               |
|E+I|DI |DI |IRQ|   |   |   |SI |   |DT | FC|MT4|MT3|MT2|MT1|  Write
|clr|set|clr|clr|   |   |   |   |   |clr|   |   |   |   |   |
-----------------------------------------------------------------
```

Status Register bits:

- MT1-MT4:
  'Master Trigger' 1 to 4 Trigger Bus lines. The coded Master Trigger set in the Master Trigger Module inputs from the experiment's electronic. Can be set by program in the Master Trigger Module at any time. Cleared by RESET or at the end of TDT.

- FC:
  Fast Clear pulse of 50 - 500 ns duration (adjustable by on-board switch H4) available on local ECL front panel output after writing a 1. This signal is not bussed to other Trigger Modules. It does not set or reset anything except the front panel pulse to clear ADCs or TDCs not read out before.

- TDT:
  Common 'Total Dead Time' Trigger Bus line status. Cleared by RESET or by DTclr in all Trigger Modules.

- DT Clr:
  Clear local 'Dead Time' by writing a 1.

- MIS:
  'Mismatch' status. Occurs on Event Counter mismatch. Cleared by RESET.

- SI:
  Subevent Invalid Line of the Trigger Bus. Activated by writing a 1. If done within FCA time performs fast clear.

- EC1-EC5:
  Local 'Event Counter' bits 1 to 5. Incremented at the end of CTIME if no Fast Clear

(FC) came from the Master Trigger Module. In the Master Module it is incremented at the end of the Fast Clear Acceptance Time (FCATIME) if no REJECT was set. Cleared by RESET.

- DI:
  The 'Delay Interrupt' Trigger Bus line. Set by writting DIset=1 cleared by DIclr=1 in all modules with DI on. (the ECL Delay Interrupt Input signal must be cleared externally).

- EON:
  The data are ready for read out in this module. Cleared by RESET or by E+Iclr.

- E+I clr:
  Clear event and 'Interrupt' in the Trigger Module by writing a 1. In case of VME module also clears IRQ* on the VME bus.

- IRQ clr:
  Used in VME standard only. Clears interrupt generator, used in case of error condition.

Control Register:

```
  16  15  14  13  12  11  10  9   8   7   6   5   4   3   2   1
-----------------------------------------------------------------
|   |   |   |   |   |   |   |   |   |   |   |   |   |MTM|GO |INT|   Read
|               |               |               |            ena|
|               |               |               |               |
|   |   |   |BUS|BUS|   |   |   |   |RES|MTM|HLT|INT|MTM|GO |INT|   Write
|   |   |   |dis|ena|   |   |   |   |   |dis|   |dis|ena|   |ena|
-----------------------------------------------------------------
```

Control Register bits:

- INT:
  Interrupt enabled in this Trigger Module when INT=1. Enebled by writting INTena=1, disabled by INTdis=1.
- RES:
  Reset the Trigger Module by writing a 1. Produces a Reset pulse in the Trigger Module and on the Trigger Bus Line thereby reseting all Trigger Modules. Produced by power up also.
- GO:
  GO Trigger Bus line status. In 'Go" state by writing GO=1, in 'Halt' state by writing HLT=1.
- MTM:
  Master Trigger Module. Selected by writing MTMena=1, disabled by writing MTMdis=1.

FCATIME Register:

Fast Clear Acceptance Time is programmable in 100 ns steps up to 6553.6 μs (= 16 bits) (read/write).

CTIME Register:

CTIME is programmable in 100 ns steps up to 6553.6 μs (= 16 bits)(read/write).

Registers Addresses

All FASTBUS trigger registers can be accessed from the AEB via internal EBI port bus and by the FEP. EBI addresses from AEB (baseaddr. = 0) are:

- status 3001000
- control 3001004
- FCA 3001008
- CVT 300100C

EBI addresses from E6 (Crate nr.1) are:

- status F2201000
- control F2201004
- FCA F2201008
- CVT F220100C

CAMAC trigger registers are red by F(0) and written by F(16) functions. Subaddresses A values are:

- status A(0)
- control A(1)
- FCA A(2)
- CVT A(3)

VME module registers can be accesed in extended addressing mode as follows:

- status 02000000
- control 02000004
- FCA 02000008
- CVT 0200000C

VME Interrupt Settings

Setting of the VME interrupt level IRQ*, interrupt ackowledge level INTACK* and of the interrupt vector value can be done by means of jumpers and the switch as shown in the chapter " Internal jumpers and switches".

5.1.5 Front Panel

Front Panel LEDs

-5	Power	T1	
+5	Power	T2	Master Trigger Bus lines
TE	TBUS terminator in	T3	1 - 4
MT	Master Trigger Module, control bit	T4	
GO	Go, TBUS bit		
TD	Total Dead Time, TBUS bit	MI	Mismatch, status bit
IR	Interrupt request, status bit	FC	Fast clear, TBUS line
ID	Interrupt delay, TBUS line	SI	Subevent invalid, status bit

Trigger Bus Connector

The Trigger Bus TBUS is a differentially driven bus (it is not an ECL bus!!). On the front panel it is represented by the 34-fold (17 pairs) flat cable connector with 3 pairs reserved for future use.

Pin	Describtion	
1-2	ID	Interrupt delay
3-4	SI	Subevent invalid
5-6	GO	Go
7-8	TDT	Total dead time
9-10	RESET	Reset
11-12	FC	Fast clear
13-14	MT1	
15-16	MT2	Master trigger lines
17-18	MT3	1 - 4
19-20	MT4	
21-22	EC1	
23-24	EC2	Event counter lines
25-26	EC3	1 - 4
27-28	EC4	
29-30		
31-32		Spare pairs
33-34		

ECL Front Panel I/O

The inputs/outputs available at the front panel of each Trigger Module are ECL differential standard signals. One 16 pin (8 pairs) flat cable connector for input and one for output are installed.

The ECL inputs are available on the upper 16 pin connector on the front panel of each trigger module.

Pin	Description	
1-2	T1	
3-4	T2	Trigger inputs
5-6	T3	1 - 4
7-8	T4	
9-10	REJECT	Reject
11-12		
13-14		
15-16	DI	Delay interrupt

The ECL outputs are available on the lower 16 pin connector on the front panel of each trigger module.

Pin	Description	
1-2	MT	Master trigger
3-4	FC	Fast clear
5-6	SOR	Start of readout
7-8	TDT	Total dead time
9-10	GO	Go bit
11-12	GO*	Go inverted
13-14		
15-16		

5.1.6 Internal Jumpers and Switches

There are 4 switches, 4 sliders each, that are installed on the trigger board in order shown below and enabling to set 4 important pulses. The sum of the constant value and of that switched on defines the duration of the pulse.

<u>ECL MASTER TRIGGER PULSE SWITCH:</u>

Switch H1

Installed	Added when slider on				TOTAL RANGE
on board	1	2	3	4	
ns	ns				ns
50	50	50	100	250	50 - 500

<u>GATE TIME SWITCH:</u>

Switch H2

Installed	Added by switch				TOTAL RANGE
on board	1	2	3	4	
ns	ns				ns
50	50	50	100	250	50 - 500

SETTLE DOWN TIME SWITCH:

Switch H3

Installed on board	Added by switch				TOTAL RANGE
	1	2	3	4	
us	us				us
0.1	1.9	3	10	35	0.1 - 50

ECL CLEAR PULSE SWITCH:

Switch H4

Installed on board	Added by switch				TOTAL RANGE
	1	2	3	4	
ns	ns				ns
50	50	50	100	250	50 - 500

VME IRQ* LEVEL JUMPER: The interrupt request level IRQ* in the VME module can be set by shortening the proper pins in the IRQ* LEVEL CONNECTORS row:

```
-------------------------------------
|   IRQ* LEVEL CONNECTOR SETTING    |
|             FOR IRQ4*             |
|-----------------------------------|
|   IRQ1*   IRQ2*   IRQ3*   IRQ4*   |
|     x       x       x       x     |
|                             |     |
|     x       x       x       x     |
-------------------------------------
```

VME INTACK* LEVEL JUMPERS: The IRQ* level must correspond with setting for encoded address bits that are sent over the bus during the interrupt acnowledge cycle. This can be done by jumpers installed in the INTACK* LEVEL row:

```
----------------------------------------------------------
|            INTACK* LEVEL CONNECTOR SETTING             |
|                          FOR                           |
|--------------------------------------------------------|
|    IRQ1*    |     IRQ2*    |    IRQ3*    |    IRQ4*    |
|--------------------------------------------------------|
|    x x x    |     x x x    |    x x x    |    x x x    |
|        |    |       |      |      | |    |    |        |
|    x x x    |     x x x    |    x x x    |    x x x    |
|    | |      |     |   |    |    |        |      | |    |
|    x x x    |     x x x    |    x x x    |    x x x    |
----------------------------------------------------------
```

VME INTERRUPT VECTOR SWITCH: The interrupt vector that is sent over the VME bus during the interrupt acknowledge cycle can be choosen by means of the INTERRUPT VECTOR SWITCH. Every single slider in "on" position activates the corresponding bit in the interrupt vector.

5.1.7 Cirquit Description

The FASTBUS trigger module is installed on the PCB located at the rear of the crate. The same PCB is used in the CAMAC module being connected to the dataway over an interface.
The VME module was designed as a new board, however, its trigger part is a copy of those in CAMAC and FASTBUS. Again, the trigger part communicates with the VME bus over an interface.
The performance of the trigger cirquit is controlled by:

- status register described in the manual on page 11 and shown in the diagram on page 10
- control register described in the manual on page 13 and shown in the diagram on page 11.
- fast clear acceptance time (FCATIME) register an internal register installed in two 74LS593 IC's shown on page 2 of the diagram.
- conversion time (CTIME) register an internal register installed in two 74LS593 IC's shown on page 3 of the diagram.

The commands used to acces the registers are decoded in the commands decoder FDEC shown on page 11 of the diagram and the necessary addresses or NAFs are described on page 14 of this manual. For better understanding of the trigger work it should be noted that the module represents a part of the trigger system organized around the trigger bus. Because of that all trigger bus signals used in the module are the output signals from the bus. This rule is valid in the single trigger system also.

Normal trigger sequence acc to diagram 1 and 2

The sequence starts with registration of the rising edge of any trigger pulse that is supplied either by software /10-8G/ or by ECL input /8-4G/. The triggers that are registered in the trigger register of the master unit /1-4A:E/ are ORed /1-3F/and start the GTIME* /1-9D/ together with the master trigger output MT /1-9G/.
The GTIME* pulse starts the measurement of the FCATIME*/2-2B/ and CTIME* /3-1B/. In both cases the measurement is performed by counting the clock pulses /4-4B/ in 16 bit counters built of two 74LS593 ICs /2-5E:H for FCA and 3-5E:H for CTIME*/. The rising edge of the CTIME* pulse sets the dead time flip-flop in the module /5-9E/. The signal DTIN produced by this flip-flop is sent on the trigger bus /7-6F/ to be ORed with the dead times of other triggers.If the interrupt was enabled in this module /11-6D/ the interrupt pulse /6-9B/ is set in parallel to the DTIN and to the start of read out ff /6-9C/. The INT and SOR signals can be delayed by setting the delay interrupt bit DIIN in the status register /10-9C/ of any trigger on the bus. The DIIN is first sent on the trigger bus /7-6H/ and used later as the DIOUT signal /6-0B/ in all trigger modules. At the end of the read-out routine the read out processor prepares the trigger module for the next event by writting E+I clr and DT clr in the status register /10-8E

and 10-9F/. E+I clr pulse resets the event ff /6-2C/ and interrupt ff /6-4C/while the dead time clr pulse clears the dead time ff /5-5D/. Together with the last dead time in the slowest module of the system the total dead time line on the trigger bus is cleared. This clears the trigger register in the master trigger module /1-4A:E/. The events are counted in the event counter shown on page 9 of the diagram. The master counts at the falling edge of the FCATIME /9-0E/ while slaves at the end of the CTIME /9-0D/. The counters are compared in slave modules at the end of start of read-out pulse /9-1C/ to set the mismatch bit /9-9C/ in case of unequal contents of counters. The resulting MI bit lights the coresponding LED /4-6E/ and sets the bit in the status /10-0C/.

Fast clear sequence acc to diagram 3

The REJECT pulses that start the fast clear sequence come within the FCATIME either from the ECL input /8-5E/ or are sent per software /10-8F/. The time condition is arbitrated in the event reject section shown on page 6 of the diagram. If it is fullfilled the FCIN* pulse /6-8F/ is sent on the trigger bus. The related FCOUT* pulse received in each trigger module in the system stops the normal triggering sequence that was described above. This is done by clearing the FCATIME* /2-8B/, CTIME* /3-8B/ and DTIN* /5-9E/. The FCOUT produces the SC pulse that triggers the settle down time ff /5-5E/ thus expanding the local dead time DTIN /5-9E/ and disables the counting of the reseted event /9-0E/.

SI sequence acc to diagram 4

The REJECT pulse that comes after the FCATIME was ended results in the subevent invalid SI sequence. The SIOUT pulse that is produced at the beginning of the sequence cuts the CTIME measurement, if on /3-1A/ and triggers the settle down time ff /5-5E/. In parallel the SI LED is lighted /4-6D/ and SI bit in the status register is set /10-1C/.

Software trigger

To enable for proper asynchroneus software triggering the EP610 PLD chip was installed in the status register /10-4H:G/. The chip is loaded with the triggering combination of 4 least significant bits during the load status operation LSTAT*. It keeps this information as long as the current event is in progress. The software trigger sequence starts with presetting of the trigger register /1-4A:E/ immediatelly after the TDT was cleared. The hardware trigger is inhibited at that time. The timing that follows is the same as in the case of hardware trigger.

5.1.8 Testing

1. Inspect the board for:

```
- mechanical errors,
- mounting errors,
- settings of switches and jumpers,
- installation of terminators.
```

2. Apply the power.
3. Read and write the registers and counters checking for consistency.
4. Check LEDs for consistency with status and control registers.
5. Use logic analyzer to obtain the diagrams 1, 2, 3 of this manual.

