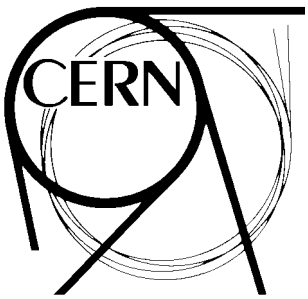
CERN ECP/FEX-CA 97-1
Revision 3_01
Mars 1997




# CASCADE

## User's Guide

**CERN - ECP Division - FEX Group - Cascade Section**

## Copyright


## CERN

**European Organization for Nuclear Research**

**CH-1211 Geneva 23**

**Switzerland**

# Abstract

CASCADE (CERN Architecture and System Components for an Adaptable Data-acquisition Environment) is a software package developed at CERN by the data-acquisition group of the ECP Division for the construction of distributed, real-time, data-acquisition systems for high-energy physics experiments. Originally targeted to the NOMAD experiment, CASCADE has been designed to adapt to a wide range of system configurations and to provide physicists with a set of software building blocks so that they can construct homogeneous data-acquisition systems.

CASCADE has been used in a number of applications at CERN. After several improvement phases, it is now a stable product dedicated to the NOMAD and LHCB experiments.

# Intended Audience

This document is intended for the users of the Cascade data-acquisition system.

# Product Version

This document refers to Cascade version 3_01.

# Document Information

Comments, suggestions, queries and criticisms about the present document should be sent to the editor: Yves Perrin (CERN ECP/FEX, phone: +41.22.7673194, E-mail: Yves.PERRIN@cern.ch).

# Document Access

This document, as well as other CASCADE related material can be retrieved:

- via WWW through URL http://cascade.cern.ch/Documentation

# Table of Contents


**1 Overview 1**

1.1 Basic Concepts . . . . . 1
1.2 Building Elements . . . . . 2
1.2.1 The Stage . . . . . 2
1.2.2 The Inter-Stage Link . . . . . 3
1.3 Event Producers . . . . . 4
1.4 Event Building . . . . . 4
1.5 Event Consumers . . . . . 4
1.5.1 Event Recording . . . . . 4
1.5.2 Event Monitoring . . . . . 5
1.6 Run Control . . . . . 5
1.7 Error Reporting . . . . . 6
1.8 Infrastructure . . . . . 6
1.8.1 Component Packages. . . . . 6
1.8.2 General Purpose Packages Used . . . . . 7
1.9 Application-dependent Parts . . . . . 7
1.10 Software Distribution . . . . . 8
1.11 Status . . . . . 8

**2 Required Layered Products 9**

2.1 Network services and daemons . . . . . 9
2.2 System access permissions . . . . . 10
2.3 Mandatory Layered Products . . . . . 10
2.4 Optional Layered Products . . . . . 11
2.5 The CERN ECP/FEX-CA I/O drivers and libraries . . . . . 11

**3 Getting the Software 13**

3.1 Where to get the software from . . . . . 13
3.2 Which software to download where . . . . . 13
3.3 Setting up the directory infrastructure necessary to host CASCADE . . . 13
3.4 Downloading of the CASCADE software . . . . . 14

**4 Building a CASCADE System 19**

4.1 The Cascade demo system . . . . . 19
4.2 Building a Cascade system for a real application . . . . . 20
4.3 Event Production and Event Building Function Development . . . . 22
4.4 Stage Generation . . . . . 23
4.5 Monitoring programs Development . . . . . 24
4.6 Execution of the demo monitoring programs. . . . . 24
4.7 Configuration and Run Control Files Preparation . . . . . 25
4.8 Writing EMU message and route files . . . . . 25
4.9 Preparing the script files to start the application . . . . . 25

**5 Booting a CASCADE System 27**

5.1 Introduction . . . . . 27

5.2 Stage ’start’ Scripts . . . . . 27
5.2.1 The Stage Environment Variables . . . . . 28
5.3 Recorder ’start’ Scripts . . . . . 29
5.3.1 The Recorder Environment Variables . . . . . 30

**6 Run Control 31**

6.1 Introduction . . . . . 31
6.2 Environment Requirements . . . . . 31
6.3 The CASCADE configuration database. . . . . 32
6.4 Starting CASCADE Run Control . . . . . 32
6.5 The CASCADE user interface . . . . . 33
6.5.1 The DAQ menu . . . . . 35
6.5.2 The individual stage pop-up menus . . . . . 39

**7 Event Production 45**

7.1 Introduction . . . . . 45
7.2 Programming Considerations. . . . . 45
7.3 Event headers and event production templates . . . . . 47
7.4 User Routines - First Group . . . . . 47
ProdStageInit 49
ProdPortInit 50
ProdStageStartRun 51
ProdPortStartRun 52
ProdPortStopRun 53
ProdStageStopRun 54
ProdInput 55
7.4.1 CASCADE 'event' functions . . . . . 56
STG_GetSpace 57
STG_ReleaseSpace 58
STG_DeclEvent 59
STG_SetEvAttr 60
STG_GetRunState 61
STG_GetPortPriority 62
STG_SetPortPriority 63
Remark about event sizes 64
7.5 User Routines - Second Group . . . . . 64
7.6 ZEBRA FZ User Vector used in the distributed templates . . . . . 64
USTG_StageInit 66
USTG_PortInit 67
USTG_StageStartRun 68
USTG_PortStartRun 69
USTG_StageStopRun 70
USTG_PortStopRun 71
USTG_GetVICNo 72
USTG_InputPortParam 73
USTG_InputEnbTrig 74
USTG_InputDsbTrig 75
USTG_EvTrgAna 76
USTG_MaxEventSize 77
USTG_InputEvent 78

USTG_FZuserVec 79

**8 Event building 81**

8.1 Introduction . . . . . 81
8.2 userevb templates . . . . . 81
8.3 Application-dependent event building functions . . . . . 82
UEVB_Init 83
UEVB_GetMinInfo 84
UEVB_GetMoreInfo 85
UEVB_FillHdTr 86

**9 Event Monitoring 87**

9.1 Introduction . . . . . 87
9.2 Monitoring program templates . . . . . 87
9.2.1 Monitoring Program Structure . . . . . 89
9.3 Error Testing In A Monitoring Program . . . . . 90
9.4 Exception Handler In The Monitoring Program . . . . . 90
9.5 Debugging Tools . . . . . 91
9.6 Pipes . . . . . 92
9.7 Sampling Routines . . . . . 93
sh_mconnect 94
sh_declare_signal 96
sh_request_event 97
sh_wait 98
sh_get_data 99
sh_release_event 100
sh_disconnect 101
sh_message 102
9.7.1 FORTRAN Interface . . . . . 103

**10 Remote Monitoring Facility 105**

10.1 Introduction . . . . . 105
10.2 Remote monitoring functions . . . . . 107
RM_init 109
sh_wait_timeout 110
sh_mconnect 111
sh_request_event 112
sh_wait 113
sh_get_data 114
sh_release_event 115
sh_disconnect 116
sh_message 117
10.3 Clients and servers . . . . . 118
10.4 Installation on OS-9 . . . . . 120
10.5 Installation on LYNXOS . . . . . 120

**11 Data Recording 123**

11.1 Introduction . . . . . 123
11.2 Event formatting and packing . . . . . 124
11.3 Starting the Recorder . . . . . 124

11.4 The dummy recorder . . . . . 125
11.5 Tape Recorder for SUMMIT . . . . . 125
11.5.1 Environment Variables for SUMMIT Tape Recording . . . . . 126
11.5.2 Unlabelled Tapes . . . . . 126
11.5.3 Labelled tapes . . . . . 127
11.6 Tape recorder for DLT . . . . . 130
11.7 Tape recorder for EXABYTE . . . . . 130
11.7.1 Environment variables for EXABYTE tape recorder . . . . . 130
11.7.2 Recording session . . . . . 130
11.8 Remote disk recorder . . . . . 131
11.8.1 Environment variables for remote disk recording . . . . . 131
11.8.2 Setting up the disk server . . . . . 132
11.9 Logging of tape information . . . . . 132

**12 Error and Message Handling and Reporting 135**

12.1 Introduction . . . . . 135
12.2 General Flow of Cascade Error and Message Handling . . . . . 135
12.3 Error Message Utility EMU in CASCADE . . . . . 137
12.3.1 Overview . . . . . 137
12.3.2 The EMU kernel . . . . . 137
12.3.3 The EMN network layer . . . . . 137
12.4 Message streams and Severity in CASCADE error message handling . . . 137
12.4.1 Message Streams . . . . . 137
12.4.2 Severity . . . . . 138
12.4.3 Reserved names for CASCADE . . . . . 138
12.4.4 Example . . . . . 139
12.5 Message injection from CASCADE into EMU . . . . . 139
EMH_UsrMsg and EMH_SysMsg 140
12.6 EMU Message Decoding . . . . . 142
12.7 EMU Message Routing . . . . . 142
12.8 Example . . . . . 142
12.8.1 Installation . . . . . 143
12.8.2 Example Configuration . . . . . 143
12.8.3 EMN setup . . . . . 143
12.8.4 The EMU Message file . . . . . 144
12.8.5 The EMU Router file . . . . . 145
12.8.6 Message injection . . . . . 145
12.8.7 EmuDisplay and logfile . . . . . 146

**13 Configuration Files 147**

13.1 Overall File Structure . . . . . 147
13.2 Global System Parameters . . . . . 147
13.3 Stage Entry . . . . . 148
13.3.1 Global Stage Parameters Section . . . . . 148
13.3.2 Stage Input Ports Section . . . . . 149
13.3.3 Stage Output Ports Section . . . . . 149
13.4 Inter-stage Links . . . . . 150
13.4.1 VICbus links . . . . . 150
13.4.2 Network Links . . . . . 150
13.4.3 Stage to Recorder Shared Memory links . . . . . 151