```
- use test set-up and program from P. Liebold in case of
  CAMAC module
- use test set-up and program from D. Schall in case of VME,
- the FASTBUS modules can be tested as CAMAC units first
  and reconfigured for FASTBUS later.
```

6. integrate the module in multitriger system and check proper operation of event counters /mismatch - MI LED/ and fast or software clear /FC and SI/.

5.1.9 Troubleshooting

1. Check conditions in the crate /power supply, overload, overheating etc/.
2. Check installation of cables, terminators,switches and jumpers:

```
- the arrows on plugs and sockets must match,
- the first and last module on the trigger bus
  must be terminated /observe LEDs/,
- GTIME setting must be shorter than CTIME and FCATIME,
- in case of VME module the interrupt level jumper and
  the interrupt  acknowledge level jumper must match.
```

3. Check if only one master in the system.
4. Check GO, TDT, DI, FC, SI LEDs for proper operation.
5. Check if proper software in use:

```
- the sequence of hardware initialization,
          1. after power-up the system must be reseted,
          2. enable trigger bus in every unit,
          3. load regisers of all slaves,
          4. load registers of the master,
          5. write GO in the master unit.
- the values of GTIME, CV and FCA times:
          1. GTIME must be shorter than CV and FCA
          2. FCATIME in master module must be shorter than
            the shortest CVTIME in the system,
- the proper event read-out routine,
- the proper use of reset function,
- the proper use of TRIGBUSEN*.
```

6. contact EL/EX group representative.

Chapter 6

Commands and Functions (CMD)

6.1 Commands for Listmode, Loading and Testing

Commands will be received by the NET processor as DECnet Message Control Blocks (MCB). The NET processor scans Command and Subcommand code to determine the desired action. The NET processor determines the destination processor, writes the message to the destination message buffer and triggers the destination processor according the protocol outlined in chapter **??**.

6.1.1 Start Acquisition

- Destination processor: FEP 1, EB

- Function: Clear buffer/event counter in EB, Reset Stop FF in trigger unit of FEP 1 and set HG to GO.

6.1.2 Stop Acquisition

- Destination processor: FEP 1, EB

- Function: Set Stop FF in trigger unit of FEP 1, set HG to HALT, send the current number of subevents in FEP - subevent queue to the EB, after completion of the last subevent the EB sends 'last buffer' to host.

6.1.3 Load VME Tasks

- Destination processor: NET

- Function: Load system software for one FEP, the EB or the STR

6.1.4 Load VME Parameters

- Destination processors: NET
- Function: Load parameter list to one FEP, the EB or the STR

6.1.5 Initialize Front End Systems

- Destination processors: all FEPs
- Function: Start initialization procedures and report errors to host

6.1.6 Set Acquisition

input:standard, network

- Destination processors: one FEP, EB, STR
- Function: Switch the data input to the message buffer path to accept input data from the NET processor or to the listmode buffer path

all channels

- Destination processors: selected FEPs
- Function: Read out all channels of ADC system

valid channels

- Destination processors: selected FEPs
- Function: Read out only valid ADC system channels according to bit pattern

enable, disable subevent pre-scanning

- Destination processors: selected FEPs
- Function: activate/deactivate the user-supplied subevent pre-processing routine

enable, disable software trigger

- Destination processor: EB
- Function: Send the output buffers to STR for processing or directly to host

set output standard, net

- Destination processors: one selected FEP, EB, STR
- Function: Send the output buffers to NET or to next level processor or host

Chapter 7

FASTBUS readout (FBREAD)

7.1 Processes on AEB

7.1.1 AEB_READOUT

- started by STARTUP
- does not run in system mode, but is (first) the only process to run on the AEB
- create CDM (control data module). The size and structure of the CDM is specified in file CDM.H
- initialize communication with VSB. (details later)
- receive and execute cmds from VSB via EBI
- receive trigger interupts (via trigger board + EBI) from experiment
- if a terminal has been attached to the AEB, a monitor function can be turned on by pressing any key. It is turned off by option 'x'.

7.1.2 AEB_MONITOR

can be started as a separate process, but all functions are as well available under AEB_READOUT after a terminal has been attached and any key has been pressed.

7.1.3 Other processes

First, there should be no other processes on AEB except AEB_READOUT. However, there are certain mechanisms implemented, to possibly run an additional process parallel to AEB_READOUT:

- locking mechanism of readout list for a given trigger number: a readout module table is locked during LOAD LIST execution (no other process must access the list!), readout, execution of RESET (other processes may read the table at the same time). The locking and releasing is performed using the TAS assembler instructions testing and setting flags belonging to each readout list for a given trigger number (see /n11/defs/cdmlock.c).
- Because there is free CPU time between the SC (start conversion) and the SR (start readout) signal from the trigger module, another process might be running during this interval. The other process should have a low priority and AEB_READOUT might fall asleep using "tsleep(1);" (5 msec). However, this time intervall is probably too long!
- Alternative choice: provide a default call to a routine "after_readout();", which is a dummy routine by default and can be replaced by any other user routine. This is not yet implemented.

7.2 Organization on AEB

7.2.1 Source files

Source code (on **/n11/FBNEW**):

file name	purpose
AEB_READOUT.c	main program for readout process: • create CDM • execute commands • perform readout • monitor (optional)
AEB_MONITOR.c	main program monitor process
cmdexec.c	execute a command from FEP
clearacq.c	execute CLEAR command
datmod.c	create an OS9 data module and get pointer to data
ebi_conf.c	determine EBI base addresses
ebi_comm.c	do communication via EBI: • initialize communication • put msg into queue to FEP • send msg to FEP
fbutil.c	FASTBUS handling routines (currently not used) • open/close FB environment • error handling
monitor.c	auxiliary monitor routines
panel.c	display handling (VT220)
readout.c	standard readout routine
loadlist.c	load readout list into CDM
trigger.c	execute TRIGGER (internal GATE) command for a selected readout list (test only!)
LRS18XX.c	init, clear and readout for LRS18xx modules
STR136.c	init, clear and readout for STR136
STR200.c	init, clear and readout for STR200

Include files (on **/n11/DEFS** except those marked by ∗):

file name	purpose
aeb_error.h	error codes (includes "xgoovme.h")
ebi_handle.h	aux. macros to write/read parts of long words to/from EBI, and to compute relative and absolute addresses in EBI RAM.
ebi_map.h	EBI memory mapping and TCB definitions
fbmacros.h	some auxiliary macros for FASTBUS handling (includes "stdio.h")
fbmodules.h	definitions for FASTBUS modules: CSR#0, command codes, CSR addresses, max. number of channels for different module types, etc.
cdmlock.h	lock a single readout table (S_RDTTAB): `p_rdttab` must point to the table to be locked (or released) and a `#define` *verb* must be specified before `#include <cdmlock.h>`. *verb* may be `ALLOC_READ`, `ALLOC_WRITE`, `FREE_READ`, `FREE_WRITE`.
∗ cfbdef.h	FASTBUS definitions
panel.h	definitions for display handling (VT220)
s$cdm.h	structure definitions for CDM (includes "trigger.h")
s$cmd.h	structure of command buffer
∗ stdio.h	standard I/O
trigger.h	definitions concerning trigger module
vcmd.h	definition of command types and subtypes
xgoovme.h	definition of global error codes (generated on VAX)

7.2.2 Compile and link commands

The following commands are executed after

```
CHD /n11/fbnew
CHX /n11
```

All subroutines are compiled using the command

```
MAKELIB
```

which merges all routines into the library file AEB.L. The main routine for the entire readout and communication, AEB_READOUT, is compiled and linked by

```
MAKE [T=AEB_READOUT]
```

and AEB_MONITOR by

```
MAKE T=AEB_MONITOR
```

7.2.3 User readout routine

A user readout routine can be implemented in the following way:

Entry name

readout()

Parameters

none

Includes

`#include <aeb_error.h>`	error codes
`#include <ebi_map.h>`	EBI memory mapping and TCB definitions
`#include <cfbdef.h>`	FASTBUS definitions
`#include <s$cdm.h>`	structure definitions for CDM

Interface to AEB_READOUT

`extern long l_lwords_rd;`	(output) set to total length of data to be transferred to FEP
`extern long *p_start_data;`	(input) start of region in EBI, where data can be written to (output) start address of data
`extern S_CDM *p_cdm;`	(do not modify!) points to CDM
`extern S_RDTTAB *p_rdttab;`	(do not modify!) points to readout table selected by trigger number
`return(l_status);`	return value from readout() • XVME_AEB_SUCCESS : readout successful • XVME_AEB_ERROR : readout error occured • XVME_AEB_TAB_LOCKED : readout table currently not available, no readout performed

Compile user_readout.c by

```
MAKE -f=mksub T=user_readout
```

and link to AEB_READOUT by

```
MAKE USERLIB=-l=user_readout.r
```

7.3 FASTBUS environment

The FASTBUS default environment `FBDEID` is used by the readout process. The FASTBUS modules are read using asynchronous block transfer via `fb_read_dat_block`, synchronization and error handling is done using `fb_completion_wait` afterwards. The environment parameters used are set as follows:

```
fbopen();
fb_set_par(FBDEID, FBNOWT, FB_TRUE);
fb_set_par(FBDEID, FPPRIV, BUSRST);
fb_set_par(FBDEID, FPARBL, 6);
```

7.4 Using OS9-signals to trigger readout

Each signal send from one process to another is "accompanied" by a 16-bit signal code. This code can be used to code some information.

01.08.89:
the delay time from a front panel interrupt to signal which occurs in a `signal_handler()` activated by `intercept()` is about $250\mu s$. An interrupt service routine has been used for the level 1 front panel interrupt (IR vector directly replaced by address of ISR) to send a signal to the readout process.

This time is much too long, hence signals cannot be used to transfer the trigger signal to the readout routine.

7.5 Protocol for communication between AEB and VSB (FIC) via EBI

7.5.1 Segmentation of EBI Memory – Transfer control blocks (TCB) and data regions

All transfers between AEB and VSB are performed using transfer control blocks TCB's at fixed memory positions in EBI's CSR. There are (up to now) three transfer types (TT's):

1. commands from VSB to AEB
2. messages from AEB to VSB
3. list mode data from AEB to VSB

Each address constant `EBI_TCB_......` points to one transfer control block, which has the following structure (structure `S_TCB`):

```
typedef struct s_tcb {
                long    l_tcb_type;  /* transfer type        */
                long    *p_tcb_start; /* rel.start address   */
                long    l_tcb_lwords; /* length of data      */
                long    l_tcb_status; /* transfer status     */
} S_TCB;
```

structure S_TCB

31 28 24 20 16 12 8 4 0	Offset
Transfer type	0
Relat. start addr. of data	4
length of data (lwords)	8
Transfer status	12

Figure 7.1: Transfer control block for communication between FIC and AEB.

Transfer paths:

- EBI_TCB_V2A_CMD: commands from VSB to AEB (EBI_A2V_TT_DATA_ON_CMD),
- EBI_TCB_A2V_MSG: messages from AEB to VSB (A2V_MSG). These messages may be either error messages during command execution and FASTBUS readout (EBI_A2V_TT_DATA_ON_CMD), or completion messages after command execution (EBI_A2V_TT_ERRMSG)
- EBI_TCB_A2V_LMDATA: list mode data from AEB to VSB (EBI_A2V_TT_LMDATA). This TCB is at the same address as for messages to VSB, because messages and listmode data are send synchronous to the FIC anyhow.

The following data regions are reserved for transfers: (current size 16kB each: 24 FASTBUS modules $\times$ 100 channels $\times$ 4 Byte $\approx$ 10kB needed for V2A_CMD (load list) and A2V_LMDATA. A2V_MSG could be smaller currently, but there might be a command to read all modules by hand.

EBI_ADD_V2A_CMD	start addr. of region for V2A_CMD transfer (max. EBI_V2A_CMD_MAX_LWORDS lwords)
EBI_ADD_A2V_MSG	start addr. of region for A2V_MSG transfer (max. EBI_A2V_MSG_MAX_LWORDS lwords)
EBI_ADD_A2V_LMDATA	start addr. of region for A2V_LMDATA transfer (max. EBI_A2V_LMDATA_MAX_LWORDS lwords)

7.5.2 Initialization of communication (AEB)

- disable and clear EBI front panel interrupt
- disable and clear interrupt VSB to AEB.
- clear transfer of MSG's and (LMDATA) listmode data to VSB,i.e. : set TCB:
  type=`EBI_TT_EMPTY`,
  and for sake of completeness:
  start=`NULL`, *lwords*=0,
  status=`EBI_ERC_SUCCESS`
  ($\boxed{???}$ also for CMD from VSB to AEB)
- clear queue of pending msgs to VSB.

7.5.3 Initialization of communication (VSB)

- clear and enable interrupt AEB to VSB
- clear CMD transfer from AEB to VSB, i.e. set TCB:
  type=`EBI_TT_EMPTY`,
  perhaps also:
  start=`NULL`, *lwords*=0,
  status=`EBI_ERC_SUCCESS`

7.5.4 CMD from VSB to AEB

VSB action	AEB action
• VSB writes command buffer (structure `S_CMD`) to EBI, starting at `EBI_ADD_V2A_CMD`. • VSB writes TCB (type=`EBI_V2A_TT_CMD`, start, lwords) to EBI, starting at `EBI_TCB_V2A_CMD`. • VSB sets interrupt to AEB via EBI (`EBI_CMD_SET_IR_V2A`)	
	• AEB recognizes IR by polling on the corresponding IR bit within the Status Register of EBI. The transfer type must be `EBI_V2A_TT_CMD`. If set, the command is fetched from EBI and executed. • AEB clears IR from VSB to AEB (`EBI_CMD_CLR_IR_V2A`) • A msg is sent to VSB containing the completion code of the command: *type*=`EBI_A2V_TT_DATA_ON_CMD`, *start*=`EBI_ADD_A2V_MSG`, *lwords*, *status*=return code from executed command • AEB sets the interrupt (`EBI_CMD_SET_IR_A2V`) to send the msg to VSB
• FIC, waiting for `EBI_A2V_TT_DATA_ON_CMD`, receives the msg ...	

7.5.5 Error message from AEB to VSB

VSB action	AEB action
	• error occurs on AEB • The error code (usable on VAX with msg library) and, if required, data is returned to VSB by setting up a MSG transfer from AEB to VSB (a list of the possible error codes is given in section **??**) i.e. set TCB: *type*=`EBI_A2V_TT_ERRMSG`, *start*=`EBI_ADD_A2V_MSG`, *lwords*, status=error code) • AEB sets IR to VSB (`EBI_CMD_SET_IR_A2V`) to initialize transfer of MSG to VSB.
• FEP receives interrupt, fetches msg from EBI, sets transfer type within TCB to `EBI_TT_EMPTY`, and clears interrupt (`EBI_CMD_CLR_IR_A2V`)	

7.5.6 Trigger interrupt and transfer of listmode data to VSB

Polling is used for the following reasons

- fast (time delay is of the order of a few instructions)
- command execution and readout should be carried out by only one process, because any other process running concurrently with the readout process means a possible delay of the activation of the readout process in the order of 1 tic (5 msec).
- furthermore, readout and command execution must be synchronized (e.g. loading of a new readout list !). Hence, these two tasks must be carried out both either within the same (!) interrupt service routine (perhaps difficult to debug ?), the signal_handler (time delay of more than 200 μs !), or in the main routine, when the process has been woken up, and the signal has been put into a queue by the signal_handler (even slower, but nice peace of software).

VSB action	**AEB action**
	• AEB polls on interrupt bit from event trigger in EBI status register. • If set, AEB checks whether the last LMDATA transfer of listmode data to VSB has been executed, i.e. whether (EBI_TCB_A2V_LMDATA)->l_tcb_type = EBI_TT_EMPTY. • If the last transfer has been ended, the trigger number of this event is fetched from the trigger module (TRL_TRIGGER_NUMBER). • The data from FASTBUS modules are read using the readout list for this trigger number (locked !) and written to EBI, starting at EBI_ADD_V2A_LMDATA. • clear trigger interrupt bit in EBI status register • set up transfer of LMDATA to VSB: using EBI_TCB_A2V_LMDATA, i.e. set TCB to: *type*=EBI_A2V_TT_LMDATA, *start*=EBI_ADD_A2V_LMDATA, *lwords*, *status*=EBI_ERC_SUCCESS or error code • AEB sets IR to VSB (EBI_CMD_SET_IR_A2V) to initialize transfer of LMDATA to VSB. (if there will be more than one FASTBUS crate atteched to one FEP, only subcrate 1 is allowed to set this interrupt)
• FEP receives interrupt, fetches data from EBI, sets transfer type within TCB to EBI_TT_EMPTY, and clears interrupt (EBI_CMD_CLR_IR_A2V) • If acquisition is active, the trigger logic is enabled afterwards (TRL_CMD_EN_TRIGGER).	

7.5.7 Queue for MSG's from AEB to VSB

All MSG's (data requested by CMD from VSB and error messages by AEB) are hold in an internal msg queue on the AEB and written to EBI using TCB `EBI_TCB_A2V_MSG`, i.e. set TCB:
type,
start=`EBI_ADD_A2V_MSG`,
lwords,
status=`EBI_ERC_SUCCESS` or error code

7.6 Supported FASTBUS Modules

7.6.1 Struck STR136/2 ECLPIO

- CSR#0 = $ 6101 ....
- data consist of two 32-bit words.

7.6.2 Struck STR200 Scaler

- CSR#0 = $ 6823 ....
- data consist of 32 words of 32 bit

7.6.3 Le Croy 1872/75/82/85 (64/96 channel 12/15 bit TDC/ADC)

- CSR#0 = $ 1048 .... (LRS 1872)
- CSR#0 = $ 1049 .... (LRS 1875)
- CSR#0 = $ 1044 0300 (LRS 1882 F)
- CSR#0 = $ 1045 3300 (LRS 1885 F)
- only N-mode supported, because buffering of up to 8 events under F-mode leads to the necessity of synchronization
- **Pedestals:** They are provided by the user and transferred to AEB as part of the readout list loaded down from VAX via VME.
- Currently pedestal subtraction on the AEB is not supported.

7.6.4 Kinetic Systems Corp. KSC F432 (64 channel 14-bit TDC)

- CSR#0 = $ 1008 ....
- pedestals are stored within the module, automatic zero supression available.

7.7 Commands

The standard command format is
(structure S_CMD)