13.4.4 Detector to Front-end Stage links . . . . . 151
13.5 Additional Stage Information Related to Event Building . . . . . 151
13.5.1 The stage event type section . . . . . 152
13.6 Configuration File Templates. . . . . 153
The demo scripts suite 155

**References** **157**

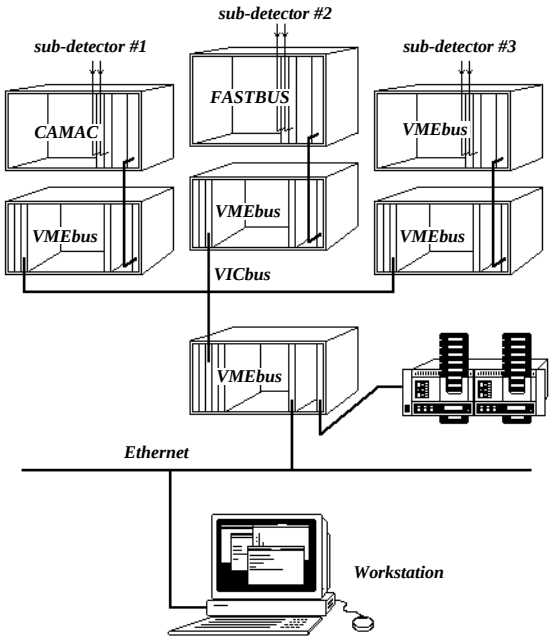
# 1 Overview

CASCADE (CERN Architecture and System Components for an Adaptable Data-acquisition Environment) is a software package developed at CERN by the data-acquisition group of the ECP Division for the construction of distributed, real-time, data-acquisition systems for high-energy physics experiments. Originally targeted to the NOMAD experiment, CASCADE has been designed to adapt to a wide range of system configurations and to provide a set of software building blocks so that data-acquisition systems can be constructed in a homogeneous way.

## 1.1 Basic Concepts

Most data-acquisition systems in high-energy physics can be viewed as multiprocessor, tree-structured architectures. Data flow through the architecture under the form of *events* which group information related to a given trigger. All events progress in the tree in the same direction. In their migration, events are manipulated for various reasons such as filtering, formatting, merging, monitoring, routing, etc. These operations are performed by processes running in a set of processors distributed in the architecture. Depending on their role and on their physical environment in the configuration, these processors may be of different types, may run different operating systems, and may be linked by a variety of inter-processor links as shown in Figure 1. To maximize the overall efficiency, data buffers and processing elements are distributed in the architecture so that different levels of the chain can work concurrently on different events.

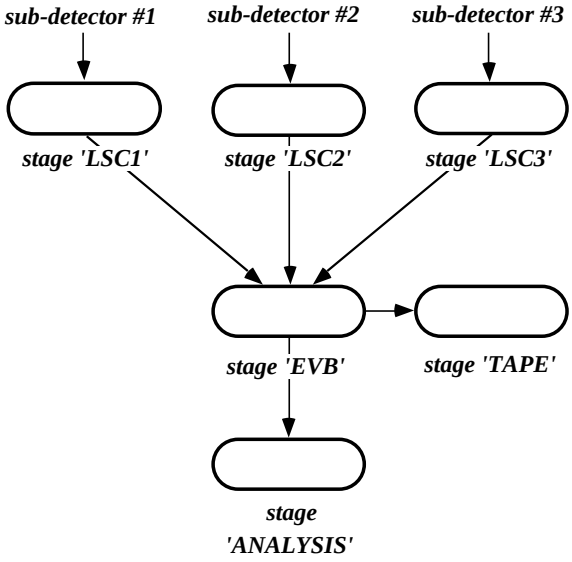
**Figure 1** Physical System Representation




To a large extent, CASCADE allows its users to concentrate more on the functional aspects of the data-acquisition than on its physical implementation. This is achieved by grouping the most common functions in two basic building elements called the **stage** and the **inter-stage link** respectively. The stage has been implemented on a number of

hardware and software platforms widely used at CERN. Using these basic elements, which can be replicated at all levels, it is possible to make a logical representation (Figure 2) of the actual physical configuration and specify the mapping between the two representations in a **configuration file**. In this way, the data-acquisition architecture can be viewed and handled in a more homogeneous way.

**Figure 2** Logical System Representation




## 1.2 Building Elements

### 1.2.1 The Stage

A stage has the basic functionality of a general single-processor, single-process, data-acquisition kernel. It is structured and parameterised so that several stages can be grouped together to form sub-systems such as event builders, farms, etc. As far as the data-flow is concerned, a stage could be viewed as a pipeline with one or more inputs and one or more outputs as shown in Figure 3.

Upon reception of a trigger the stage inputs an event, performs a number of possible operations on this event, and finally passes the results to other stages and to monitoring tasks which subscribed to this type of event. Inside the stage, events are handled and buffered in the form of event descriptors. The event data are copied in the stage only if strictly necessary. To minimize the dead time and to allow operations to take place concurrently on different events so that the stage can deal with different input and output rates, the stage is organized in several threads of execution. Each thread corresponds to a given operation to be performed on an event or to a control action to be done on the stage. Threads directly involved with the main data-flow are called **phases**.

In a stage, event descriptors transit sequentially through:

- the input phase, which creates an event descriptor and, if necessary, copies the event data;

- the construction phase, which, in the case of an event builder, gradually links together the descriptors of the sub-events until a complete event is created.

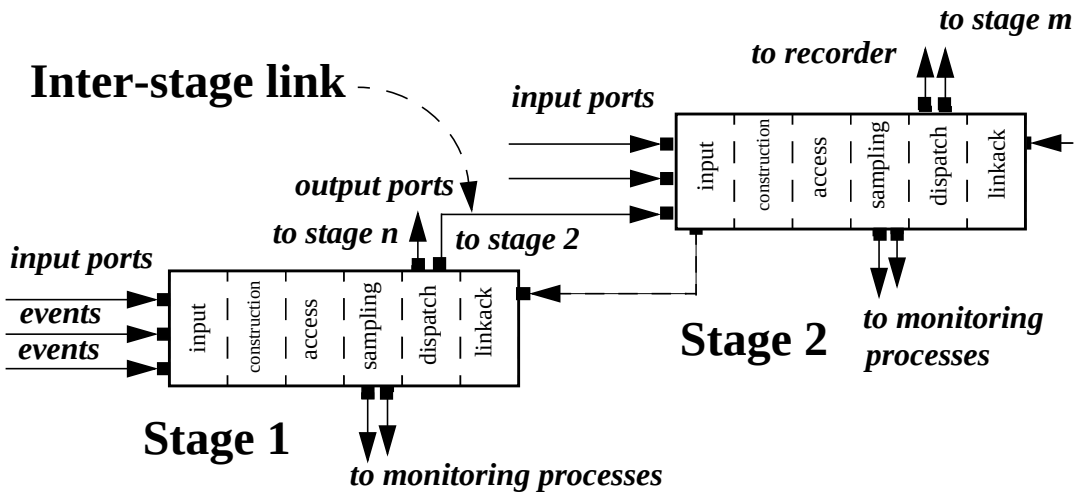
From there, event descriptors are pushed to:

- the access phase where they are marked for monitoring;
- the dispatch phase which format, and output them to one or several other stages.

In case of monitoring programs connected to the stage, event descriptors transit through the sampling handler phase which transfers the relevant pointers and sizes to the monitoring process so that it can access the event.

A scheduler has been developed to control the execution of the threads. It reacts to signals associated with each thread and issued either externally by other processes (stages, monitoring programs, control programs) or internally by one of the other threads. If necessary, a thread can suspend its execution until a global shared resource (e.g. memory space) has been released. Thread scheduling is done on a priority basis but a thread cannot be pre-empted to give control to any other thread before it terminates its execution.

**Figure 3** The stage and the inter-stage link




### 1.2.2 The Inter-Stage Link

Two consecutive stages in the data-flow topology can be linked physically in a number of ways. A high-level interface and a handshake protocol have been specified, and implementations have been done for a number of hardware and software platforms widely used at CERN. This approach provides homogeneity in the overall system. It makes the communication between stages transparent to the application-dependent code. At present, implementations exist allowing stages to be linked via VICbus [1] or via Ethernet (in that case using TCP/IP).

Communication between two stages is initiated by the dispatch phase of the upstream stage which triggers the input phase of the downstream stage by sending it a signal.

The protocol includes exchange of a message containing the event descriptor, possibly followed by the transfer of the event data if the type of link makes it necessary or if the user has explicitly specified it in the configuration file. An acknowledge message is sent by the downstream stage once it is ready to work on the event.

## 1.3 Event Producers

Events are produced by stages which have input ports declared to be of type USER in the application configuration file. These stages are generally (but they don't have to be) 'front-end' stages. They differ from the others only by the fact that some application-dependent **event production functions** have to be linked with the stage modules when the system is generated. At execution time, when the stage is triggered on one of its input ports, the input phase calls the appropriate event production functions if the input port is of type USER. These functions, for which templates are available, must read the event data and declare the event to the stage. CASCADE and a number of I/O libraries, listed later in this document, are available to read events from a variety of busses often used in HEP experiments.

## 1.4 Event Building

In most cases stages have to supply their output ports and their monitoring programs with the events they have collected from their input ports. If necessary, it is possible to force stages to perform multi-level event-building operations on the (sub)events collected from the input ports. This mode of operation has to be specified in the application configuration files together with the event types and some dependency rules involved in the building process. A number of application-dependent **event building functions** have to be written and linked with the stage modules at system generation. These functions, for which templates are available, are automatically called by the stage to get information on the subevents and, if necessary, re-format them before performing the actual building operation.