```
typedef struct s_cmd {
    long   l_cmd_lwords;           /* data length (lwords)    */
    long   l_cmd_subtype;          /* command subtype         */
    long   l_cmd_type;             /* command type            */
} S_CMD;
```

structure S_CMD

31 28 24 20 16	12 8 4 0	Offset
Data length (lwords)		0
Type	Subtype	4
data long word #1		8
⋮		
data long word #n		4+4n

Figure 7.2: $\boxed{???}$ Preliminary version of command buffers transferred from VSB to AEB.

7.7.1 Load Readout List

type: VCMD__LOA_FB
subtype: VCMD__S_LOA_FB
data: FASTBUS readout list for this subcrate (only #1 allowed) including 2 long words belonging to header (crate, offset, subcrate / trigger number).

Actions:

- Lock readout list for selected trigger number. If locked, repeat later.
- Build up readout module table (array of `FB_module`) with following informations for each module:
  - geographical address
  - module id (CSR #0)
  - number of active channels (i.e.: $1 \ldots n$)
  - module header containing geogr.addr. and mod.id, only the data length must be inserted after readout

  - number of pedestal data
  - pedestal data
  - entry addresses of routines for INIT, READ und CLEAR of module
  - number of bytes to be read by `fb_read_dat_block`

- The module id of each FASTBUS module is read and comparted to the value specified in the readout list.
- The INIT routine is called, which performs the following actions:
  - Check whether specified number of active channels is valid
  - Check whether number of pedestal data is valid (0; for ADC's and TDC's, also a number equal to the number of active channels is allowed)
  - LRS188x modules are checked whether they are in N-mode.
- Every error detected for any module is reported by sending a separate error message to VSB and an overall flag is set, to report an error during initialization.

7.7.2 START ACQUISITION

type: `VCMD__STA_AC`
subtype: `VCMD__S_STA_AC`
data: -

Actions:

- Check, whether at least one readout list has been loaded before. If not, report error.
- If acquisition had been started before, report error.
- Set internal flag for 'acquisition active' to 1.

7.7.3 STOP ACQUISITION

type: `VCMD__STO_AC`
subtype: `VCMD__S_STO_AC`
data: -

Actions:

- If acquisition had been stopped before, report warning.
- Set internal flag for 'acquisition active' to 0. (`TRL_CMD_EN_TRIGGER`).

7.7.4 RESET

type: `VCMD__RESET`
subtype: `VCMD__S_RESET`
data: -

Actions:

- reset FASTBUS
- reset locks for readout module tables for all trigger numbers
- clear / reset each module currently in readout module table.
- initialize communication with VSB (page 62)

7.7.5 TRIGGER FASTBUS modules (for test only)

type: `VCMD__FB_TRIGGER`
subtype: trigger number
data: -

Actions:

- lock readout module table for selected trigger number
- trigger (internal gate) all modules contained in this readout table.
- free readout module table for selected trigger number

7.7.6 Execute FASTBUS Call

type: `VCMD__FB_CALL`
data: see below
(structure `S_FBCALL`)

```
typedef struct s_fbcall {
    long   l_fbcall_id;             /* FB environment id       */
    long   l_fbcall_prim_add;       /* primary address         */
    long   l_fbcall_sec_add;        /* secondary address       */
} S_FBCALL;
```

Action:

- perform FASTBUS call.

structure S_FBCALL

31 28 24 20 16 12 8 4 0	Offset
FASTBUS environment id	0
primary address (geogr. address)	4
secondfary address	8
additional data(see below)	12

Figure 7.3: $\boxed{???}$ Preliminary version of command buffers transferred from VSB to AEB.

- **fb_read_csr**
  subtype: `VCMD__S_FB_FRC` (=1)
  data: (structure `S_FBCALL`)
  Action: read single data from CSR

- **fb_write_csr**
  subtype: `VCMD__S_FB_FWC` (=2)
  data: (structure `S_FBCALL`) + (1 lword) data for CSR
  Action: write single data to CSR

- **fb_read_data**
  subtype: `VCMD__S_FB_FRD` (=3)
  data: (structure `S_FBCALL`)
  Action: read single data from data space [1]

- **fb_write_data**
  subtype: `VCMD__S_FB_FWD` (=4)
  data: (structure `S_FBCALL`) + (1 lword) data to be written to data space
  Action: write single data to data space

- **fb_read_csr_block**
  subtype: `VCMD__S_FB_FRCB` (=5)
  data: (structure `S_FBCALL`)
  + (1 lword) n = number of data lwords to be read from CSR
  + (n lwords) data lwords to be read from CSR
  Action: block read from CSR

- **fb_write_csr_block**
  subtype: `VCMD__S_FB_FWCB` (=6)
  data: (structure `S_FBCALL`)
  + (1 lword) n = number of data lwords to be written to CSR
  + (n lwords) data lwords to be written to CSR
  Action: block write to CSR

[1] for some modules this action requires a re-read command, which will be issued automatically

- **fb_read_data_block**
  subtype: `VCMD__S_FB_FRDB` (=7)
  data: (structure `S_FBCALL`)
  + (1 lword) n = number of data lwords to be read from data space
  + (n lwords) data lwords to be read from data space
  Action: block read from data space: [2]

- **fb_write_data_block**
  subtype: `VCMD__S_FB_FWDB` (=8)
  data: (structure `S_FBCALL`)
  + (1 lword) n = number of data lwords to be written to data space
  + (n lwords) data lwords to be written to data space
  Action: block write to data space

7.8 Treatment of pedestals

- first no command for measurement of pedestals is provided
- pedestals can be loaded down as part of the readout list: DATA=(list of pedestal values)
- KSC F432 (TDC) : pedestals are loaded down as data within readout list and stored within module. The data are passed to FEP after subtraction and zero suppression.
- LRS 18XX (ADC/TDC): these modules do not provide an automatic pedestal subtraction. In order not to increase the deadtime, this will probably *not* be done on the AEB, but might be done by a user supplied routine on the FEP. The pedestal subtraction and data compression takes in the order of 2 μs of CPU time per channel on the AEB. There would be about 200–300 μs time between the SC (start conversion) and SR (start readout) signal from the trigger module. That is not sufficient (only for up to 100 channels).

7.9 Format of Data Transferred to VSB

Data are transferred following the subevent datastructure as described in the VME system description. The data from each FASTBUS module (sub-subevent) are provided with a module header:
(structure `S_FBMOD_HEAD`)

```
typedef struct s_fbmod_head {
    short   i_fbmod_head_id;        /* CSR#0 upper 16 bit         */
    short   c_fbmod_head_geo_addr;  /* geographical address       */
    short   c_fbmod_head_lwords;    /* number of data lwords read */
```

[2]see previous footnote

```
} S_FBMOD_HEAD;
```

structure S_FBMOD_HEAD

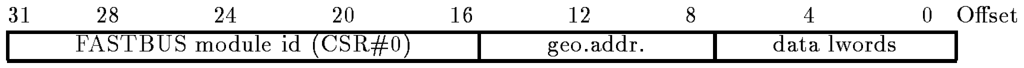

Figure 7.4: Header within listmode data provided for each module

The subevent header is written by the FEP, which collects the sub-subevents from different FASTBUS crates.

7.10 Maintanance

7.10.1 Introduction of a new FASTBUS module type

The following actions must be carried out, if a new FASTBUS module shall be supported:

- add definitions in file **fbmodules.h**
- modify **loadlist.c** to support new module type
- support set of routines for init, clear and readout of new module type (like **STR136.c**). The names for the new routines are
  - `<module type>_init()` : perform init
    1. check and compare FASTBUS module id (CSR#0)
    2. check/set mode of module
    3. clear module (see below)
  - `<module type>_clear()` : clear module
    1. prepare module for next event trigger
  - `<module type>_read()` : read module
    1. read data from module using block transfer
    2. perhaps in a later version: pedestal subtraction

    This readout routines is currently not used for sake of speedup (a subroutine call needs in the order of 20 μs)
- Update command **makelib** to compile also the new routimes written in the previous steps.
- Update subroutine library by invoking command **makelib**.
- If necessary, add definitions for new error codes in **aeb_error.h** (local codes) or **xgoovme.h** (global codes transferred to VAX – definition resides there)

Chapter 8

VMS GOOSY VME Support (VMS)

8.1 VME Listmode Data

8.1.1 Buffer and Event Structures

The VME Event Builder sends data buffers with standard GOOSY format to the VAX. New buffer and event types have to be specified depending on the readout. (see Appendix). The FEPs provide a header to each subevent. The type and subtype are defaulted to standard GOOSY values. The user written FEP routine may change these values. On the EB, the complete event must be provided to a user written event routine. A header is provided for each event with type and subtype defaulted to standard GOOSY values. The user written event routine may change these values. Because of the event spanning one buffer should contain only events of the same type. Therefore, if the user event routine produces several types of events, several buffer types must be supported. An additional buffer counter is needed to count buffers of the same type (specific buffer number). In the EB for each different buffer type there is one current buffer. If this buffer is full it will be sent to the Host (VAX). On the VAX, the Transport Manager ($TMR) may dispatch buffers depending on their type to different output channels.

8.1.2 Event Spanning

Because of the size of the new events it will be necessary to span one event over more buffers. The GOOSY buffer structure supports this spanning. Since infinite buffer spanning would make it impossible to use parallel processing, there should be a maximum number of buffers with fragments. This method works only if all processing programs provide enough buffers to store the maximum number of spanning buffers. These buffers **in order** must be processed by one process. The length of each fragment in words is written in the fragment (event) header. Thus the total event length is the sum of all fragment lengths. In the buffer header information is stored about fragments in the buffer (first event in the buffer is a fragment, last event is a fragment, or both). The event counter in the buffer header contains the number of fragments.

Generally a module working on buffers must return status XIO_CONTBUF to signal the caller that one more buffer is needed. It must store internally (by static variables) any data and information from the last buffer needed for the continuation buffer, because the last buffer is no longer accessible after returning. If the first buffer element is a fragment, the specific buffer number must be checked to prove, that the buffer is the continuation buffer.
NOTE:When buffers of different types contain spanning events, the order of differnet events can be distroyed.
Another method would be to dispatch events or subevents rather than buffers. The advantage is that all buffers may contain spanning events, the disadvantage that the events must be copied. The event spanning concerns the following packages:

Transport Manager

The event/buffer printout routine. At that point all buffers should arrive in order. The printout routines check for fragments. Incomplete fragments can be printed out followed by a message. Filling asynchronous output channels like mailbox or DECnet a strategy must be found to send enough contiguous buffers of the same type containing at least one complete event.

Analysis Manager

The event loop routine I$BUFFER and the unpack routines must be prepared to receive buffers with incomplete fragments. These fragments must be ignored in asynchronous mode. When the last event is a fragment, XIO_CONTBUF must be returned. When the next buffer is the next contiguous buffer of same type, the fragment can be processed, otherwise it must be skipped. If a buffer is not the next contiguous one, a fragment at the beginning must be skipped. In synchronous mode all missing fragments must result in an error. Therefore the Analysis Manager must know the mode.

8.2 GOOSY VME Commands

GOOSY commands for the VME system are implemented similar to the existing commands concerning the J11-based Single CAMAC Crate system. The communication uses however a private ethernet protocol. The message structure is described in appendix A. The command allocates an EMCB, fills it with a command and subcommand number and all required arguments. Then it sends the MCB to the NET processor and waits for completion. The NET processor returns an EMCB with a status (VMS error code).

8.2.1 Transport Manager

Most commands already exist but will be extended to support the VME equipment.

8.2.2 Control Manager

The Control Manager could be a separate process executing control functions for the VME system. Maybe this task could be executed by a workstation.

8.3 Readout Procedures

The readout procedures run either on the FEPs or on the ROPs. In the first implementation the readout procedures are provided by GOOSY and execute tables. These tables are generated on the VAX from description files.

8.3.1 Description File

The current CAMAC description file can be extended to keep all information needed for the more complex crate structure and other bus types. This is the VME crate and station number of ROC or ROP. It should be possible to include other description files by @filename. The modules I$J11TAB and I$J11GEN can be used as base for I$VMETAB generating a readout table and I$VMEGEN generating readout procedures from the decription file. This method can only generate simple standard readout.

8.3.2 User Written Readout Procedures

The second development step is to write macros supporting the reset, readout and reset functions using the language C or ASM. When the syntax of the readout is fixed, step two can be implemented. The first version of a readout procedure could be generated from the description file. It may then be extended by the user.

Event Description Language

For more complex setups and a more complex readout logic a more powerful description language is needed. The EDL package from Giessen could be adapted to the needs of SIS experiments.

8.4 Error Messages

It would be very good, if the error messages reported from the VME system to the VAX would follow the standard error codes. Therefore each processor in VME must have an error code array in the control space where VMS error codes are stored. The programs can use an indexed access to this table to receive VMS message codes and send them to the VAX. Eventually it would be convenient to build only one table containing all error messages on the VAX and load them together with the software into the FEPs and EB. The order of the messages in this table would be holy. This method can be used only if there are not too many error codes needed.

8.5 Transport Manager

The Transport Manager will need some enhanced features.

8.5.1 Buffer Spanning

See above.

8.5.2 Data Dispatching

As mentioned above, it may be necessary for the TMR to dispatch buffers of different type to different output channels. It must be possible to configure these channels. A control structure must be defined and some commands to define, modify and show the switching. This dispatcher should also support several input channels.

Commands