## 1.5 Event Consumers

### 1.5.1 Event Recording

Recording is done by a special type of stage called a **recorder** which has the unique task of storing events on an I/O device. Recorders must run in the same CPU as the stage which feeds them. Recorders have only one input and have no output to other stages. The interface and the protocol used to communicate with a recorder are the same as for inter-stage communications but the inter-stage link is based on shared memory. The feeder stage, in its dispatch phase, formats the event and sends it to the recorder which, then, writes it in fixed-length records on the storage device. Splitting the formatting and the actual recording over two processes permits concurrent execution of these two operations. CASCADE supports the CERN ZEBRA FZ format. At present, recording can be done on disk files (locally or via NFS) and, under OS-9 and LYNXOS, it can be done on the IBM3480-compatible STK4280 cartridge device, on the EXABYTE or on a DLT [Digital Linear Tape] device. The recorder can also be configured to use its own disk recorder server on a UNIX system accessed via the network.

### 1.5.2 Event Monitoring

Monitoring programs run as separate processes either locally in the same CPU as the stage from which they retrieve events or remotely in an other CPU. A set of functions, provided in the form of a library, can be called by the monitoring programs to connect to a given stage and specify the event sampling criteria, to request an event, to release an event, and to disconnect from a stage.

There are two modes of sampling. In the **request mode** the monitoring program receives an event as soon as there is one available but with no guarantee that a minimum percentage of events will be seen. In the **fixed mode** the monitoring program is guaranteed to receive at least the percentage of events that it has specified and more if possible. In both modes the monitoring program is notified asynchronously as soon as an event of the requested type becomes available.

An event is made available to the monitoring program by means of pointers and sizes. The monitoring program does not know whether the event data is local to the stage or remote in a previous stage. The space occupied by the event is automatically locked and must be explicitly released as soon as the monitoring program has finished with it.

Remote monitoring is based on a client/server approach where a client monitoring program, usually running in a workstation, relies on a server program, running in the same processor as the stage, to 'transparently' handle all of its monitoring requests. The client and the server communicate via Ethernet using TCP-IP.

The communication between the monitoring programs (or their servers in case of remote monitoring) and the stage is based on pipes for the exchange of request and reply messages, and on shared memory for access to events. In the stage a service thread, called the sampling handler, serves the asynchronous requests issued by the various monitoring programs. It also handles the event bookkeeping in conjunction with the stage access phase.

## 1.6 Run Control

The run control facility provided as part of the CASCADE package involves a process called XRC running on a Solaris or HPUX workstation.

**XRC** is a modular, general-purpose control program allowing complex data-acquisition systems to be modeled in an object-oriented way. It is based on software originally designed by the CERN OPAL experiment and adapted in collaboration with NOMAD. Operator interaction with the data-acquisition system is achieved through a X11/Motif layer which provides a run-time configurable graphical interface including menus, dialog boxes and various types of display panels.

XRC is a process controlling data-acquisition (DAQ) units such as stages, recorders, monitoring programs and user-specific processes. Its main purpose is to provide synchronization between various DAQ units and to hold their respective states. Within XRC, each element is described in a uniform way as a Finite State Machine (FSM), an object having a predefined set of allowed states and allowed transitions between these states. Since DAQ units are external to XRC, they are represented by internal FSM

correspondents. A hierarchy of internal FSMs can be introduced to control subsets of the entire DAQ system.

Communication between XRC and the DAQ units uses NIC, the CASCADE Network component package based on the TCP/IP protocol. The XRC process is the network server and the DAQ units are the clients, which can connect dynamically to the server. XRC maintains the states of all the data-acquisition components as well as run-time parameters and saves this information at the end of each session. A run control domain is defined by an identifier used by XRC and the connected DAQ units. In a given domain each object is identified by its unique ASCII name. More than one instance of the run control facility may be running at a given time using different addressing domains. For example a full production run can coexist with the test or calibration of a particular sub-detector. A simple but powerful user library allows for easy preparation of user specific DAQ unit software.

The DAQ units configuration is stored in the CASCADE run control configuration kept in a mini SQL database [2]. Mini SQL (mSQL) is a lightweight database engine, available in the public domain, designed to provide fast access to stored data with low memory requirements. mSQL offers a subset of the ANSI SQL specification.

Run control and associated stage mechanisms allow to, dynamically detach/re-attach stages for different DAQ runs of the same application. This facility permits to run with subsets of the run control configuration with which the application has been initialised without having to update the mSQL database and restart the application.

## 1.7 Error Reporting

Several facilities are available to handle error messages originating from both the application specific modules and the CASCADE package itself. They allow message preparation outside the application code, selective message routing at run time, message transport across heterogeneous operating system platforms, support for a variety of message destination types from log files to MOTIF windows.

These facilities have been implemented both on UNIX-like and OS-9 platforms using a number of packages which are not specific to CASCADE such as EMU [20] [21], the Error Message Utility, EMUNET [22], the transparent TCP/IP network connection of distributed EMU systems and ED [23], a MOTIF-based EMU messages display program.

## 1.8 Infrastructure

The CASCADE package relies itself on a number of other packages, each of them providing a given category of required services. Some of the requirements have been implemented by developing packages specifically for this project. These packages are therefore 'components' of CASCADE. For other requirements, it was possible to use already existing packages.

### 1.8.1 Component Packages

Most of these packages are written in C and some of them in C++. They essentially handle the following services:

- management of event descriptors, linked lists, space allocation, and shared memory segments
- thread scheduling
- network communication
- inter-process pipe communication
- run control services
- monitoring services
- application configuration file interpretation
- event building
- recording facilities
- debugging facilities

### 1.8.2 General Purpose Packages Used

In addition to some packages already mentioned in the context of run control and of error reporting, CASCADE relies also on various other packages which are not specific to CASCADE. These mainly concern the I/O infrastructure required for physics input/output, intercrate communication, and data recording.

The basic hardware and software platforms on which CASCADE is supported are as follows. The stages should execute in VMEbus crates either on the MC68040-based CES FIC8234 [4] running the OS-9 operating system or on PowerPC-based CES RIO2 8062 running LYNXOS or in workstations running Solaris or SunOS.

Front-end stages may read data from VME and CAMAC under OS-9 and LYNXOS, and from FASTBUS under OS-9. Interfacing to CAMAC is provided via a VME-to-CAMAC branch interface [7] or via a dedicated CAMAC controller connected to VICbus [8] with software support in the form of standard NIM/ESONE/IEEE libraries. A FASTBUS-to-VSB interface [9] provides the connection between a VME crate and FASTBUS. A comprehensive multiuser implementation of the NIM FASTBUS library is available for this module [10] under OS-9. Triggering is possible from CAMAC (lams) and VMEbus (via a general-purpose VME trigger module called CORBO [11]).

To provide a fast channel for data transfers and for the exchange of messages, the VME crates are interconnected via memory-mapped VICbus links [1]. Ethernet is used for remote event monitoring, inter-stage links and less-time critical applications such as run control and file access, via NFS.

The inter-stage communication across VICbus is based on a message exchange library developed at CERN [12], while the communication across Ethernet uses the standard NFS/TCP/IP package available on OS-9, LYNXOS and UNIX. Under OS-9 and LYNXOS, data may be recorded via SCSI, to an IBM3480-compatible STK4280 tape drive [13], to an Exabyte [14], to a DLT [Digital Linear Tape] device, or via Ethernet to a remote disk.

## 1.9 Application-dependent Parts

In summary, the application-dependent parts in a CASCADE-based data-acquisition system are:

- **the event production functions** to be linked with the front-end stages;
- **the event building functions** to be linked with the event builder stages;
- **the monitoring programs** which may attach to stages;
- **the data-flow configuration file** which specifies the functional and physical characteristics of every stage as well as the topology of the system;
- **the run control mini SQL data-base / configuration file** used to describe the run control data structures.
- **the 'start' script files** executed on request of the run control to boot the application "DAQ units" (stages, recorders, mps) in their respective host processors.
- **the error reporting decoder and router files**

Templates are distributed for all the modules listed above together with template makefiles and script files to load and start execution of the CASCADE related processes in the various processors of the application.

## 1.10 Software Distribution

The CASCADE package is distributed as a set of libraries, executable programs and template files of all sorts grouped into an installation kit. The software and the installation procedure can be retrieved directly from the World Wide Web CERN ECP/FEX-CAserver.

## 1.11 Status

CASCADE has been used at CERN by the NOMAD experiment and the Energy Amplifier project since 1994, and by the NA44 experiment in 1995 and 1996. These applications have quite different system configurations and event characteristics. After a continuous program of consolidation and improvements as well as porting to new platforms over the past few years, CASCADE is now a stable product dedicated to the present applications and to the LHCB experiment testbeam activities.

# 2 Required Layered Products

Cascade applications use one or more front-end systems for the data acquisition itself and one or more workstation(s) for run control, remote monitoring. etc. In most cases (and as assumed in this document) one of the workstations is also used as boot and NFS server for the front-end systems. On both the front-end and back-end systems, the use of Cascade requires the availability of a number of products and services.

Therefore, before attempting to install Cascade itself, users should request their system managers to set up:

- the required bootp and NFS client/server as explained in Section 3.2 and Section 3.3
- several network services and associated daemons to be declared to the system
- a number of additional products on top of the operating system.

The following sections present the main required services and products as well as the operating system environment in which they are needed.

## 2.1 Network services and daemons

- The network service file:

  On back-end workstations and front-end LynxOS:/`etc/services`
  On OS-9:`/os9/system/etc/services`

needs to be modified to include the following services:

```
casc-stg    7731/tcp    #CASCADE Inter-Stage comm server
casc-drec   7733/tcp    #CASCADE Disk recorder (REC) server
casc-rmp    7734/tcp    #CASCADE Remote monitoring server
casc-rc     7741/tcp    #CASCADE run control (RC) server
casc-emn    7757/tcp    #CASCADE EMUnet server
msql        7780/tcp    #miniSQL server
```

- On Unix systems, the following daemon entries need to be done in the file `/etc/inetd.conf`:

  On back-end workstations:

```
casc-drec     stream          tcp     nowait          cascade
/usr/local/online/bin/rec_server rec_server
```

  On LynxOS:

```
casc-rmp          stream          tcp     nowait     cascade[1]
/usr/local/online/bin/mp_server mp_server -f /tmp/rmp.log
```

- On back-end workstations, msqld, the mSQL daemon (see reference to mSQL below), has to be started at system startup.

- On OS-9 systems, the program `tcp_daemon` available in the directory `/os9/online/cmds` on the CERN FEX-CA OS-9 cluster has to be launched at system start-up by the following sequence of commands:

```
load -d /os9/online/cmds/mp_server
load -d /os9/online/cmds/tcp_daemon
tcp_daemon casc-rmp mp_server -f /dd/tmp/rmp.log <>>>/nil &
```

## 2.2 System access permissions

The names of the back-end workstations need to be entered in the appropriate files of the front-end systems:

On LynxOS: `.hosts` in the user home directory
On OS-9: `/system/etc/rhosts`

## 2.3 Mandatory Layered Products

The following layered products are required in order to be able to build, run and control any CASCADE application:

| Layered Product | Platform(s) |
|---|---|
| Motif | UNIX workstation |
| miniSQL [a] version 1.xx [2] | UNIX workstation |
| GNU C compiler | OS-9 + LYNXOS + UNIX workstation |
| rsh - remote shell utility [15] | OS-9 + LYNXOS + UNIX workstation |
| Bourne shell | OS-9 + LYNXOS + UNIX workstation |

a. The primary source of information related to miniSQL is the Hughes Technologies Web Site located at : **http://Hughes.com.au/**

1. **Warning**:

Under Unix and therefore LYNXOS, a process can send signals only to processes with the same ownership. Since the mp_server and the stage processes send signals to each other, it is mandatory under LYNXOS that stages accessed by remote monitoring programs have the same owner as the one declared for mp_server in the inetd.conf entry mentioned above.

## 2.4 Optional Layered Products

The following layered products may be needed as a function of the application hardware configuration and of the desired error reporting facilities:

| Layered Product | Platform(s) |
|---|---|
| CERN ECP/FEX-CA I/O drivers and libraries | OS-9 + LYNXOS |
| CES FASTBUS library | OS-9 |
| OCPAW [17] [18] | OS-9 + UNIX workstation |

## 2.5 The CERN ECP/FEX-CA I/O drivers and libraries

The use of CASCADE relies on the availability of specific commercially available hardware for which special software has been developed at CERN when the associated commercial software was inexistent/insufficient and did not match the requirements of the real time data-acquisition environments used at CERN. the ECP/FEX-CA (previously ECP/DS) group together with the ECP/ESS-OS group have developped a number of drivers, libraries and test programs in the OS-9 and LYNXOS environments for interfaces such as the CES VIC 8251F, RCB8047 CORBO, CBD8210 CAMAC branch driver and devices such as Storagetek STK4280 and Exabyte.

No distribution scheme exists for this software and arrangements with ECP/FEX-CA and/or ECP/ESS-OS need to be made to get it. At CERN, a service exists from the ECP/ESS-OS group to purchase, install and test various hardware components and sub-systems such as the ones used by CASCADE applications. The CERN ECP/FEX-CA I/O drivers and libraries are installed into CASCADE systems as part of that service.

**It is highly recommended to use this service and to insure proper functionning of all system components individually before attempting to run any CASCADE application.**

As a guideline, the following test programs available in the '**`tests`**' subdirectory of:

Under OS-9: `/os9/online/generic/<product>`
Under LYNXOS: `/generic/<product>`

should be run succesfully as pre-requisites to any application using the related **`<product>`** hardware/software. These test programs offer a much simpler and more efficient debugging environment than Cascade.

For CAMAC: `slottest` + `camint`
For CORBO: `vmetrig_test1`
For VICbus (product = vmx): the `tstevb` & `testlsc` pair
For Summit (Storagetek 4280): `ctexec`
For Exabyte: `xaexec`
For DLT: `exerciser`
For NIC (Network communications):

On Lynx and workstations: `/usr/local/online/bin/nictest_async`
On OS-9: `/os9/online/cmds/nictest_async`

# 3 Getting the Software

## 3.1 Where to get the software from

The CASCADE package is distributed as a set of libraries, executable programs and template files of all sorts grouped into an installation kit. The software and the installation procedure can be retrieved directly from the World Wide Web via the URL:

**http://cascade.cern.ch/Projects/CASCADE/CASCADEpublic**

Note that to be able to access this area you should first register with Yves Perrin (CERN ECP/FEX-CA, e-mail: Yves.Perrin@cern.ch) giving the name of the machine used for the WWW access (not necessarily the machine where CASCADE will be installed).

## 3.2 Which software to download where

Cascade applications use one or more front-end systems for the data acquisition itself and one or more workstation(s) for run control, remote monitoring. etc. In most cases (and as assumed in this document) one of the workstations is also used as boot and NFS server for the front-end systems. Users should request their system managers to set up the required bootp and NFS services before attempting to install Cascade.

This means that for any application, Cascade must be installed on two different platforms: the back-end workstation and the front-end system. The front-end file base being, generally, a NFS mount point in the back-end file base, it means that two save sets have to be downloaded via WWW into the back-end system.

## 3.3 Setting up the directory infrastructure necessary to host CASCADE

Before downloading any save sets, it is necessary to set up various areas and directories to host the CASCADE product according to the structure expected by the CASCADE scripts and makefiles.

This is accomplished by:

1. On the back-end UNIX workstation, setup the NFS export file and mount point for the NFS-OS-9 file system and/or NFS-LYNX (same owner for the NFS client and the NFS server nodes.
2. Choose and create a **CASCADE "online" area**. The default values are:

   `/usr/local/online` ... Ultrix, HP-UX, Solaris, SunOS.

   `/usr/local/online` ... LynxOS (NFS-based file system).

   `/os9/online` ..... OS-9 (NFS-based file system).

   The online area is used to maintain the online version of CASCADE.

You can have several online areas (for several concurrent versions of CASCADE) as long as they are on separate directory trees or on separate nodes. Be careful to choose the proper "owner" for those directories. It is better NOT to choose root or daemon, as there are some setuid utilities that might open security holes. Choose a non-priviledge user belonging to an appropriate group (normally, group has full access to the online areas). For NFS-based OS-9 and/or LynxOS file systems, the owner should exist and be the same on OS-9 and/or LynxOS and the NFS server node (Ultrix, HP-UX, Solaris or SunOS), or the installation procedure will fail for lack of privileges.

3. On one Unix host (ULTRIX, HP-UX, SOLARIS, SUNOS or LYNXOS) create a **CASCADE "repository" area**. For LynxOS this area must be a directory NFS mounted from one unix host (ULTRIX, HP-UX, SOLARIS, SUNOS). **Make sure it is not read-only**. Here you will unpack the CASCADE distribution kits and (optionally) keep your online reference copy of CASCADE (if you wish to have them). You should have one and only one central common repository area. The owner of this area should be the same as for the "online" tree.

The CASCADE repository area can be anywhere, as long as it is not the same choosen for the "online" tree.

For OS-9 systems, you need a Unix host to unpack and distribute CASCADE. It is possible to have either NFS-based or local-based file systems. In both cases, an intermediate (permanent or temporary) area is required on a Unix host.

If you have several Unix architectures (LYNXOS, HP-UX, SOLARIS and SUNOS) we suggest to export via NFS the repository area to all the nodes where you should install CASCADE online. This way there will be one central repository Unix host and several online Unix hosts where CASCADE should be available.

## 3.4 Downloading of the CASCADE software

After connecting to the Cascade Web site via your favourite WWW browser, go to the CASCADE distribution page and follow the procedure described below.

### Common installation procedure

For all platforms, the first thing to do is to transfer the CASCADE kit files and the basic installation script **into your repository area**:

1. Fetch the architecture-specific files you need. The available kits are:

   `CASCADE.release.ULTRIX.tar`
   `CASCADE.release.SUNOS.tar`
   `CASCADE.release.SOLARISOS.tar`
   `CASCADE.release.LYNX.tar`
   `CASCADE.release.HPOS.tar`
   `CASCADE.release.OS9.tar`

   `CASCADE.release.FULL.tar` (only for CASCADE team internal use)

2. Fetch the installation script

```
CASCADE.release.Install
```

3. Execute the installation script on the repository Unix host:

```
$ cd cascadeRepositoryArea
$ chmod +x CASCADE.release.Install
$ ./CASCADE.release.Install
```

This script unpacks the various products and performs some cleanup on the repository area. You should execute this script once and only once, right after the transfer of the source kits.

IF SOMETHING GOES WRONG during the last step (out of disk space, wrong permissions, invalid directory tree) you should fix the problem and restart FROM POINT 1. The procedure performs several cleanups during the installation itself and recovery on error is not possible. It is NOT possible to run the procedure mentioned in point three above several times.