```
OPEN CHANNEL INPUT name source size type-spec.
OPEN CHANNEL OUTPUT name destination /SYNCHRON
MODIFY CHANNEL name
CLOSE CHANNEL name
SHOW CHANNEL name
```

It could be necessary to specify output channels not only on buffer types, but also on event or even subevent types. That means, that one output channel is filled only with events or subevents of a specified type. This costs time to compose the buffers but could reduce the net traffic considerably.

8.6 Analysis Manager

The Analysis Manager will need some enhanced features.

8.6.1 Buffer Spanning

See above.

8.6.2 Data Dispatching

The same mechanism as in the TMR is needed in the Analysis Manager to support several input and output channels. The analysis may generate events of different types and dispatch them into different output channels (sorting).

8.6.3 Event Cruncher and Accumulation Server

A special event format may be useful to keep accumulation statements. When several analysis programs run on different nodes, they calculate new events and send the results back to one analysis doing the accumulations. This analysis executes a kind of dynamic list which must be highly optimized. There is an open problem concerning the execution of conditions. Conditions should be executed on the remote nodes, but then these nodes must run GOOSY or condition checking must be provided by UNIX and C.

8.7 Ethernet Protocol

General description of protocol

PURPOSE	Handle point to point connection via ethernet.

Description

The package of modules is devided into a physical and a logical layer. The physical layer handles sending and acknowledging of buffers (retries) and controlling the link. It splits logical buffers into physical buffers and vice versa. The logical layer is the user interface.

The logical layer defines handlers, opens and closes the link and writes or reads logical buffers of any size. All operations on physical layer are asynchronuous, whereas the logical layer is always synchronuous.

There is a master and a slave. The logical layer modules can be used from both sides in the same way, i.e the package must be initalized and the link must be opened. However, the slave must have opened the link first. Then the master can open the link sending a link request buffer. The slave regards the link down until it gets the open request from the master.

The master openes the link and starts a link checker which sends all n seconds a link control buffer to the slave. The slave must acknowledge the buffer during n seconds. When m control buffers are not acknowledged in order, the master closes the link. The checker interval and the retry number can be set dynamically.

Each buffer must be acknowledged. This is done by sending back a buffer with the same ID and type, but a special subtype. The number of acknowledge buffers to be sent can be modified dynamically. When receiving an acknowledge, the ID is compared to the last acknowledged ID. If it is less or equal it is ignored.

The master may close the link explicitely by sending a close buffer. The slave can close the link simply by not responding anymore to the master.

As mentioned before the logical operations are synchronuous. There are, however, timer for read and write. These timers can be changed dynamically. Reading a buffer of a special type means to wait until a buffer of this type arrives. Writing a buffer means to wait until the acknowledge of the last buffer arrived. Therefore normally the write timer will be much shorter than the read timer.

On logical layer handlers can be specified for buffer types. These handlers are called when a buffer of this type arrives. In a later version the handlers may be queued into a fork queue to execute on non AST level. In this case the buffer must be freed in the handler.

On AST level it is not allowed to call logical read or write routines, because these routines wait for an event flag which is set in another AST routine.

Control

Each physical buffer sent is characterized by 5 numbers:

Type	A type number specified by the sender (logical)
Subtype	A subtype number specified by the sender (logical)
Buffer ID	A buffer ID. This buffer id is incremented for each buffer. It is unique for each buffer, except the link control buffers which have always id=0. The ID is returned in the acknowledge buffer. (physical)
Buffer chain	This gives the number of physical buffers which must be copied into one logical buffer. (physical)
Buffer number	This is the current number of the buffer in a chain. It cannot exceed the chain number. (physical)

Buffertypes

The following buffer types are defined by names in GOOINC($ETHREP)

TH__TYPE_CHECK	Checker buffer type
TH__STYPE_CHECK	Checker buffer subtype
TH__TYPE_OPEN	Open buffer type
TH__STYPE_OPEN	Open buffer subtype
TH__TYPE_CLOSE	Close buffer type
TH__STYPE_CLOSE	Close buffer subtype
TH__STYPE_ACKN	Acknowledge subtype

Example

```
.......
 master
@CALL N$LE_INIT('1'B,8000,'0'B);
@CALL N$LE_OPEN('ETH_DEV','ETH_DEST');
@CALL N$LE_HANDLER(-1,0,ABORT);
@CALL N$LE_HANDLER(0,0,COLLECT);
@CALL N$LE_HANDLER(1,1,INPUT);
@CALL N$LE_TASEND(buffer,1,1,buffer,1,1);
........
.......
 slave
@CALL N$LE_INIT('0'B,8000,'0'B);
@CALL N$LE_OPEN('ETH_DEV','ETH_DEST');
@CALL N$LE_HANDLER(-1,0,ABORT);
@CALL N$LE_HANDLER(0,0,COLLECT);
@CALL N$LE_HANDLER(1,1,INPUT);
@CALL N$LE_READW(buffer,1,1);
@CALL N$LE_WRITEW(buffer,1,1);
  ........
```

Appendix A

VMEbus Mapping Tables (MAP)

A.1 Crate Numbers / Processor Numbers

Each processor is specified by a unique couple of numbers (ID), the crate number and the address mapping of the processor. The address mapping is set by a hardware switch on the processor board specifying the offset. This offset is displayed by LEDs on the front panel (0-F).

```
Processor ID = (crate,offset)
crate        = 1...15
offset       = 0...13
```

Note, however, that processor (1,0) is the event builder and that processor (1,13) cannot be used (no more address window in first crate). Crate numbers start with one, offsets with zero!

A.2 VMEbus Address Map

Crate 1, address range 0 - FFFFFF, A24/D32, Processor ID = (1,0-12) (AMO=Address Mode):

DEVICE	ID		SLOT	AMO	START	END
ISC 8221		RAM,I/O page	1	13,17	C00000	FFFFFF
VME/Q-Bus		Register	2	3B-39	FEFFC0	FF0000
FIC8230	0	1 MByte RAM	3	3B-39	0	FFFFF
FIC8213	1	1 MByte RAM	4	3B-39	100000	1FFFFF
FIC8230	2	1 MByte RAM	5	3B-39	200000	2FFFFF
FIC8230	3	1 MByte RAM	6	3B-39	300000	3FFFFF
FIC8230	4	1 MByte RAM	7	3B-39	400000	4FFFFF
FIC8230	5	1 MByte RAM	8	3B-39	500000	5FFFFF
FIC8230	6	1 MByte RAM	9	3B-39	600000	6FFFFF
FIC8230	7	1 MByte RAM	10	3B-39	700000	7FFFFF

FIC8230 8	1 MByte RAM	11	3B-39	800000	8FFFFF
FIC8230 9	1 MByte RAM	12	3B-39	900000	9FFFFF
FIC8230 10	1 MByte RAM	13	3B-39	A00000	AFFFFF
FIC8230 11	1 MByte RAM	14	3B-39	B00000	BFFFFF
FIC8230 12	1 MByte RAM	15	3B-39	C00000	CFFFFF
VMDIS 8001	VME Dia./Disp.	18			
VBR8213	inter. Register	19	3B-39	DFFE00	DFFFFF
	VMVbus window		39	E00000	EFFFFF
VBR8212		20			

Crate 2-15, address range 0 - FFFFFF,Processor ID = (2-15,0-13) 14 Processors/Crate:

DEVICE		SLOT	AM0	START	END
FIC8230 0	1 MByte RAM	1	3B-39	0	FFFFF
FIC8230 1	1 MByte RAM	2	"	100000	1FFFFF
FIC8230 2	1 MByte RAM	3	"	200000	2FFFFF
FIC8230 3	1 MByte RAM	4	"	300000	3FFFFF
FIC8230 4	1 MByte RAM	5	"	400000	4FFFFF
FIC8230 5	1 MByte RAM	6	"	500000	5FFFFF
FIC8230 6	1 MByte RAM	7	"	600000	6FFFFF
FIC8230 7	1 MByte RAM	8	"	700000	7FFFFF
FIC8230 8	1 MByte RAM	9	"	800000	8FFFFF
FIC8230 9	1 MByte RAM	10	"	900000	9FFFFF
FIC8230 10	1 MByte RAM	11	"	A00000	AFFFFF
FIC8230 11	1 MByte RAM	12	"	B00000	BFFFFF
FIC8230 12	1 MByte RAM	13	"	C00000	CFFFFF
FIC8230 13	1 MByte RAM	14	"	F00000	FFFFFF
VMDIS 8001	VME dis./dia.	18			
VBE8213	Register	19	"	DFFE00	DFFFFF
	VMVbus window		"	E00000	EFFFFF
VBR8212		20			

Appendix B

System Data Structures (SYSDAT)

B.1 NET Processor Control Space

To be done

B.2 EB and FEP Control Space

The control spaces for EB and FEP are identical. They are used like global data bases which held all informations about the system status and all parameters which are necessary for the system control and system configuration. The control space is accessible by all processor internal processes as well as via VME from other processors. It is located in the memory region $0x20002020$—$0x20002FFF$. Description of the System Control Space parameters:

b_fic_load is set to zero during the processor power up sequence. The startup procedure polls on the load flags, which are set by the NET processor if the system routines are loaded. Futhermore flags are defined in $LOADREP which are set if the CAMAC or FASTBUS readout lists or the user routines have been loaded. This flags are set by the processor itself! Valid load bits are: FIC__PROG_LOAD, FIC__CAMAC_INIT, FIC__CAMAC_READ, FIC__CAMAC_RESET, FIC__FASTBUS_LIST, FIC__USER_READOUT, FIC__USER_ANAL, FIC__USER_PROC.

l_fic_root_start is the start address of the system root process. This address has to be set by the NET processor after the completion of the system load. If the processor power up procedure recognizes the LOAD flag it directly jumps to the ROOT_START address.

b_fic_blocked can be used to block system resources or to block the access of other system components to local resources. Not yet used.

l_fic_proc_id is the value defined in the VME description file for each FIC 8230 procssor.

l_fic_proc_type is the value defined in the VME description file for each FIC 8230 procssor.

l_fic_proc_offset is the value of the rotary switch defining the master number of each FIC 8230 processor. It is read by the processor itself from the VME Status register.

l_fic_proc_crate is the VMV card cage number (crate number). It can not be determined by the processor, it has to be set by the NET processor during the system load.

l_fic_proc_index is the running number for each FIC 8230 procssor.

l_fic_proc_feps is the total number of FEP's. It is used by the Event builder.

b_fic_cpu_status should be used to keep global status informations about the processor; e.g what processes are active, if the CPU is still running, etc. Not yet used.

b_fic_event_status contains the flags describing the status of the listmode processes. The flags are defined in $STATUS:

EB__FEP_WAIT The EB waits for the announcement of the next subevent.

EB__FEP_SCAN The EB performs the FEP subevent status scan.

EB__USER_EVTANAL The user event analysis is active in the EB.

FEP__WAIT_VSB The FEP waits for a new VSB interrupt.

FEP__LIST_READOUT The standard list readout is active.

FEP__USER_READOUT A user written readout procedure is active.

FEP__USER_SUBANAL The FEP user subevent analysis is active.

FIC__WAIT_BUFFER The listmode task waits for a buffer from the buffer manager.

FIC__MSG_WAIT The listmode process waits for the message buffer to send an error message to the host.

FIC__TIMEOUT The timeout condition occured.

FIC__SUSPEND The process has been suspended due to an error.

b_fep_subevt_status contains the status of the actually processed event. This status is used only by the FEP and is announced to the EB. Valid values are:

FIC__VALID_DATA subevent contains valid data.

FIC__SKIP_SUBEVENT1···4 the subevent can be skipped due to any hardware or software descision. The standard readout uses only FIC__SKIP_SUBEVENT1. A user readout routine or subevent analysis can uses the other values to characterize the trigger condition. item[FIC__SKIP_EVENT···4] the whole event can be skipped due to any hardware or software descision. The standard readout uses only FIC__SKIP_SUBEVENT1. A user readout routine or subevent analysis can uses the other values to characterize the trigger condition.

FIC__ERROR_DATA An error occured in the subevent processing

b_fic_transfer_status contains the flags describing the status of the listmode data transfer. This status word can be used by the processor to decide if an initiated transfer is still active, finished with success or terminated with an error. The transfer status can be set directly by the processor which controls the listmode output channel (e.g the NET) or in the case of interrupt synchronized output channels (e.g DMA transfer to the Host (VAX)) by an ISP. The flags are defined in $STATUS:

- **EB__TRANS_INITIATED** The EB initiated the buffer transfer into the output channel.
- **FIC__TRANS_SUCCESS** The transfer finished succesfully.
- **FIC__TRANS_ERROR** The transfer finished with an error condition.

b_fic_message_status contains the flags describing the status of the message data transfer. The status of the message transfer completion has to be set by an ISP! The processor which receives the message writes the status into the FIFO 1 register! The triggered ISP handles the message synchronization and copies the status found in the FIFO register into the message status word. The flags are defined in $STATUS:

- **FIC__MSG_INITIATED** The message transfer is initiated.
- **FIC__MSG_ACTIVE** The message transfer is active.
- **FIC__MSG_SUCCESS** The message transfer finished successfully.
- **FIC__MSG_TIMEOUT** The message has not been transfer within the timeout.
- **FIC__MSG_ABORT** The message link to the host has been aborted.

b_fic_command_status contains the flags describing the status of the command execution. These flags are used only by the local CPU. The synchronization with the NET processor occurs with interrupts. The flags are defined in $STATUS:

- **FIC__CMD_WAIT** The command process waits for a command.
- **FIC__CMD_USER** The command is a user specific command.
- **FIC__CMD_LM** The command is executed in the listmode process
- **FIC__CMD_RESULT** The command process send back the result of command execution.

b_fic_user_status flags to describe the current status of the user process. The upper 16–bit word flags are allowed to be set by the user itself.

l_fic_debug This value can be set by command to 0,1,2. Depending on the level, the output of the FIC's is controlled.

- **0** no output.
- **1** informational output.
- **0** debug output.

l_fep_init_list is the address of the front end initialization list. Only used by the FEPs.

l_fep_readout_list is the address of the front end readout list. Only used by the FEPs.

l_fep_reset_list is the address of the front end reset list. Only used by the FEPs.

l_fic_process_id[6] keeps the pSOS process IDs of all spawned processes.

l_fic_process_links[6] is the address of the process link lists, which are used to generate communication links with other processes in the same CPU or to any process running in an other system processor. Not yet used.

l_fic_event_counter is the subevent or event number in the corresponding buffers.

l_eb_buffer_counter is the buffer counter for the final EB–output buffers.

l_fic_fep_buffer has different meanings in the FEP and EB. In the FEP it keeps the buffer queue index of the actually used subevent buffer. In the EB it keeps the subevent index which is actually read by the subevent collector.

l_fic_eb_buffer keeps the buffer queue index of the actually used event buffer. It is used only by the EB!

l_fic_receive_msg is the address of the message/command buffer into which the NET processor is allowed to copy the next received message/command buffer.

l_fic_send_msg is the address of the message buffer which should be sent to the Host (VAX).

b_fic_msg_input_mode determines the channel and mode for the message and command input. The flags are defined in $MODEREP.

b_fic_msg_output_mode determines the channel and mode for the message and command output. The flags are defined in $MODEREP.

l_fic_msg_input_source address of message input buffer

l_fic_msg_output_dest address of message output buffer

b_fic_lm_receive is set if listmode data has been received on the listmode data input channel. The subevent readout tasks in the FEPs poll on that flag which is set by the VSB interrupt service routine. If this flag has been recognized the subevent readout starts. In the EB this slot is not yet used.

b_fic_lm_msg_flag should be set if a command has been received which should be executed in the listmode task (e.g. the download of user parameters). If the flag is set the command is executed in the listmode task. Attention, all commands are received by the command handling task. This tasks decides which command have to be executed in the listmode process.

l_fic_lm_user_buffer is the address of a buffer to held temporary subevent/event data. It is not directly accessible by the user!

b_fic_lm_input_mode keeps the information about the active listmode data input channel and the mode in which it works. The flags are defined in $MODEREP:

- **FEP__IN_ROC** FEP input channel is the CAMAC readout controller processed in standard mode.
- **FEP__IN_ROCFAST** the CAMAC readout controller is processed in thge fast mode, which ignores the execution codes.
- **FEP__IN_ROP** The FEP input channel is the CAMAC readout processor.
- **FEP__IN_FASTBUS** The FEP input channel is the interface to the Aleph Event Builder.
- **FEP__IN_FADC** The FEP input channel is the Heidelberg FADC–System.
- **FEP__IN_TEST** The test input channel is active.
- **EB__IN_FEP** The standard FEP subevent collector is active.
- **EB__IN_CAOTIC** The FEP subevent collection is performed in the asynchronous mode.
- **EB__IN_TEST** The test input channel is active.

The EB input channels can work in the following modes, which are coded in the upper 16–bit word of the whole status longword.

- **EB__FEP_NORMAL** The input channle mod is set to the normal mode.
- **EB__FEP_IGNORE** All FEP's with subevent errors are ignored, no message will be send to the host, as done in the standard mode.
- **EB__FEP_COPY** The subevents are not copied directly into the final event buffer. First they are copied contigious into the EB–memory, to get user routines the possibility to work on the complete event.

b_fic_lm_output_mode keeps the information about the active listmode data output channel and the mode in which it works. The flags are defined in $MODEREP:

- **FEP__OUT_EB** Output to Event Builder.
- **EB__OUT_NET** EB output channel is the NET processor.
- **EB__OUT_CES** The EB–buffer transfer occurs via the CES–parallel interface.
- **EB__OUT_SOFT** Output to a software trigger system.

l_fic_lm_input_source is the address at which the listmode data input buffer can be found. This address is only used for special input channels, e.g. the test input.

l_fic_lm_output_dest is the address where the listmode data output buffer can be written to. In the FEP this slot keeps the base address of the FEP–map segment in the EB memory!

l_fic_queue is the length of the internal buffer queue. **For all FEPs the internal buffer queue length has to be identical!**

l_fic_cluster_size determines the buffer cluster number. In the EB the event buffers can be divided into different clusters, each cluster has it own buffer type and subtype. The user can determine which cluster should be used for the actually processed event. This allows a simple dispatching to different analysis at the VAX. Not yet implemented.

l_fic_timeout is the timeout counter if a wait for any system resources is necessary. It is the time in units of $10\mu sec$.

l_fic_buf_manager is the address of the buffer manager structure.

l_fep_buf_queue is the length of the buffer queue in each FEP. This slot is used only by the EB in its subevent collector, this length has to be the same as in the l_fic_queue slot of all FEPs!

l_eb_max_spanning maximum number of EB–output buffers, which are allowed for spanning. If this value is reached the buffer is not filled completely and no fragment will be at the end of the buffer.

l_fic_max_bufsize default size reserved for each buffer.

l_fic_act_bufsize actually used buffer size. It is possible to reduce the length of the internal buffer queue and to increase the buffer size.

l_fic_buffer_adress start address of the buffer segment.

l_eb_fep_list start address of the FEP–map list. This is used by EB only.

l_fic_exception_stack address at which the exception stack has been copied during the exception processing.

b_fic_user_enable flags to activate the loaded user routines. The flags are defined in $LOAD-REP.

l_fic_user_buffer address at which a copy of an event or subevent exists, which can be processed by a separate user analysis process.

b_fic_user_lock lock flag for the user subevent and event buffer. The flag is set by the listmode task of the EB or FEP and has to be cleared by the user process if the buffer has been processed.

l_fic_user_counter counter for the number of buffers passed to the user process.

b_fep_crates_online Flags which are set for each CAMAC or FASTBUS crate which is found to be online.

b_fep_crates_readout flags for each front–end crate, which should be included in the readout.

l_fic_lmd_counter counter for the number of commands executed in the listmode task.

l_fic_cmd_counter counter for the number of commands executed in the command process.

l_fic_msg_counter number of messages send by the message sender.

l_fic_event_statistic[12] counter to keep up to 12 statistical informations about the subevent status in the FEP or the event status in the EB. In the FEP the user routine is allowed to return with the codes defined in $STATUS, which are used to increment the corresponding counters.

l_fic_cmd_statistic[12] counter to keep statistical informations about the executed command types, which are described in $BUFFER_TYPE.

l_buffer_counter counter for buffer/sec.

B.3 CAMAC, Fastbus, VME Readout Tables

The tables used for the initialization, readout and reset of the CAMAC, Fastbus, or VME crates are formatted like GOOSY buffer elements, i.e. they are preceded by a standard buffer element header. There are three types (CAMAC, Fastbus and VME) and three subtypes (Init, Readout, Reset).

B.3.1 CAMAC List Format

CAMAC Initialization

This table is used to initialize CAMAC modules. The structure is mapped by SI$FEPLIST in GOOINC(SI$VMELIST). The data longword is optional depending on the CAMAC function.

Event Type 2001, Subtype 1

31 28	24 20	16 12	8 4 0	Offset
Data length [words]				0
Subtype = 1		Type = 2001		4
controller	Subcrate	Processor offset	Processor crate	8
Trigger number				12
1st CAMAC CNAF				16
1st repetition counter				20
1st CAMAC data longword (optional)				24
2nd CAMAC CNAF				28
...				

Figure B.1: CAMAC initialization table

```
DCL 1 SI$FEPLIST BASED(P_SI$FEPLIST),
    2 LI$FEPLIST_len BIN FIXED(31),
    2 II$FEPLIST_type BIN FIXED(15),
    2 II$FEPLIST_subtype BIN FIXED(15),
    2 HI$FEPLIST_procrate BIN FIXED(7),
    2 HI$FEPLIST_procoff BIN FIXED(7),
    2 HI$FEPLIST_subcrate BIN FIXED(7),
    2 HI$FEPLIST_control BIN FIXED(7),
    2 LI$FEPLIST_trigger BIN FIXED(31),
    2 LAI$FEPLIST_data(LI$FEPLIST_len/2-2) BIN FIXED(31),
    2 LI$FEPLIST_next BIN FIXED(31);
```

Front End Processor ID	Processor number calculated from the crate and address mapping of the FEP. See Appendix A.
CAMAC CNAF	Longword containing encoded CAMAC function.

CAMAC CNAF

31	28	24	20		16	12	8	4	0	Offset
–	E	C	O	P	R	N	F	A	–	0

E   Bit 25-28: Execution code telling the FEP how to execute the CAMAC function. Valid values:

```
E = 0000 : execute only
E = 0001 : execute and signal error if no X-response
E = 0010 : execute and signal error if no Q-response
E = 0100 : execute until Q=1
E = 1000 : execute until Q=0
E = 0011 : execute and signal error if no X- and Q-response
E = 0101 : execute and check X-response until Q=1
E = 1001 : execute and check X-response until Q=0
```

C   Bit 21-24: Crate number (1-15)

O   Bit 19-20: CVI Memory Offset

P   Bit 17-18: not used

R   Bit 16: CVI fast readout bit

N   Bit 11-15: Station number (1-23)

A   Bit 2-5: Subaddress (0-15)

F   Bit 6-10: Function code (0-31). The codes between 16 and 22 need the CAMAC data longword, codes between 0 and 15 do not.

The longword is matched by a structure SI$CNAF in library GOOINC:

```
/* Numbers from 1 to 32. Arguments for PL/1 POSINT function: */
%REPLACE CNAF__A BY 3;  %REPLACE CNAF__A_L BY 4;
%REPLACE CNAF__F BY 7;  %REPLACE CNAF__F_L BY 5;
%REPLACE CNAF__N BY 12; %REPLACE CNAF__N_L BY 5;
%REPLACE CNAF__R BY 17; %REPLACE CNAF__R_L BY 1;
%REPLACE CNAF__P BY 18; %REPLACE CNAF__P_L BY 2;
%REPLACE CNAF__O BY 20; %REPLACE CNAF__O_L BY 2;
%REPLACE CNAF__C BY 22; %REPLACE CNAF__C_L BY 4;
%REPLACE CNAF__E BY 26; %REPLACE CNAF__E_L BY 4;
DCL P_SI$CNAF POINTER;
DCL 1 SI$CNAF BASED(P_SI$CNAF),
    2 BI$CNAF_X BIT(2), /* not used   */
    2 BI$CNAF_A BIT(4), /* subaddress */
    2 BI$CNAF_F BIT(5), /* function   */
```

```
2 BI$CNAF_N BIT(5), /* station    */
2 BI$CNAF_R BIT(1), /* RAM bit =1 */
2 BI$CNAF_P BIT(2), /* not used   */
2 BI$CNAF_O BIT(2), /* Offset  =1 */
2 BI$CNAF_C BIT(4), /* crate      */
2 BI$CNAF_E BIT(4), /* execution  */
2 BI$CNAF_Z BIT(3); /* not used   */
```

Repetition Counter tells the FEP how often it has to execute this function according the execution bits.

CAMAC data contains the 24 bit CAMAC data. The data can only be written, therefore only the write-functions require a data word. Write-functions are the function having F-codes in the range from 16 - 22. If no write function is required, the data word must be omitted.

CAMAC Readout

The structure is mapped by SI$FEPLIST in GOOINC(SI$VMELIST). The readout lists (subtype 2) contains coded CAMAC addresses followed by a longword for the repetition counter (Fig. B.2).

Event Type 2001, Subtype 2

31 28 24	20 16	12 8	4 0	Offset
Data length [words]				0
Subtype = 2		Type = 2001		4
controller	Subcrate	Processor offset	Processor crate	8
Trigger number				12
1st CAMAC CNAF				16
1st repetition counter				20
2nd CAMAC CNAF				24
...				

Figure B.2: CAMAC readout table

Front End Processor ID Processor number calculated from the crate and address mapping of the FEP. See Appendix A.

CAMAC CNAF Longword containing encoded CAMAC function.

Repetition Counter tells the FEP how often it has to execute this function according the execution bits.

CAMAC Reset

The reset lists (subtype 3) contains coded CAMAC addresses followed by a longword for the repetition counter (Fig. B.3). The structure is mapped by SI$FEPLIST in GOOINC(SI$VMELIST).

Event Type 2001, Subtype 3

31 28	24 20	16 12	8 4 0	Offset
Data length [words]				0
Subtype = 3		Type = 2001		4
controller	Subcrate	Processor offset	Processor crate	8
Trigger number				12
1st CAMAC CNAF				16
1st repetition counter				20
2nd CAMAC CNAF				24
. . .				

Figure B.3: CAMAC reset table

Front End Processor ID Processor number calculated from the crate and address mapping of the FEP. See Appendix A.

CAMAC CNAF Longword containing encoded CAMAC function.

Repetition Counter tells the FEP how often it has to execute this function according the execution bits.

B.3.2 Fastbus List Format

There is only one Fastbus list subtype (subtype = 1) (Fig. B.4). This list contains a type code for the supported Fastbus devices. The structure is mapped by SI$FEPLIST in GOOINC(SI$VMELIST). The module type structure is mapped by SI$FBMOD in GOOINC(SI$VMELIST).

```
DCL 1 SI$FBMOD BASED(P_SI$FBMOD),
    2 II$FBMOD_type BIN FIXED(15),
    2 II$FBMOD_addr BIN FIXED(15),
    2 II$FBMOD_len BIN FIXED(15),
    2 II$FBMOD_chan BIN FIXED(15),
    2 LAI$FBMOD_data(II$FBMOD_len) BIN FIXED(31),
    2 LI$FBMOD_next BIN FIXED(31);
```

Geogr.Address Fastbus module geographical address (3-24)

Module type code Type of module (1-4):

Event Type 2002, Subtype 1

31 28 24	20 16	12 8	4 0	Offset
Data length [words]				0
Subtype = 1		Type = 2002		4
not used	Subcrate	Processor offset	Processor crate	8
Trigger number				12
Geogr.Address		Module type code		16
Active channels		Data length [longwords]		20
1st Data longword				24
2nd Data longword				28
...				

Figure B.4: Fastbus table

- Type 1 : LRS 1800 series ADCs/TDCs
- Type 2 : KSC F432 TDCs
- Type 3 : STR 136 ECL input/output latch
- Type 4 : STR 200 32 channel 100 MHz counter

Active channels	Number of channels per module (1-96)
Data length	Number of words following.
Data longword	Depends on module type, e.g. pedestal values.

B.3.3 VME Module List Format

This list does only have the purpose to check the existence of the specified modules. It will not be sent to any FEP but executed from the ISC8221 only. Therefore there is only one VMEbus address and one dataword containing a value specific to this module to make sure that the module is plugged in and that it is in the proper state to run. The list format is shown in figure B.5. The structure is declared as SI$VMEMOD in GOOINC(SI$VMELIST).