### Ultrix installation procedure

On all the ULTRIX "online" nodes run the ULTRIX installation script:

```
$ cd cascadeRepositoryArea
$ ./CASCADE.release.ULTRIX.Install UltrixOnlineArea
```

You can run this procedure several times, as you might want to:

- rebuild a corrupted CASCADE online tree
- have several concurrent versions of CASCADE on separate online trees and/or on separate nodes

### SunOS installation procedure

On all the Sun "online" nodes run the SUNOS installation script:

```
$ cd cascadeRepositoryArea
$ ./CASCADE.release.SUNOS.Install SunosOnlineArea
```

You can run this procedure several times, as you might want to:

- rebuild a corrupted CASCADE online tree
- have several concurrent versions of CASCADE on separate online trees and/or on separate nodes

### HP-UX installation procedure

On all the HP-UX "online" nodes run the HP-UX installation script:

```
$ cd cascadeRepositoryArea
```

```
$ ./CASCADE.release.HPOS.Install HposOnlineArea
```

You can run this procedure several times, as you might want to:

- rebuild a corrupted CASCADE online tree
- have several concurrent versions of CASCADE on separate online trees and/or on separate nodes

### Solaris installation procedure

On all the Solaris "online" nodes run the SOLARIS installation script:

```
$ cd cascadeRepositoryArea
$./CASCADE.release.SOLARISOS.Install SolarisOnlineArea
```

You can run this procedure several times, as you might want to:

- rebuild a corrupted CASCADE online tree
- have several concurrent versions of CASCADE on separate online trees and/or on separate nodes

### LynxOS installation procedure

On all the Lynx "online" nodes run the LYNXOS installation script:

```
$ cd cascadeRepositoryArea
$ ./CASCADE.release.LYNX.Install LynxOnlineArea
```

You can run this procedure several times, as you might want to:

- rebuild a corrupted CASCADE online tree
- have several concurrent versions of CASCADE on separate online trees and/or on separate nodes

### OS-9 installation procedure (NFS-based file system)

**1.** On Unix (NFS server host), run the NFS server installation procedure:

```
$ cd cascadeRepositoryArea
$ ./CASCADE.release.OS9.Install nfsMountPoint
```

where nfsMountPoint is the full path to the "online" NFS server export to OS-9.

You can run this procedure several times, as you might want to:

- rebuild a corrupted CASCADE online tree
- have several concurrent versions of CASCADE on separate online trees

You should run this procedure once per NFS mount point.

**2.** Login on any of the OS-9 NFS client(s) and do:

```
$ chd os9OnlineArea
```

```
$ OS9.Install
```

You should run this procedure once per NFS mount point.

You can repeat this procedure several times to restore a corrupted CASCADE OS-9 online tree. You should repeat the above procedure for all the OS-9 "online" trees you want to export (NFS mount points).

## Post-Installation cleanup

1. If you are not interested in online copies of CASCADE, you can delete all the files within the repository area:

```
$ rm -rf repositoryArea/*
```

If you want to delete selected copies of CASCADE from the Unix repository area do the following:

```
$ cd repositoryArea
$ ls CASCADE.*.descriptor
  << list of all releases of CASCADE stored in the
repository area >>
$ ./CASCADE.releaseToDelete.Cleanup
```

Repeat the last step for all the versions of CASCADE you want to delete.

Once a release has been removed, you will have to follow the complete installation procedure to restore corrupted online directory trees. If you decide to keep the files online, you will be able to re-execute at any time the system-specific (ULTRIX, LYNXOS, HP-UX, SOLARIS, SUNOS or OS-9) procedure.

2. On the OS-9 system (NFS and local file systems) you can remove the "ytar" files and the installation scripts from the "online" area.

```
$ chd os9OnlineArea
$ del -f *
```

Ignore all error messages returned by the last statement.

Once these files have been removed, you will have to follow the complete installation procedure to restore a corrupted CASCADE OS-9 directory tree. If you decide to keep these files online, you will be able to re-execute at any time the OS-9 installation procedure.

## TROUBLESHOOTING

1) On Unix (ULTRIX, LynxOS, HP-UX, Solaris or SunOS): if the system-specific installation procedure fails with "can't create" or "permission denied" status, check the protection of the main directories and the relative subdirectories. The protection mask should be at least "u+rwx".

# 4 Building a CASCADE System

Building a CASCADE-based data acquisition system for a given application requires to go through a number of steps described below. However, to help new users to get started and to provide a framework to automate the building and booting of their future applications, a suite of demo script and make files are provided enabling the user to build and run a simple CASCADE demonstration system by typing just a few commands.

## 4.1 The Cascade demo system

The Cascade demonstration example includes a stage, a tape recorder and a disk recorder client running in a front-end system (OS9 or LYNXOS) and the run control, remote monitoring and the disk recorder server running on a back-end Unix workstation.

The front-end stage gets its triggers from a CORBO module and generates random length events filled with incremental data.

Tape recording requires the availability of a STK4280 device called /ct0 under OS-9 or the availability of a DLT device called /dev/cexb5 under LYNXOS.

After having checked that:

- the back-end system has access to the front-end (if necessary edit the front-end `.rhosts` file in the user home directory)
- the environment variable DISPLAY is defined and pointing to the appropriate display

the demonstration can be built and run from the back-end workstation simply by typing:

```
mkdir <demo_directory>
cd <demo_directory>
/usr/local/online/templates/copy_demo fe_os fe_system fe_dir
```

where:

`<demo_directory>` is the directory to host the demo on the back-end

`fe_os` is the type of operating system run on the front-end (OS9 or LYNX)

`fe_system` is the name of the front-end system (e.g. eposly11)

`fe_dir` is the full path of the directory to create for the demo on the front-end

Warning and/or error messages may be displayed during the execution of the demo scripts. Messages such as:

```
rm: lstat /home2/ype/DEMO_LYNX/daqconf.upd: File or directory
doesn't exist
```

or

```
Error unlinking named semaphore NIC_SEM_36_4: Protection
violation
```

result from commands attempting to remove processes or resources which are either already absent or belonging to some other users. If they don't result in the script to be aborted they should be ignored. However, if the script exits prematurely (before the run control panel pops up) fix the problem related to the last error message and then:

- if the problem was during the copying phase, type:
  ```
  demoFromTempl fe_os fe_system fe_dir
  ```
- if the problem was during the building phase, type:
  ```
  demoFromBuild fe_os fe_system fe_dir
  ```
- if the problem was in the run control execution phase, type:
  ```
  demoFromRc
  ```

The EMU error reporting chain can be started automatically (after having given the front-end system access to the back-end via the back-end .rhosts file in the user home directory) by typing:

```
/usr/local/online/templates/emuStart fe_os fe_system fe_UHomeDir
```

where:

`fe_os` is the type of operating system run on the front-end (OS9 or LYNX)

`fe_system` is the name of the front-end system (e.g. eposly11)

`fe_UHomeDir` is the full path of the user home directory on the front-end system

Note:

By default, the demo stage and recorders are not reporting messages via EMU (instead they print in their respective log files). To get messages via EMU, it is therefore necessary to edit the start scripts in the front-end directory to specify:

```
setenv ERRORS_TO_EMU 1
```

## 4.2 Building a Cascade system for a real application

Building a Cascade based data-acquisition for a given application requires to go through the sequence described below.

Once per stage:

- create a user work directory
- copy the distributed templates into the user work directory
- write the event production functions to be linked to the stage (one of the supplied templates can be used for stages with no USER type input ports)
- write the event building functions to be linked to the stage (the supplied template can also be used for stages which do not perform event building operations)
- write any application dependent run control related functions to be linked to the stage (the supplied template can be used if nothing special is required)
- build the stage

- write and build the monitoring programs which may attach to the stage

Once for the whole application:

- write the configuration file which specifies the functional and physical characteristics of every stage as well as the topology of the system
- write the application run control configuration in a mSQL database/flat file
- write a message file and a route file if the user plans to report errors via EMU
- write the script files necessary to load and start the CASCADE related processes in the appropriate processors.

A full description of the library functions to be used and all information necessary to write these application dependent modules are provided in the corresponding chapters of this document. In addition, templates are distributed for all these modules together with template makefiles and script files to load and start execution of the CASCADE related processes in the various processors of the application.

Since the Cascade demo suite provides the necessary templates to start from and the make and script files which partially automate the building and the execution of the application, users are strongly recommended to use the Cascade demo to get familiar with Cascade and possibly even to build their real application (if it includes only one stage and a recorder).

The demo suite scripts automatically:

- copy the necessary files in user work directories (on both the front-end and back-end systems)
- adapt the scripts: start.stage, start.rectape, start.recdisk to the application characteristics: machine names, stage names,
- generate the input file for the run control mSQL data base (democonfig)
- define the required environment variables
- generate the application configuration file (daqconf.upd)
- build the various processes (stage, monitoring programs)
- launch execution of the run control.

When starting from scratch the whole suite is initiated by the **`copy_demo`** script described above in the ’Cascade demo system’ paragragh..

**Warning**:

The user is warned that, after having adapted the templates in a directory, he/she should not run the copy_demo script again in the same directory since that will re-copy the templates and, therefore, overwrite all the modifications that had been done.

However various other scripts (shown in Appendix A) are available to help users in re-executing certain steps of this procedure, starting the procedure from different points, building and executing the application from the modified source files sitting in the current directory.

For example during an application development phase, once the run control has been started, the producer functions might have to be modified, the stage to be rebuilt and

rexecuted. This can be achieved easily by killing all the stage and recorder processes from the run control 'DAQ commands' menu, modifying the userprod source and simply run the **demoBuildLynx** or **demoBuildOs9** script to recompile and rebuild the stage. Selecting the 'Stop run' item in the run control 'DAQ commands' menu will reload the stage and recorders so that a run can be started with the modified stage.

Syntax:

```
demoBuildLynx  fe_system  fe_dir
demoBuildOs9  fe_system  fe_dir
```

It is possible to use different stage and recorder names by specyfing extra parameters to the copy_demo or demoFrom Build scripts.

Example:

```
demoFromBuild fe_os fe_system fe_dir -sstageName -ttapeName -
ddiskName
```

where:

`fe_os` is the type of operating system run on the front-end (OS9 or LYNX)

`fe_system` is the name of the front-end system (e.g. eposly11)

`fe_dir` is the full path of the directory to create for the demo on the front-end

`stageName` is the name to be given to the stage

`tapeName` is the name to be given to the tape recorder

`diskName` is the name to be given to the disk recorder

## 4.3 Event Production and Event Building Function Development

To build a CASCADE stage using `STG_makefile_demo` the user has to provide three modules in relocatable format.

The first is known generically as `userprod` and contains the event producer code. The 'copy_demo' script copies three event producer templates with names of the type: `userprod_xxx_sglev.c` where xxx qualifies the trigger source (signal, corbo, camac). By default, the demo expects a Corbo module to be available in the front-end system and therefore automatically generates a `userprod.c` module which is a copy of `userprod_corbo_sglev.c`. Depending on the type of trigger source to be used in the application the user should start from the most appropriate event producer template and should copy it manually into `userprod.c`.

The second module is called `userevb` and contains the user event building functions.

The third module is called `userctl` and contains dummy functions which, if necessary, should be replaced by actual code to transfer application dependent control and status information between the run control and the stage.

The files `userprod.c`, `userevb.c` and `userctl.c` are compiled in a separate makefile, called `USR_makefile_demo`.

Please note that these examples are entirely based on the event header described in Section 7.3.

As explained in Section 1.2.2, CASCADE supports a number of inter-stage link types so that handling of the event flow between stages is transparent to the user. Therefore, the application dependent event production functions really concern events originating from stage input ports not connected to a stage (called USER type input ports). However, since all stages are built on the same skeleton, they all need to be linked to event productions functions even if those will not be called because none of the stage input port is of type USER. For such stages, it is recommended to use the userprod templates which do not call any 'hardware specific' libraries and which can, thus be compiled and linked on all platforms.

For the same reason user event building functions must be linked to all stages even to those not configured to perform event building. The distributed userevb template can be used directly as it is to be compiled and linked to those stages.

### USR_makefile_demo

This makefile is used to produce the object files for the three user modules needed in the stage makefile: `userprod`, `userevb` and `userctl`. The files can be built, individually or all together, by executing `USR_makefile_demo` with the appropriate target file. For example:

Under OS-9:

```
make -b -f=USR_makefile_demo userprod.r
```

or

```
make -b -f=USR_makefile_demo
```

Under LYNXOS:

```
gmake -f USR_makefile_demo userprod.o
```

or

```
gmake -f USR_makefile_demo
```

will result in the building of the **`userprod`** object file and the three user module object files, respectively.

## 4.4 Stage Generation

A template makefile called `STG_makefile_demo` is provided as an example of how to generate a stage. It possibly needs to be adapted to:

- use the file directory organisation of the application system;

- link to the appropriate I/O libraries (the ones called directly or indirectly by *userprod()*).

Stages can then be built by running the make utility as follows:

Under OS-9: **make -b -f=STG_makefile_demo stage**
Under LYNXOS: **gmake -f STG_makefile_demo stage**

## 4.5 Monitoring programs Development

Users are advised to develop their monitoring programs starting from the template source file and makefile distributed as part of the CASCADE package. These files (`mp_demo.c` and `SH_makefile_demo`) are automatically copied into the user work directory on both the front-end and back-end systems and the local (`mp_demo`) and remote (`rmp_demo`) processes are built when executing the copy_demo script described above. They can also be copied by executing the `copy_templates_demo` script file.

Local (running in the front-end system) monitoring programs for the Cascade demo can be built by doing:

Under OS-9: **make -b -f=SH_makefile_demo mp_demo**
Under LYNXOS: **gmake -f SH_makefile_demo mp_demo**

Remote (running in the back-end system) monitoring programs for the Cascade demo can be built by doing:

Under Unix: **gmake -f SH_makefile_demo rmp_demo**

## 4.6 Execution of the demo monitoring programs

The `mp_demo` local monitoring program is executed by typing:

```
mp_demo  stg_name  0  0  no_events  nwords_dumped  print_rate 0
```

where:

`stg_name` is the name of the stage to retrieve events from
`no_events` is the number of events to monitor before exiting the program
`nwords_dumped` is the number of event words to dump in case of error
`print_rate` is the rate (nb of events) at which a report message has to be displayed

Note:

Events can be printed by running `mp_demo` in interactive mode (start it with only the first parameter)

The `rmp_demo` remote monitoring program is executed by typing:

```
rmp_demo stg_name stg_host 0 no_events nwords_dumped print_rate 0
```

where:

`stg_name` is the name of the stage to retrieve events from

`stg_host` is the (front-end) name of the system where the stage is running

`no_events` is the number of events to monitor before exiting the program

`nwords_dumped` is the number of event words to dump in case of error

`print_rate` is the rate (nb of events) at which a report message has to be displayed

**Warning**:

Since under Unix and therefore LYNXOS, a process can send signals only to processes with the same ownership and since the mp_server and the stage processes send signals to each other, **it is mandatory under LYNXOS that stages accessed by remote monitoring programs have the same owner as the one declared for mp_server in the inetd.conf entry**.

Note:

Events can be printed by running r`mp_demo` in interactive mode (start it with only the first two parameters)

All details on the templates and on how to write and build a monitoring program can be found in Chapter 9 and in Chapter 10.

## 4.7 Configuration and Run Control Files Preparation

The configuration file contains a detailed description of all the stages in a CASCADE configuration including a definition of the parameters for the input and output ports and the event building process. More details on the configuration file can be found in Chapter 13.

The run control file contains a more global description of the CASCADE stage configuration as required by the run control program to allow an orderly execution of the CASCADE processes. More details on the control file can be found in Section 6.3.

Configuration file and control file templates are distributed as parts of the CASCADE package. These files are automatically copied into the user work directory when executing the `copy_templates_demo` script file.

## 4.8 Writing EMU message and route files

If it is planned to report errors from the application dependent modules by using EMU, both a message file and a route file have to be provided. All details on error reporting and on how to write these files can be found in Chapter 12.

## 4.9 Preparing the script files to start the application

As explained in Chapter 5 booting the various data-acquisition processes involved in a CASCADE application requires to prepare a ’start’ script for every such process.’Start’ script templates are distributed with CASCADE and are explained in that chapter.

# 5 Booting a CASCADE System

## 5.1 Introduction

Booting the various data-acquisition elements involved in a CASCADE application is an operation handled by the run control which, therefore, needs to be started first (see Chapter 6). according to information retrieved from a data base and describing the various "DAQ units" (stages, recorders, eventually monitoring programs, etc) involved in the application and controlled by the CASCADE run control. Both the setting of the necessary environment variables and the launching of the processes associated with the DAQ units are done from the run control when it is requested to change the state of one or more DAQ units from ABSENT to STOPPED.

These operations are done by retrieving commands from the run control data base / configuration file to initiate the necessary transitions. In the case of booting the various DAQ units processes, these commands start execution of a shell 'start' script in the DAQ unit target processor. This is achieved by using the remote shell utility (**rsh**) generally available under UNIX and LYNXOS and which has been ported to OS-9 [15]. It is the user responsability to prepare a 'start' script for every DAQ unit.'Start' script templates are provided as part of the Cascade demo system in the Cascade distribution kit. Script files to help in adapting these templates and to run a simple Cascade application are described in Chapter 4.

## 5.2 Stage 'start' Scripts

A Bourne shell script must be written for every stage used in the application. The purpose of this script is to:

- kill any instance of the stage and associated monitoring programs left from a previous session
- release the CASCADE related system resources (e.g shared segments) left from a previous session
- declare the necessary stage environment variables (see below)
- start execution of the stage process

The `start.stage` template should be copied from the online/templates directory into the application work directory (either manually or by executing the `copy_templates_demo` script) and adapted to the application.

Stage 'start' scripts are passed the following six parameters:

1. the name of the directory which contains the stage process and the application configuration file (assumed to be in the same directory)
2. the name of the stage (not the stage process name) to be started
3. the run control host name
4. the run control domain
5. the full pathname of the configuration file
6. the run control status frequency

### 5.2.1 The Stage Environment Variables

#### Buffer space management

Once acquired, events are kept by the stage as long as all monitoring programs which subscribed to events of that type have not seen them or until the event buffer space gets full. When the buffer space becomes full a ’tidyup’ operation is automatically initiated to scan the whole buffer and get rid of all events declared 'removable' (their deletion will not put any of the mp's below its minimum sampling percentage). Deletion of an event takes a significant but fixed amount of time but searching a removable event includes scanning the stage access list and the search time is proportional to the position of the event in the list. For this reason it is important to keep the access list reasonably small and thus to force ’tidyup’ operations at convenient moments. Since conveniency moments vary from one application to an other, two environment variables called **TIDYUP_REF_INPUT** and **TIDYUP_DELAY** are used to specify when in the application cycle, a ’tidyup’ operation should be forced.

TIDYUP_REF_INPUT should specify the name of an input port

TIDYUP_DELAY should specify a delay in seconds.

Triggering the input port specified in TIDYUP_REF_INPUT activates a timer for the number of seconds given in TIDYUP_DELAY. When the timer expires, the function ED_TidyUp gets called. If TIDYUP_DELAY is not defined, null or negative, the function ED_TidyUp is called immediatly after the input signal has occured. For this mechanism to work properly, it is mandatory to label the input ports defined in the DAQCONF file (e.g. INPUT2 = NU2) or at least the one used in TIDYUP_REF_INPUT.