```
DCL 1 SI$VMEMOD BASED(P_SI$VMEMOD),
    2 LI$VMEMOD_len BIN FIXED(31),
    2 II$VMEMOD_type BIN FIXED(15),
    2 II$VMEMOD_subtype BIN FIXED(15),
    2 LI$VMEMOD_trigger BIN FIXED(31),
    2 SI$VMEMOD_item((LI$VMEMOD_len-2)/6),
    3 LI$VMEMOD_crate BIN FIXED(31),
    3 LI$VMEMOD_addr BIN FIXED(31),
    3 LI$VMEMOD_data BIN FIXED(31),
    2 LI$VMEMOD_next BIN FIXED(31);
```

Event Type 2003, Subtype 1

31 28 24 20 16	12 8 4 0	Offset
Data length [words]		0
Subtype = 1	Type = 2003	4
Trigger number		12
1st module's VMV crate number		16
1st VME address (A24)		20
1st Data longword		24
2nd module's VMV crate number		28
2nd VME address (A24)		32
2nd Data longword		36
...		

Figure B.5: VME table

VMV crate	VMV crate number (1-15)
VME address	VMEbus address 0-FFFFFF (hex), specific for module.
Data Longword	Specific for module.

B.4 Event structures

B.4.1 Event structure

This structure is composed by the EB. It is mapped by SA$VE10_1 in library GOOINC.

Event Type 10, Subtype 1

31 28 24 20 16	12 8 4 0	Offset
Data length [words]		0
Subtype = 1	Type = 10	4
Not used	Trigger	8
Event counter		12
Subevent 1		16
...		
Subevent n		16+x

Figure B.6: Event Structure

```
/* ================= GSI VME Event header ====================== */
DCL P_SA$ve10_1       POINTER;
DCL 1 SA$ve10_1         BASED(P_SA$ve10_1),
    2 LA$ve10_1_dlen    BIN FIXED(31),
    2 IA$ve10_1_type    BIN FIXED(15),
    2 IA$ve10_1_subtype BIN FIXED(15),
    2 IA$ve10_1_dummy   BIN FIXED(15),
    2 IA$ve10_1_trigger BIN FIXED(15),
    2 LA$ve10_1_count   BIN FIXED(31),
    2 IA$ve10_1(LA$ve10_1_dlen-4)    BIN FIXED(15),
    2 LA$ve10_1_next    BIN FIXED(31);
/*------------------------------------------------------------------*/
```

B.4.2 CAMAC Subevent Structure

This subevent structure is written by the ROP or the FEP. It is defined in SA$VES10_1 in library GOOINC.

```
/* ================= GSI VME Subevent header ====================== */
DCL P_SA$ves10_1       POINTER;
DCL 1 SA$ves10_1         BASED(P_SA$ves10_1),
    2 LA$ves10_1_dlen    BIN FIXED(31),
    2 IA$ves10_1_type    BIN FIXED(15),
    2 IA$ves10_1_subtype BIN FIXED(15),
```

Subevent Type 10, Subtype 1

31 28 24	20 16	12 8 4 0	Offset
Subevent Data length [words]			0
Subevent subtype = 1		Subevent type = 10	4
Control	subcrate	Processor ID	8
CAMAC value		CAMAC module ID	12
...			

Figure B.7: CAMAC Subevent Structure

```
    2 IA$ves10_1_procid  BIN FIXED(15),
    2 HA$ves10_1_subcrate BIN FIXED(7),
    2 HA$ves10_1_control BIN FIXED(7),
    2 IA$ves10_1(LA$ves10_1_dlen-2)    BIN FIXED(15),
    2 LA$ves10_1_next    BIN FIXED(31);
/*-----------------------------------------------------------------*/
```

B.4.3 AEB Subevent Structure

This subevents are written from the AEB. The header structure is defined in SA$VES10_1 in library GOOINC.

Subevent Type 10, Subtype 2

31 28 24	20 16	12 8 4 0	Offset
Subevent Data length [words]			0
Subevent subtype = 2		Subevent type = 10	4
Control	subcrate	Processor ID	8
Fastbus module header			16
...			

Figure B.8: Fastbus Subevent Structure

The following structure maps to the data field. It is defined in SA$vesfb in library GOOINC.

```
/* Fastbus module header maps to IA$ves10_2(i) */
DCL P_SA$vesfb       POINTER;
DCL 1 SA$vesfb BASED(P_SA$ves_fb),
    2 IA$vesfb_id BIN FIXED(15),
    2 HA$vesfb_addr BIN FIXED(7),
```

Fastbus module header

31 28 24	20 16	12 8 4 0	Offset
Longwords	Geo.addr.	Module ID	0
1st data word			4
. . .			

Figure B.9: Fastbus Module header

```
    2 HA$vesfb_lwords BIN FIXED(7),
    2 LA$vesfb_data(HA$vesfb_lwords)    BIN FIXED(31),
    2 LA$vesfb_next    BIN FIXED(31);
```

One data word looks like

Fastbus data word

31 28	24		20	16	12 8 4 0	Offset
Geo.addr.	Event	R	Channels	Dummy	Data Word	0

Figure B.10: Fastbus Data Word

```
/* Structure of data words */
/* Numbers from 1 to 32 can be used in POSINT */
%REPLACE FBDATA_d    BY 1;  %REPLACE FBDATA_d_l  BY 12;
%REPLACE FBDATA_x    BY 13; %REPLACE FBDATA_x_l  BY 4;
%REPLACE FBDATA_ch   BY 17; %REPLACE FBDATA_ch_l BY 7;
%REPLACE FBDATA_r    BY 24; %REPLACE FBDATA_r_l  BY 1;
%REPLACE FBDATA_ev   BY 25; %REPLACE FBDATA_ev_l BY 3;
%REPLACE FBDATA_ad   BY 28; %REPLACE FBDATA_ad_l BY 5;
DCL P_SI$FBDATA POINTER; /* maps to LA$vesfb_data(i) */
DCL 1 SI$FBDATA BASED(P_SI$FBDATA),
    2 BI$FBDATA_d  BIT(12) /* data word */
    2 BI$FBDATA_x  BIT(4), /* dummy     */
    2 BI$FBDATA_ch BIT(7), /* channel   */
    2 BI$FBDATA_r  BIT(1), /* range     */
    2 BI$FBDATA_ev BIT(3), /* event     */
    2 BI$FBDATA_ad BIT(5); /* geo addr. */
/*-------------------------------------------------------------------*/
```

B.4.4 Scaler Subevent Structure

This structure is in local memory of the FEP. It may be filled by user subevent routines. It is read by the EB any n events, where n can be specified by command. The interpretation of the array is up to the user.

Subevent Type 11, Subtype 1

31 28 24 20 16	12 8 4 0	Offset
Subevent Data length [words]		0
Subevent subtype = 1	Subevent type = 11	4
1st value		8
...		

Figure B.11: Scaler Subevent Structure

B.4.5 Error Subevent Structure

This subevent is written from the FEP. It is defined in SA$VESERR in library GOOINC. The error subevent is written behind the data subevent which is erraneous. The type of the erraneous data subevent is negative indicating that an error subevent follows.

Subevent Type, Subtype

31 28 24	20 16	12 8 4 0	Offset
Subevent Data length [words]			0
Subevent subtype		Subevent type	4
Control	subcrate	Processor ID	8
Error code			12
...			
Total length in words			n

Figure B.12: Error Subevent Structure

```
/* ================= GSI VME Subevent error header ================== */
DCL P_SA$veserr       POINTER;
DCL 1 SA$veserr         BASED(P_SA$veserr),
    2 LA$veserr_dlen    BIN FIXED(31),
    2 IA$veserr_type    BIN FIXED(15),
    2 IA$veserr_subtype BIN FIXED(15),
    2 IA$veserr_procid  BIN FIXED(15),
    2 HA$veserr_subcrate BIN FIXED(7),
2 HA$veserr_control BIN FIXED(7),
2 LA$veserr_code    BIN FIXED(31),
2 IA$veserr(LA$veserr_dlen-2)    BIN FIXED(15),
2 LA$veserr_wlen    BIN FIXED(31),
2 LA$veserr_next    BIN FIXED
```

B.5 Message Buffer Format

This format is used for the communication between VAX and VME control processor. It maps into the data field of a GOOSY message control block SN$EMCB. The structure maps over a command header for one FEP (SN$EMCB_CMD). The structure is defined in SN$EMCB_MSG

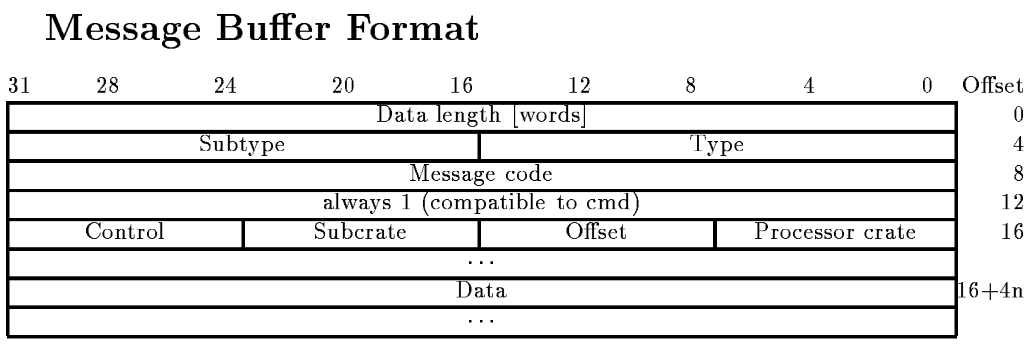

Figure B.13: Message Buffer

in library GOOINC:

```
/* Generated from GOOVME(SN$EMCB_MSG)  */
  /* ==== message buffer for VME-TMR communication ============= */
DCL P_SN$emcb_msg POINTER ;
DCL 1 SN$emcb_msg BASED(P_SN$emcb_msg),
    2 LN$emcb_msg_dlen BIN FIXED(31)  /*  Length of data + 6 words */,
    2 IN$emcb_msg_type BIN FIXED(15),
    2 IN$emcb_msg_subtype BIN FIXED(15),
    2 LN$emcb_msg_code BIN FIXED(31)  /*  message code  */,
    2 LN$emcb_msg_feps BIN FIXED(31)  /*  always = 1    */,
    2 HN$emcb_msg_procrate BIN FIXED(7)  /*  VME crate number */,
    2 HN$emcb_msg_procoff BIN FIXED(7)  /*  Processor address offset */,
    2 HN$emcb_msg_subcrate BIN FIXED(7)  /*  Subcrate number */,
    2 HN$emcb_msg_control BIN FIXED(7)  /*  Processor type code */,
    2 IN$emcb_msg_data(LN$EMCB_MSG_dlen-6) BIN FIXED(15);
```

The message codes are defined in GOO$MSG:XVME.MSG. With command MESDEF include files for C programs are generated.

B.6 Command Buffer Format

This format is used for the communication between VAX and VME control processor. It maps into the data field of a GOOSY message control block SN$EMCB. The structure is defined in

Command Buffer Format

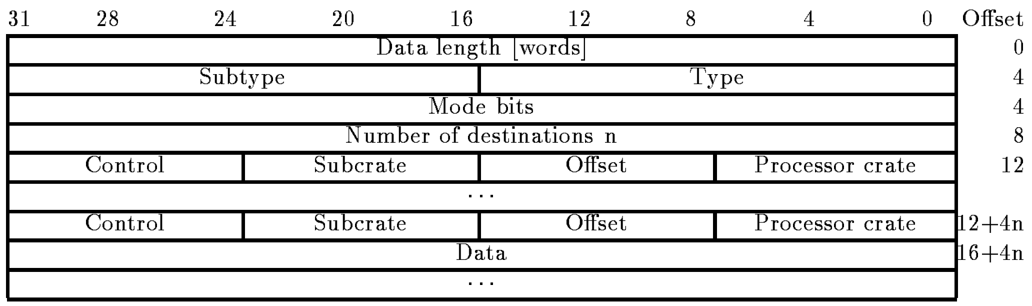

Figure B.14: Command buffer

SN$EMCB_CMD in library GOOINC:

```
/* Generated from GOOVME(SN$EMCB_CMD)  */
  /* ==== command buffer for VME-TMR communication ============ */
DCL L_SN$EMCB_cmd_feps BIN FIXED(31);
DCL P_SN$emcb_cmd POINTER ;
DCL 1 SN$emcb_cmd BASED(P_SN$emcb_cmd),
    2 LN$emcb_cmd_dlen BIN FIXED(31)  /*  Length of data field in words */,
    2 IN$emcb_cmd_type BIN FIXED(15),
    2 IN$emcb_cmd_subtype BIN FIXED(15),
    2 BN$emcb_cmd_mode BIT(32) ALIGNED  /*  Mode pattern */,
    2 LN$emcb_cmd_feps BIN FIXED(31)  /*  Number of FEP's */,
    2 SN$emcb_cmd_fep(L_SN$EMCB_cmd_feps REFER(LN$EMCB_cmd_feps)) ,
    3 HN$emcb_cmd_procrate BIN FIXED(7)  /*  VME crate number */,
    3 HN$emcb_cmd_procoff BIN FIXED(7)  /*  Processor address offset */,
    3 HN$emcb_cmd_subcrate BIN FIXED(7)  /*  Subcrate number */,
    3 HN$emcb_cmd_control BIN FIXED(7)  /*  Processor type code */,
    2 LN$emcb_cmd_data(1) BIN FIXED(31)  /*  Data */;
/*------------------------------------------------------------*/
```

The command codes and modes are defined in $VCMDREP in library GOOINC.

B.7 Ethernet Data Structures

B.7.1 Message Control Block

This format is used for the ethernet communication between VAX and VME control processor.

Message Control Block

31 28 24	20 16	12 8	4 0	Offset
Forward pointer (local)				0
Back pointer (local)				4
Used words in data field				12
Message subtype		Message type		16
Reply subtype		Reply type		20
Not used		User ID		16
Size of data field in Bytes (local)				8
Message ID				24
Number of chained buffers				28
Number of buffer in chain				32
Mode bits				36
Ethernet Status (local)				40
Ethernet Status (local)				44
Data Byte 4	Data Byte 3	Data Byte 2	Data Byte 1	48
...				

Figure B.15: Message buffer

The structure is defined in SN$EMCB in library GOOINC:

```
/* Generated from GOOVME(SN$EMCB)  */
  /*  GOOSY Ethernet message structure  */

DCL L_HN$emcb_data BIN FIXED(31);

DCL P_SN$emcb POINTER ;
DCL 1 SN$emcb BASED(P_SN$emcb),
    2 SN$emcb_header ,
    3 PN$emcb_queue(2) POINTER  /*  queue link  */,
    3 LN$emcb_data_size BIN FIXED(31)  /*  Real message size words  */,
    3 IN$emcb_type BIN FIXED(15)  /*  Message type  */,
    3 IN$emcb_subtype BIN FIXED(15)  /*  Message sub-type  */,
    3 IN$emcb_rep_type BIN FIXED(15)  /*  Message type for reply  */,
    3 IN$emcb_rep_subtype BIN FIXED(15)  /*  Message sub-type for reply  */,
    3 IN$emcb_user_id BIN FIXED(15)  /*  VMS user group number   */,
    3 IN$emcb_dummy BIN FIXED(15)  /*  not used   */,
    3 LN$emcb_alloc_size BIN FIXED(31)  /*  Allocation size  */,
    3 LN$emcb_write_id BIN FIXED(31)  /*  Transaction number  */,
    3 LN$emcb_nobuf BIN FIXED(31)  /*  Number of buffers in set  */,
    3 LN$emcb_curbuf BIN FIXED(31)  /*  Current buffer number in set  */,
    3 BN$emcb_mode BIT(32) ALIGNED  /*  Flags  */,
    3 IN$emcb_iosb_stat BIN FIXED(15),
    3 IN$emcb_iosb_size BIN FIXED(15),
    3 HN$emcb_iosb_dummy1 BIN FIXED(7),
    3 HN$emcb_iosb_line_stat BIN FIXED(7),
    3 HN$emcb_iosb_error BIN FIXED(7),
    3 HN$emcb_iosb_dummy2 BIN FIXED(7),
    2 HN$emcb_data(L_HN$emcb_data REFER(LN$emcb_alloc_size)) BIN FIXED(7);
```

B.7.2 Ethernet Data Base

The ethernet data base for the VAX side is in GOOINC(SN$BASE):