Note: For the NOMAD event builder stage, the environment variable TIDYUP_AT_SYNC can still be used. It will have the same effect has setting TIDYUP_REF_INPUT to SYNC and TIDYUP_DELAY to 0.

#### Event recording format

The environment variable **FZ_EV_PACKING** is used both by the recorder and by its feeder stage. The stage retrieves it from the environment and the recorder gets it from the feeder stage shared segment. For a stage with output ports connected to recorders (type ZEBRA), the events will be formatted according to the FZ specification. If FZ_EV_PACKING is defined in the environment and is non-zero then, the events will be packed into fixed length physical records which will be wriiten on recording device. Otherwise, as before, every event will use an integer number of physical records (at least one) on the recording device.

#### Error reporting

Reporting of errors is done using either printf or EMH_SysMsg calls as set by the environment variable **ERRORS_TO_EMU** :

0 - use printf (default)

1 - use EMU

NB: if EMU is chosen, be sure that emu is started (otherwise the stage process will be stopped by a full pipe).

### Use of DMA in inter-stage links data transfers (only under OS-9)

The environment variable **USE_DMA** allows to control the use of the DMA facility in VICbus inter-stage links data transfers. If USE_DMA is not defined (default) or is set to 0 then the DMA will not be used.

### Event data space address allocation

If the environment variable **ALIGNMENT_RESOLUTION** is defined and is non-zero then the event space is allocated in such a way that the event (in fact its header) starts at an address which is a multiple of ALIGNMENT_RESOLUTION x 32 bit words. This has been implemented to improve the block transfer speed for certain types of inter-stage links. Therefore the value to be given to ALIGNMENT_RESOLUTION depends on the type of link used.

### Stage behaviour tracing (for experts)

When it is defined and set to a non-zero value, the environment variable **TIDYUP_TIMING** activates time stamping during ED_TidyUp operations (requires the presence of a VME CORBO module in the crate housing the CPU on which the stage is running)

When it is defined and set to a non-zero value, the environment variable **PHASES_TIMING** activates time stamping to trace the execution of the stages phases (requires the presence of a VME CORBO module in the crate housing the CPU on which the stage is running).

An ’history’ mechanism has been implemented in CASCADE. At present, it can only be used to keep track of event building operations. Its use is controlled by two environment variables **HRY_ENABLING** and **HRY_SEGMENTSIZE**.

HRY_ENABLING has to be different from 0 to enable the history.

HRY_SEGMENTSIZE must be set to the size (in bytes) to be used for the history shared segment.

## 5.3 Recorder ’start’ Scripts

One Bourne shell script must be written for every recorder used in the application. The purpose of this script is to:

- kill any instance of recorders and associated monitoring programs left from a previous session
- release all CASCADE related system resources (e.g shared segments) left from a previous session
- declare the necessary recorder environment variables (see Chapter 11)
- start execution of the appropriate recorder process (see Chapter 11)

The `start.rectape` and `start.recdisk` templates should be copied from the online/templates directory into the application work directory (either manually or by executing the `copy_templates_demo` script) and adapted to the application.

Recorder 'start' scripts are passed the following six parameters:

1. the name of the directory which contains the application configuration file
2. the name of the recorder (not the recorder process name) to be started
3. the run control host name
4. the run control domain
5. the full pathname of the configuration file
6. the run control status frequency

### 5.3.1 The Recorder Environment Variables

See Chapter 11 on Data Recording

# 6 Run Control

## 6.1 Introduction

The run control facility provided as part of the CASCADE package involves a process called XRC running on a Solaris or HPUX workstation.

**XRC** is a modular, general-purpose control program allowing complex data-acquisition systems to be modeled in an object- oriented way. It is based on software originally designed by the OPAL experiment and adapted in collaboration with NOMAD. Operator interaction with the data-acquisition system is achieved through a X11/Motif layer which provides a run-time configurable graphical interface including menus, dialog boxes and various types of display panels.

XRC is a process controlling data-acquisition (DAQ) units such as stages, recorders, monitoring programs and user-specific processes. Its main purpose is to provide synchronization between various DAQ units and to hold their respective states. Within XRC, each element is described in a uniform way as a Finite State Machine (FSM), an object having a predefined set of allowed states and allowed transitions between these states. Since DAQ units are external to XRC, they are represented by internal FSM correspondents. A hierarchy of internal FSMs can be introduced to control subsets of the entire DAQ system.

Communication between XRC and the DAQ units uses NIC, the CASCADE Network component package based on the TCP/IP protocol. The XRC process is the network server and the DAQ units are the clients, which can connect dynamically to the server. XRC maintains the states of all the data-acquisition components as well as run-time parameters and saves this information at the end of each session. A run control domain is defined by an identifier used by XRC and the connected DAQ units. In a given domain each object is identified by its unique ASCII name. More than one instance of the run control facility may be running at a given time using different addressing domains. For example a full production run can coexist with the test or calibration of a particular sub-detector. A simple but powerful user library allows for easy preparation of user specific DAQ unit software.

## 6.2 Environment Requirements

To be able to exchange messages, XRC and the controlled DAQ units must use a common address known by all cooperative processes. To enable this the following environment variables must be set before starting any run control related process.

`RC_HOST` is the host name of the processor running XRC

`RC_DOMAIN` is an integer value ranging from 0 to 9 used to compute the communication address

The TCP port number is determined by adding the port number associated to the IP service **casc-rc** (usually in **/etc/services**) and the value of RC_DOMAIN**.** Two or more instances of the run control program may be running concurrently provided they run on different domains. The default value for `RC_DOMAIN` is 0. Stage names must be unique

within the same domain. Two stages with the same name can be controlled by two instances of the run control program running on two different domains.

When the XRC program starts, information about the application DAQ is read from a configuration database. This information is used to build the run control data structures such as the FSM definitions, the default state of objects, the state transitions, the elements of the graphical user interface. Run time parameter values are also maintained by XRC.

## 6.3 The CASCADE configuration database

The CASCADE configuration is stored in a mSQL database. mSQL is a lightweight database engine designed to provide fast access to stored data with low memory requirements. mSQL offers a subset of the ANSI SQL specification.

- Creation of a mSQL data base

Assuming the mSQL daemon (msqld) is running, the creation of a new database can be done by executing the following command as root:

```
msqladmin create DatabaseName
```

- Ceation of a mSQL configuration.

This is achieved by invoking the msql utility which accepts interactive SQL commands or a set of SQL commands stored in a script file. This is achieved by typing the following

```
msql DatabaseName < msql-script
```

or even better:

```
msql DatabaseName < msql-script | grep -i error
```

Template configuration files distributed with CASCADE can be used as a starting point to build a user specific configuration.

## 6.4 Starting CASCADE Run Control

The CASCADE Run Control process (XRC) is started by entering the following command:

```
xrc [-p <profile> -e <EMU | SCREEN>] &
```

When XRC starts a default command profile is read ( named **.profile.xrc** in the current directory). The command profile usually contains a list of commands to select a database, load a particular configuration and setup specific operation modes and display. An alternate profile can be specified on the command line using the -p option. By default error messages are output to stderr (SCREEN). They can be redirected to EMU by explicitely specifying the -e EMU option.

After some time (between 5 and 30 seconds depending on the complexity of the configuration) the Run Control top level menu together with a status matrix is displayed .

## 6.5 The CASCADE user interface

Operator interaction with the CASCADE Run Control is achieved through an X11/Motif interface which converts a sequence of menu selections and other graphical interactions into commands which can be interpreted and executed by XRC. All this interaction is mouse-driven and the mouse buttons work as follows:

Mouse-Button-1 (later referenced as MB1) or Left Button

Used for all pulldown and acknowledgment menus. Double-clicking on a matrix cell removes/add that cell to the list of cells to control.

Mouse-Button-2 (later referenced as MB2) or Middle Button:

Normally not used. This button may be used to drag/drop text.

Mouse-Button-3 (later referenced as MB3) or Right Button):

Used for selecting popup menus for control of individual DAQ units/cells. All item-specific choices, e.g. tape special commands and status are in these menus.

The elements of the graphical interface are briefly described:

**a.** Menus and submenus.

Menus/Submenus are opened by clicking them using MB1. A menu will remain open until an item is selected or MB1 is clicked outside of the menu. Menus may contain submenus (indicated by a > symbol to the right of a menu item. One touch operation (press and drag) is also possible for the selection of menu items.

**b.** Status matrices.

A status matrix is a graphical representation of the FSMs in a partition. A status matrix is composed of cells representing the state of the DAQ items known to the Run Control. The colors of the cells indicate their status as follows:

- black Item is ABSENT (dead or not started)
- coral Item is STOPPED.
- yellow Item is READY to start
- green Item is RUNNING and data is arriving.
- cyan Item is IDLE (running but there has been no data recently).
- tan Item has PAUSEd data-taking.
- white Item is DISABLED.An item can be re-enabled by double-clicking its cell
- red Item has returned serious ERROR status
- dark blue A state transition is pending for this item.

Normally you should wait for a transition to complete before attempting another state change for this item. If, however, a state change seems not to proceed as usual you can simply retry it by selecting the same menu option again because the Run Control automatically aborts the current stage change before starting a new one.

**c.** Cell menus.

Select an item from the menu or a submenu in the same way as for menu bar menus (see above) but use MB3 instead of MB1 while pointing to the status cell corresponding to the desired menu.

**d.** Message boxes.

These boxes pop up on the display from time to time and contain informative messages. Sometimes they contain an **OK Seen it** button in which case you must remove it by clicking with MB1 while pointing to the button.

**e.** Dialog boxes.

These boxes are used for displaying and/or updating the main Run Control parameters. The parameters are displayed in various ways such as:

- Numbers or Character Strings.

The input field can be modified using keyboard input. You need to get input focus by pointing to the field and clicking with MB1.