```
/* Local control structure for GOOSY Ethernet */

DCL 1 SN$BASE STATIC,

/* Control bits: */
  2 BN$BASE_link_open   BIT(1) ALIGNED,    /* Link is open */
  2 BN$BASE_link_check  BIT(1) AlIGNED,    /* cleared by checker, set by ackn */
  2 BN$BASE_write_ackn  BIT(1) ALIGNED,    /* cleared by write, set by ackn */
  2 BN$BASE_read_ackn   BIT(1) ALIGNED,    /* cleared by read, set by read AST */
  2 BN$BASE_link_master BIT(1) ALIGNED,    /* Master bit */
  2 BN$BASE_local       BIT(1) ALIGNED,    /* If set, use mailbox */
  2 BN$BASE_swap        BIT(1) ALIGNED,    /* If set, do byte swap */
  2 BN$BASE_stream_i    BIT(1) ALIGNED,    /* If set, read stream input */
  2 BN$BASE_stream_o    BIT(1) ALIGNED,    /* If set, write stream output */
  2 BN$BASE_debug       BIT(1) ALIGNED,    /* Output debug information */
  2 BN$BASE_dummy2      BIT(1) ALIGNED,    /* */
  2 BN$BASE_dummy3      BIT(1) ALIGNED,    /* */
  2 SN$BASE_IOSB UNION,                    /* Status for ETH routines */
    3 SN$BASE_IOSB_1,
     4 IN$BASE_IOSB_stat      BIN FIXED(15),
     4 IN$BASE_IOSB_size      BIN FIXED(15),
     4 HN$BASE_IOSB_dummy1    BIN FIXED(7),
     4 HN$BASE_IOSB_line_stat BIN FIXED(7),
     4 HN$BASE_IOSB_error     BIN FIXED(7),
     4 HN$BASE_IOSB_dummy2    BIN FIXED(7),
    3 SN$BASE_IOSB_2,
     4 LN$BASE_IOSB_stat      BIN FIXED(31),
     4 LN$BASE_IOSB_arg       BIN FIXED(31),

/* Characteristics: */
  2 CVN$BASE_address      CHAR(17) VAR,    /* Phys. address xx-xx-xx-xx-xx-xx */
  2 CN$BASE_device        CHAR(5),         /* Ethernet interface logical name */
  2 SN$BASE_iaddress,                      /* Address block for ETH routines */
    3 HAN$BASE_idest(6)   BIN FIXED(7),    /* Destination address */
    3 HAN$BASE_isource(6) BIN FIXED(7),    /* Source address */
    3 IN$BASE_iprotocol   BIN FIXED(15),   /* Protocol type */
  2 SN$BASE_oaddress,                      /* Address block for ETH routines */
    3 HAN$BASE_odest(6)   BIN FIXED(7),    /* Destination address */
    3 HAN$BASE_osource(6) BIN FIXED(7),    /* Source address */
    3 IN$BASE_oprotocol   BIN FIXED(15),   /* Protocol type */
  2 LN$BASE_header_size   BIN FIXED(31),   /* buffer header size */
  2 LN$BASE_buffer_size   BIN FIXED(31),   /* Physical buffersize */
  2 LN$BASE_buffer_max    BIN FIXED(31),   /* Maximal buffersize */

/* I/O statistics: */
  2 CN$BASE_open_date     CHAR(23),        /* Date in ASCII */
  2 CN$BASE_close_date    CHAR(23),        /* Date in ASCII */
  2 LN$BASE_write_logbuf  BIN FIXED(31),   /* Logical buffers written */
  2 LN$BASE_write_physbuf BIN FIXED(31),   /* Physical buffers written */
  2 LN$BASE_read_logbuf   BIN FIXED(31),   /* Logical buffers received */
  2 LN$BASE_read_physbuf  BIN FIXED(31),   /* Physical buffers received */
  2 LN$BASE_write_totbyte BIN FIXED(31),   /* Total bytes written */
  2 LN$BASE_write_netbyte BIN FIXED(31),   /* Data bytes written */
  2 LN$BASE_write_retry1  BIN FIXED(31),   /* Number of single retries */
  2 LN$BASE_write_retryn  BIN FIXED(31),   /* Number of multiple retries */
  2 LN$BASE_log_retry1    BIN FIXED(31),   /* Logical single retries */
  2 LN$BASE_log_retryn    BIN FIXED(31),   /* Logical multiple retries */
  2 LN$BASE_write_maxret  BIN FIXED(31),   /* Maximum retries occured */
  2 LN$BASE_read_totbyte  BIN FIXED(31),   /* Total bytes received */
  2 LN$BASE_read_netbyte  BIN FIXED(31),   /* Data bytes received */
  2 LN$BASE_write_id      BIN FIXED(31),   /* Current buffer id */
  2 LN$BASE_ackn_id       BIN FIXED(31),   /* id of last acknowledge */
  2 LN$BASE_last_id       BIN FIXED(31),   /* id of last buffer */
  2 LN$BASE_nobuf         BIN FIXED(31),   /* Number of buffers in set */
  2 LN$BASE_curbuf        BIN FIXED(31),   /* Current buffer number in set */

/* Flags and channels: */
  2 LN$BASE_write_flag    BIN FIXED(31),   /* Used for write and wait */
  2 LN$BASE_read_flag     BIN FIXED(31),   /* used for read and wait */
  2 LN$BASE_chan          BIN FIXED(31),   /* Channel */

/* Timer values: */
  2 LN$BASE_link_timer      BIN FIXED(31), /* flag for link timer */
  2 LN$BASE_write_timer     BIN FIXED(31), /* flag for write timer */
  2 LN$BASE_read_timer      BIN FIXED(31), /* flag for read timer */
  2 LAN$BASE_link_dtime(2)  BIN FIXED(31), /* link timeout */
  2 LAN$BASE_write_dtime(2) BIN FIXED(31), /* write timeout */
  2 LAN$BASE_read_dtime(2)  BIN FIXED(31), /* read timeout */
  2 LN$BASE_ackn            BIN FIXED(31), /* number of ackn to do */
  2 LN$BASE_write_retry     BIN FIXED(31), /* number of write retries to do */
  2 LN$BASE_link_retry      BIN FIXED(31), /* number of link retries to do */
```

```
/* Global control variables: */
  2 LN$base_mbx_read_flag   BIN FIXED(31), /* Flag for mbx read */
  2 LN$base_mbx_write_flag  BIN FIXED(31), /* Flag for mbx write */
  2 IN$base_mbx_ichan       BIN FIXED(15), /* Channel of read mbx */
  2 IN$base_mbx_ochan       BIN FIXED(15), /* Channel of write mbx */
  2 LN$base_palloc          BIN FIXED(31), /* Allocated physical buffers */
  2 LN$base_lalloc          BIN FIXED(31), /* Allocated logical buffers */
  2 LN$base_palloc_free     BIN FIXED(31), /* Free physical buffers */
  2 LN$base_lalloc_free     BIN FIXED(31), /* Free logical buffers */
  2 LN$base_error           BIN FIXED(31), /* Retry counter checker */
  2 LN$base_output          BIN FIXED(31), /* Total size counter */
  2 LN$base_last            BIN FIXED(31), /* Last buffer number in stream */
  2 BN$base_read_pending    BIT(1) ALIGNED,/* Open read waiting for buffer */
  2 BN$base_read_abort      BIT(1) ALIGNED,/* Logicall read timed out */
  2 BN$base_stream_ok       BIT(1) ALIGNED,/* Stream is OK */
  2 BN$base_dummy           BIT(1) ALIGNED,/* */

/* Buffer pointers: */
  2 PN$BASE_lbuf_head(2) POINTER,         /* queue for logical buffers */
  2 PN$BASE_pbuf_head(2) POINTER,         /* queue for phys buffers */
  2 PN$BASE_buf_ackn     POINTER,         /* to acknowledge buffer */
  2 PN$BASE_buf_check    POINTER,         /* to checker buffer */
  2 PN$BASE_buf_read     POINTER,         /* to user read buffer */
  2 PN$BASE_buf_log      POINTER,         /* to logical read buffer */
  2 PN$BASE_buf_write    POINTER,         /* to physical write buffer */
  2 PN$BASE_buf_swap     POINTER,         /* to swapped write buffer */
  2 IN$BASE_read_type    BIN FIXED(15),   /* user read requested type */
  2 IN$BASE_read_stype   BIN FIXED(15),   /* user read requested subtype */

/* Handler: */
  2 EN$BASE_write ENTRY(POINTER)
                 RETURNS(BIN FIXED(31)),  /* write AST */
  2 EN$BASE_handler ENTRY(POINTER)
                 RETURNS(BIN FIXED(31)),  /* Default AST */
  2 EN$BASE_abort ENTRY
                 RETURNS(BIN FIXED(31)),  /* Abort AST */
  2 PN$BASE_handler POINTER;              /* to SN$BASE_hdl */


/* Handler element */
DCL 1 SN$BASE_HDL BASED,
  2 PN$BASE_hdl_next    POINTER,            /* To next element */
  2 LN$BASE_hdl_type_l  BIN FIXED(31),      /* EMCB type low limit */
  2 LN$BASE_hdl_type_h  BIN FIXED(31),      /* EMCB type high limit */
  2 LN$BASE_hdl_stype_l BIN FIXED(31),      /* EMCB subtype low limit */
  2 LN$BASE_hdl_stype_h BIN FIXED(31),      /* EMCB subtype high limit */
  2 BN$BASE_hdl_mode    BIT(32) ALIGNED,    /* Mode bits */
  2 EN$BASE_hdl ENTRY(POINTER) RETURNS(BIN FIXED(31));
```

B.8 Types and Subtypes

B.8.1 Ethernet Buffer Types

The types in SN$EMCB are defined in $ETHREP in library GOOVME:

```
/* Ethernet Message types and subtypes */
/* Link management types ---------------------------------- */
DEFINE ETH__TYPE_CHECK 1000
                             DEFINE ETH__STYPE_CHECK   1
DEFINE ETH__TYPE_OPEN  1001
                             DEFINE ETH__STYPE_OPEN    1
DEFINE ETH__TYPE_CLOSE 1002
                             DEFINE ETH__STYPE_CLOSE   1
DEFINE ETH__TYPE_MAINT 1003
                             DEFINE ETH__STYPE_MAINT   1
                             DEFINE ETH__STYPE_ACKN    1000
/* Control systems ---------------------------------------  */
DEFINE ETH__CTRL_LOCAL 100
                             DEFINE ETH__SCTRL_LOCAL   1
DEFINE ETH__CTRL_DISP  101
                             DEFINE ETH__SCTRL_DISP    1
DEFINE ETH__CTRL_MSG   102
                             DEFINE ETH__SCTRL_MSG     1
DEFINE ETH__CTRL_DATA  103
                             DEFINE ETH__SCTRL_DATA    1
/* GOOSY systems -----------------------------------------  */
DEFINE ETH__GOOSY_LOCAL 1
                             DEFINE ETH__SGOOSY_LOCAL  1
DEFINE ETH__GOOSY_DISP  2
                             DEFINE ETH__SGOOSY_DISP   1
DEFINE ETH__GOOSY_MSG   3
                             DEFINE ETH__SGOOSY_MSG    1
DEFINE ETH__GOOSY_ACKN  4
                            DEFINE ETH__SGOOSY_ACKN   1
DEFINE ETH__GOOSY_LMD   5
                            DEFINE ETH__SGOOSY_LMD    1
```

B.8.2 Command Types

The types in SN$EMCB_CMD are defined in $VCMDREP in library GOOVME:

```
! Command definitions for VME system

DEFINE VCMD__RUN        1
                               DEFINE VCMD__S_RUN          1
DEFINE VCMD__STA_AC     2
                               DEFINE VCMD__S_STA_AC       1
DEFINE VCMD__STO_AC     3
                               DEFINE VCMD__S_STO_AC       1
DEFINE VCMD__RESET      4
                               DEFINE VCMD__S_RESET        1
DEFINE VCMD__SET        5
DEFINE VCMD__S_SET_INPCAV        1
DEFINE VCMD__S_SET_INPCAVFAST    2
DEFINE VCMD__S_SET_INPCVI        3
DEFINE VCMD__S_SET_INPAEB        4
DEFINE VCMD__S_SET_INPFEP        5
DEFINE VCMD__S_SET_INPCHAOS      6
DEFINE VCMD__S_SET_INPTEST       7
DEFINE VCMD__S_SET_OUTEB         11
DEFINE VCMD__S_SET_OUTHVR        12
DEFINE VCMD__S_SET_OUTSOFT       13
DEFINE VCMD__S_SET_OUTNET        14
DEFINE VCMD__S_SET_OUTOFF        15
DEFINE VCMD__S_SET_CHBUF         16
DEFINE VCMD__S_SET_TRIGGER       21
DEFINE VCMD__S_SET_DEBUG         31
DEFINE VCMD__S_SET_CONTROL       32
DEFINE VCMD__S_SET_BUFSIZE       41
DEFINE VCMD__S_SET_BUFQUEUE      42
DEFINE VCMD__S_SET_TIMEOUT       51
DEFINE VCMD__S_SET_MAXSPAN       52
DEFINE VCMD__EXE        6
                               DEFINE VCMD__S_EXE_CNAF     1
```

```
                                DEFINE VCMD__S_EXE_INIT     2
                                DEFINE VCMD__S_EXE_RESET    3
DEFINE VCMD__ACTIVE     7
DEFINE VCMD__DEACTIVE   8
DEFINE VCMD__S_ACT_LISTREAD   1
DEFINE VCMD__S_ACT_USERREAD   2
DEFINE VCMD__S_ACT_USERANAL   3
DEFINE VCMD__S_ACT_USERPROC   4
DEFINE VCMD__S_ACT_MONITOR    5
DEFINE VCMD__ESONE      9
        DEFINE VCMD__S_ESONE        1
DEFINE VCMD__SHO        10
DEFINE VCMD__S_SHO_INPCAV        1
DEFINE VCMD__S_SHO_INPCAVFAST    2
DEFINE VCMD__S_SHO_INPCVI        3
DEFINE VCMD__S_SHO_INPAEB        4
DEFINE VCMD__S_SHO_INPFEP        5
DEFINE VCMD__S_SHO_INPCHAOS      6
DEFINE VCMD__S_SHO_INPTEST       7
DEFINE VCMD__S_SHO_OUTEB         11
DEFINE VCMD__S_SHO_OUTCES        12
DEFINE VCMD__S_SHO_OUTSOFT       13
DEFINE VCMD__S_SHO_OUTNET        14
DEFINE VCMD__S_SHO_OUTTEST       15
DEFINE VCMD__S_SHO_INPNORM       21
DEFINE VCMD__S_SHO_INPIGNORE     22
DEFINE VCMD__S_SHO_CONTROL       32
DEFINE VCMD__S_SHO_BUFSIZE       41
DEFINE VCMD__S_SHO_BUFQUEUE      42
DEFINE VCMD__S_SHO_TIMEOUT       51
DEFINE VCMD__S_SHO_MAXSPAN       52
DEFINE VCMD__GET_LMD     11
        DEFINE VCMD__S_GET_LMD        1
DEFINE VCMD__NET_ACKN    12
        DEFINE VCMD__S_NET_ACKN       1
DEFINE VCMD__LOA_CAM    2001
        DEFINE VCMD__S_LOA_CAMIN    1
        DEFINE VCMD__S_LOA_CAMRD    2
        DEFINE VCMD__S_LOA_CAMRS    3
DEFINE VCMD__LOA_FB     2002
                                DEFINE VCMD__S_LOA_FB       1
DEFINE VCMD__LOA_VME    2003
```

```
                                DEFINE VCMD__S_LOA_VME      1
DEFINE VCMD__EOLOAD     2004
                                DEFINE VCMD__S_EOLOADL      1
DEFINE VCMD__LOA_PR     2005
                                DEFINE VCMD__S_LOA_PR       1
                                DEFINE VCMD__S_EOLOADP      2
```

B.8.3 Command Types

The types in SN$EMCB_MSG are defined in $VMSGREP in library GOOVME:

```
! Message definitions for VME system

DEFINE VMSG__ACK        1
                                DEFINE VMSG__S_ACK          1
DEFINE VMSG__ERROR      2
                                DEFINE VMSG__S_STRING       1
                                DEFINE VMSG__S_LONG         2
DEFINE VMSG__LMD        3
                                DEFINE VMSG__S_LMD          1
```

Appendix C

VME Structure Declaration (DECL)

VMESTRUC

```
VMESTRUC inputfile /PLI/FOR/C/PLIB=/CLIB=/FLIB=
                   /GLPUT/DELETE
```

PURPOSE	Generate declarations from language independent source.
ARGUMENTS	
inputfile	File containing language independent decla- rations. Specify library module as library(module).
/PLI/FOR/C	Controls, which output is generated.
/PLIB/FLIB=	Optional VMS text libraries to store the generated modules.
/CLIB=	Optional directory to store generated modules. I.e. VME$INC:
/GLPUT	Use GLPUT command instead of LIB/REP or COPY
/DELETE	Delete generated files.

Description

CALLING	VMESTRUC inputfile /PLI/FOR/C/PLIB=/CLIB=/FLIB= /GLPUT/DELETE
ARGUMENTS	
inputfile	File containing language independent declarations. Default file type is .VMES. Specify library module as library(module).
/PLI	Output PL/1 include file. Filename is inputfile.PINC.
/FOR	Output FORTARN include file. Filename is inputfile.FINC.
/C	Output C include file. Filename is inputfile..
/PLIB/FLIB=	Optional VMS text libraries to store the generated modules witrh PL/1 or FORTRAN syntax.

/CLIB=	Optional directory to store generated modules with C syntax, i.e. VME$INC:
/GLPUT	Use GLPUT command instead of LIB/REP or COPY
/DELETE	Delete generated files.
FUNCTION	Declarations of constants, variables and structures are translated into the proper PL/1, FORTRAN or C statements. The syntax of the source file is described below. If none of the qualifier is selected, all three output files are generated. The syntax of each include file is checked by a test compilation.
EXAMPLE	VMESTR sysctrl generates sysctr.pinc, sysctrl.finc and sysctrl..

Implementation

Version	1.01
Author	H.G.Essel
Last Update	27-NOV-1988

Updates

Updates	Date Purpose

Internals

Utility	UTIL
Home direct.	GOO$EXE
Created	27-NOV-1988
Called prog.	MVMESTRUC

Syntax

The syntax of the source file is:
! comment at any place
DEFINE constant value
[EXTERNAL]LONG [POINTER]name[(i1,i2)]
[EXTERNAL]WORD [POINTER]name[(i1,i2)]

[EXTERNAL]BYTE [POINTER]name[(i1,i2)]
[EXTERNAL]BIT [POINTER]name[(i1,i2)]
[EXTERNAL]FLOAT [POINTER]name[(i1,i2)]
[EXTERNAL]STRUCTURE [POINTER]structure name
the structure must be known.
STRUCTURE [POINTER]structure
structure declarations
ENDSTRUCTURE

All keywords except POINTER may be abbreviated. The array dimensions may be constants previously defined. A constant is defined for each structure with the value of the length. The variable names for PL/1 and C are prefixed by type letters, i.e. L_ for longword.

MESDEF

MESDEF facility /CLIB=/NEW/GLPUT/DELETE

PURPOSE	Generate message definition file for C programs.
ARGUMENTS	
Facility	Name of facility, i.e. GOOVME.
/CLIB=	Optional directory to copy the definition file.
/NEW	New version is linked. Facility GOOVME should be always up to date.
/GLPUT	Use GLPUT instead of copy, if library is specified.
/DELETE	Delete definition file (makes sense only if library is specified).

Description

FUNCTION Program MMESDEF is called and generates a file X<facility>. with define statements for the messages. This file can be included in C programs. To add new messages, modify the message file on GOO$MSG.

EXAMPLE

```
$ GLGET GOO$MSG:XVME.MSG
  $ LSE XVME.MSG
  $ GLPUT XVME.MSG GOO$MSG
  $ MESDEF GOOVME VME$INC /GLPUT/DEL
```

Implementation

Version	1.01
Author	H.G.Essel
Last Update	14-feb-1990

Updates

Updates	Date Purpose

Internals

Utility	UTIL
Home direct.	GOO$EXE
Created	14-feb-1990
Called prog.	MMESDEF

Appendix D

VME Load Commands (LOAD)

LOAD VME PROGRAM

```
LOAD VME PROGRAM file procrate processor subcrate
                 node
                 /FEP/EB
                 /ROP/ROC/AEB/VME
                 /USER/SYSTEM
                 /[NO]LOAD
```

PURPOSE	Load programs into VME processors
PARAMETERS	
file	string File containing exec code produced by PCP. or @file containing description file. For this case only qualifier /USER/SYSTEM is used.
procrate	integer VME processor crate
processor	integer Processor offset 0-13
node	J11 node (default=J11B)
/FEP	Processor is FEP
/EB	Processor is EB
/ROP	Processor controls CAMAC crate with processor
/ROC	Processor controls CAMAC crate with controller
/AEB	Processor controls Fastbus crate with AEB
/VME	Processor controls VME crate.
/USER	Load user routine specified in file.
/SYSTEM	Load system executive.
/LOAD	Load modules (=default)

EXAMPLE	LOAD VME PROG prog.exe20 1 0 /FEP/ROC/USER LOAD VME PROG PROCR=1 PROCE=1 /FEP/ROC/SYSTEM LOAD VME PROG PROCR=1 PROCE=1 SUBCR=1 /ROP/SYSTEM LOAD VME PROG @setup.vme /SYSTEM LOAD VME PROG @setup.vme /USER

Description

FUNCTION	Loads programs to VME processors.
File name	I$ACQ_LOA_VME_P.PPL
Action rout.	I$ACQ_LOA_VME_P
Dataset	-
Version	1.01
Author	H.G.Essel
Last Update	16-feb-1989

Syntax

The description file contain lines with the following syntax:

lines	<object> <specification>,<specification>,...
object	PROCESSOR — MODULE
specification	<key>=<value>
value	string — number — (<spec>,<spec>,...) — (number,number,...)

PROCESSOR	Processor specification lines specify
	1. The software to be loaded
	2. The Bus and processor types
	BRANCH=(c,p),
	CONTROL=ROP—ROC—AEB—VME,
	TYPE=FEP—EB,CRATE=c,USER=file

MODULE	Module specification lines specify 1. the location of a hardware module 2. the readout, reset or init information.

```
BRANCH=(CRATE=c,OFFSET=p),
CONTROL=ROP—ROC—AEB—VME,
TYPE=type,CRATE=c,STATION=n,SUBADDRESS=a,
INIT=(FUNCTION=f,REPEAT=r,EXEC=e,DATA=d),
READ=(FUNCTION=f,REPEAT=r,EXEC=e),
RESET=(FUNCTION=f,REPEAT=r,EXEC=e),
CHANNEL=c,DATA=(v,v,...)
```

LOAD VME TABLE

```
LOAD VME TABLE file procrate processor subcrate
                     node
                     /FEP/EB
                     /ROP/ROC/AEB/VME
                     /USER/SYSTEM
                     /[NO]LOAD
```

PURPOSE	Load tables into VME processors
PARAMETERS	
file	string File containing description file.
procrate	integer VME processor crate
processor	integer Processor offset 0-13
node	J11 node (default=J11B)
/FEP	Processor is FEP
/EB	Processor is EB
/ROP	Processor controls CAMAC crate with processor
/ROC	Processor controls CAMAC crate with controller
/AEB	Processor controls Fastbus crate with AEB
/VME	Processor controls VME crate.
/LOAD	Load tables (=default)
EXAMPLE	LOAD VME TAB MYSETUP.VME

Description

FUNCTION	Loads tables to VME processors.
File name	I$ACQ_LOA_VME_T.PPL
Action rout.	I$ACQ_LOA_VME_T
Dataset	-
Version	1.01
Author	H.G.Essel
Last Update	16-feb-1989

Syntax

See command LOAD VME PROGRAM

Appendix E

Foreign VME Equipment (FOREIGN)

E.1 Conditions To Be Taken Into Account When Using Foreign VME Equipment

Whenever possible all equipment for experiment control and data acquisition should be CAMAC or Fastbus standard. This provides easy integration and a maximum software support for testing and read-out.
When it is impossible to avoid the use of VME modules, the devices in question should have a VSB-slave interface following the VSB Rev. C standard. In this case one dedicated front-end processor (FIC 8230 from CES) can be used for handling these VSB slave devices. This solution allows to have nearly no restriction concerning the communication and data exchange between the controller (FIC 8230) and the VSB slave devices. At least one of the VSB-slaves must have the VSB-pullup resistors on board. When these devices have a VME slave port it must be possible to disable this port providing that no VME address space is governed by these devices.
When taking this solution, the software for communication with the host computer and the event builder can be supplied. If the devices in question do only have VME slave interfaces, there are two possible solutions:

- The VME slave devices should be mounted in a seperate VME crate which can be accessed via VMV (VME crate interconnect bus).
  This crate will have the following modules:

```
Slot  1 : FIC 8230 with system controller and arbiter option,
          VME address window 0 - FFFFF, A24/D32

Slot 18 : VMDIS 8001 dataway display
Slot 19 : VBR 8213 VMV slave unit,
          VME address window DFFE00 - DFFFFF, A24/D32

Slot 20 : VBE 8212 VMV master unit,
          VME address window E00000 - EFFFFF, A24/D32
```

The software for communication with host computer and Event builder can be supplied. VME IRQ 1 level is used for the communication with the host computer.

- When the VME slave devices should be used in a crate where other modules of the data acquisition system are resident, there are the following restrictions:

```
- Free VME address space: D00000 - DFFD00   (A24/D32),
                          0      - FFFFFFFF (A32/D32),
                          0      - FFFF     (A16/D16)

- VME IRQ 4 and 5 lines are free to use

- minimum 3 to maximum 8 free VME slots

- a FIC 8230 must be used as controller to have a minimum software
  support for communication with the host computer and the event
  builder, otherwise there will be NO software support at all.

- before system integration takes place it has to be checked that
  the VME slave devices in question work properly together with
  the other devices.

- it has to be taken into account that any bus traffic caused by
  such devices will slow down the system throughput.
```

VME Glossary

Communication server NET ISC8221 processor PDP11/73 + DEQNA. Handles commands and messages. See *Net Server.*

Control Status Region CSR Global accessible memory region for each VME processor.

Event Builder EB FIC 8230 processor M68020. The event builder collects the subevents from the FEP's and packs them into the event buffer.

ESONE server Process to execute server functions like ESONE calls.

Event Total event data composed in the EB.

Event handler Process to control the listmode data stream in the EB.

FEP table Table in the EB with one entry for each FEP. The entries keep information about the FEP status. The FEP's have direct write acces to their entry.

Front End Processor FEP FIC 8230 processor M68020. Controls one or several CAMAC crates or FASTBUS crates. Each FEP has its own VSB interface to the CAMAC readout controller, CAMAC readout processor or FASTBUS readout processor.

Interrupt Service Procedure ISP Routine executed after an interrupt.

Net Server NET VME processor connected to Ethernet for command, message and optional data communication. Currently an ISC8221 + DEQNA.

PCP Philips Compiler Package running under VAX/VMS. It includes a cross compiler for Motorola 68k assembler and 'C' programs and a cross linker.

pSOS Multitasking realtime system kernel for Motorola 68000, 68010, 68020 and 68030 processors. It is used for all ROP, FEP and EB processors.

Process Global Control Structure Keeping all information and data fields common to all processes. Must be accessible via VSB and VME.

Process Local Control Structure Keeping all information and data fields of a process. Must be accessible via VSB and VME.

Process table Control structure in a FEP, EB, or ROP keeping information about the processes.

Readout Controller ROC Executes CAMAC cycles controled by the FEP.

Readout Processor ROP Reads one CAMAC crate and interrupts FEP via VSB if ready.

Software Cancel Conversion SCC When one FEP gets enough data to decide whether an event is valid during the conversion time of other crates, it may cancel the conversion.

Software Cancel Readout SCR When one FEP gets enough data to decide if an event is valid during the readout time of other FEP's, it may cancel the readout. An empty event is passed to the EB together with the number of the canceling FEP and a message code.

Software Event Filter SEF When the EB has read all subevents, it may decide if the event is valid. If not, it marks the event as unvalid and writes an empty event into the event buffer together with a message code. This kind of filter could also be executed by several processors fed by the EB.

Software Subevent Filter SSF When all FEP's have read their subevent, any of them may decide if the event is valid. If not, it marks the subevent as unvalid. The EB recognises unvalid subevents and writes an empty event into the event buffer together with the FEP number and a message code.

Software Trigger STR Programm running in a VME processor to reduce event data by experiment specific calculations.

Subevent Event data of one FEP.

Subevent Cluster Several subevents can be clustered, i.e they are transferred with one DMA from FEP to EB. The subevents are contiguous. The clustersize (number of subevents, not bytes!) can be modified.

Subevent handler Process to control the listmode data stream in the FEP's.

Supervisor Main pSOS program to startup all other processes.

Index

Contents