- Toggle buttons.

Used to switch a flag on or off. The button appears depressed when the flag is on and its background is coloured. Use MB1.

- Radio buttons.

So-called because selecting one button from a group causes any other selected buttons to be unselected. Used to choose one from a list of mutually exclusive options. Select options using MB1.

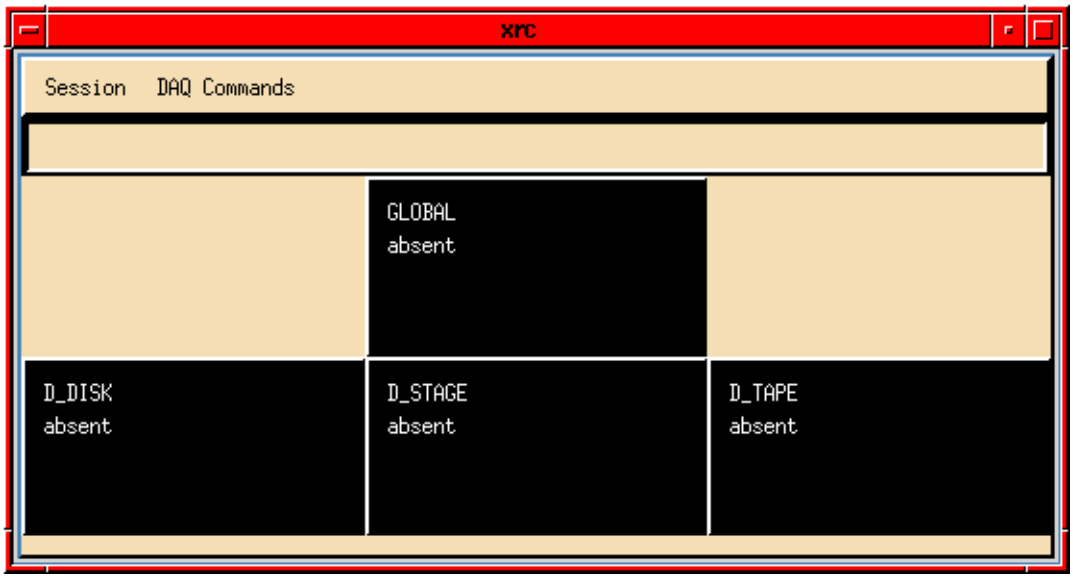
- Option menu.

An alternative to radio buttons. The currently selected option appears on the dialog box. Clicking on it with MB1 pops up a menu containing the other available options. Select an option with MB1.

- Slider scale.

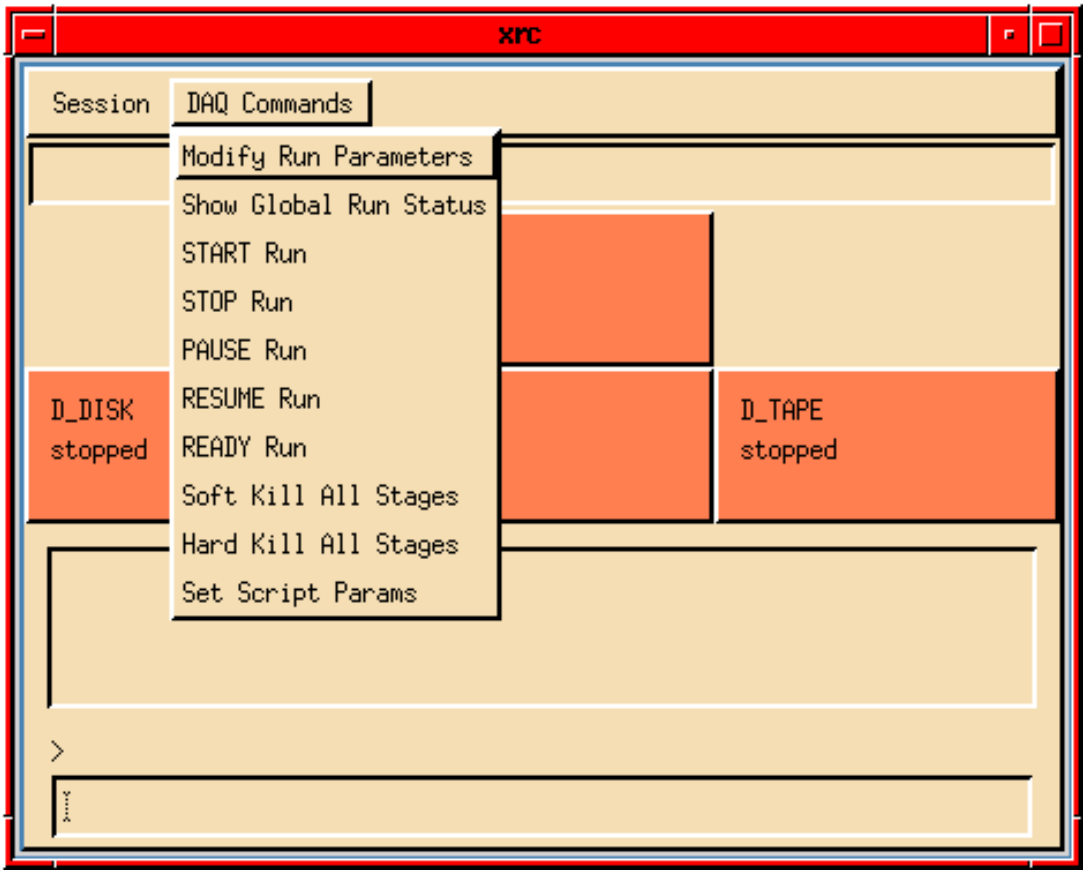
An alternative way of entering numeric information. Change a value by pointing to the slider with MB1 then press and drag to choose a new value. Confirm or cancel the changes by clicking Update or Dismiss with MB1.

When started The CASCADE Run Control displays the following top-level menu together with a status matrix showing the state of the DAQ items for the selected configuration. Most likely all the DAQ items will be ABSENT.




### 6.5.1 The DAQ menu

This menu contains options for managing CASCADE run parameter information and status in general. This menu is activated by clicking the DAQ item and its content is described below.



### a. Modify Run Parameters

This option when selected presents a dialog box consisting in text entry fields for the following variables:


1. Run Type

   This entry field contains a value which is experiment specific and may not be present with all configurations.

2. Run Number

   This entry field contains the number of the CURRENT run and is automatically incremented at Start of Run.

3. Events in Run (Maximum number of)

   Specifies the maximum number of events accepted before the run is automatically stopped. In addition, a run can be stopped, at any time, before this number is reached by selecting the STOP item in the RUN menu (as explained later in this chapter). If the maximum number of events is set to 0, it means that the user does not want the run to stop automatically after a certain number of events have been collected. In that case, the run will only be stopped on operator intervention (as explained above) or if data recording is done it will also be automatically stopped when the end of tape is reached (see Chapter 11).

4. Data Recording

   Valid options are presented in an option menu (One and only one is active at any moment)

   | | |
   |---|---|
   | ***None*** | Events are not stored on a storage medium |
   | ***Tape*** | Events are recorded on the current tape device only |
   | ***Disk*** | Events are recorded on disk only |

***Both*** Events are recorded both on tape and disk

When the **Update** button is selected the parameters visible on the **Run Parameters** dialog box are sent to the stages which are currently enabled (The ones with a non white cell).

**b. Show Global Run Status**

This option presents status information about the various DAQ stages and recorders.


**c. .START Run**

When the **START Run** option is selected, a start run message is sent to all the stages that are currently ENABLED. This message carries the list of Run Parameters and their values. If ALL stages replied with a good status, the status of the corresponding cells will change to READY and then to RUNNING. If a stage return a bad status on START Run command then its corresponding cell will change to ERROR. The reason of the error will be displayed in the EMU display or in the stage log file if EMU is not enabled.

**d. STOP Run**

When the **STOP Run** option is selected, a stop run message is sent to all the stages that are currently ENABLED. If ALL stages replied with a good status, the status of OPPED. If a stage return a bad status on START Run command then its corresponding cell will change to ERROR. The reason of the error will be displayed in the EMU display or in the stage log file if EMU is not enabled.

The end of run sequence in a multi-stage CASCADE application is somewhat complex in the sense that stages switch to the 'stopped' state only when:

- the hardware triggers have been disabled for all the stage USER input ports
- all events of the current burst have been acquired
- all events buffered in upstream stages have been collected, dispatched and acknowledged by all the stage output ports

- all the stage network based inter-stage link connections have been closed
- all the recorders attached to the stage have written a double file mark on the tape

This sequence is generally fast enough to give the impression that stages stop immediately. However, in case of complex configurations with large buffering capabilities, propagation of the stage state transitions can be noticed by seeing the stage cells in dark blue for some time.

**e. PAUSE Run**

**f. RESUME Run**

These two option are dummy.

**g. Soft Kill All Stages**

When the **Soft kill all Stages** option is selected, an abort message is sent to all the stage which are currently ENABLED. After having done a general cleanup all the stages will stop and exit. As a consequence all the stage cells will change to ABSENT. If for some reason one or more stages do not react to this command it could be necessary to use the Hard kill All stages option described below.

**h. Hard Kill All Stages**

When the **Kill all Stages** option is selected, a QUIT signal is sent to all the stage which are currently ENABLE using the remote shell utility. After having done a general cleanup all the stages will stop and exit. As a consequence all the stage cells will change to ABSENT. If for some reason one or more stages do not react to this command it could be necessary to use the Hard kill All stages option described below.

**i. Set Script Params**

This option when selected presents a list od **Read-Only** parameters used for the current active configuration:

1. Run Control  Domain

   The value of the RC_DOMAIN environment variable.
2. Status Frequency Update

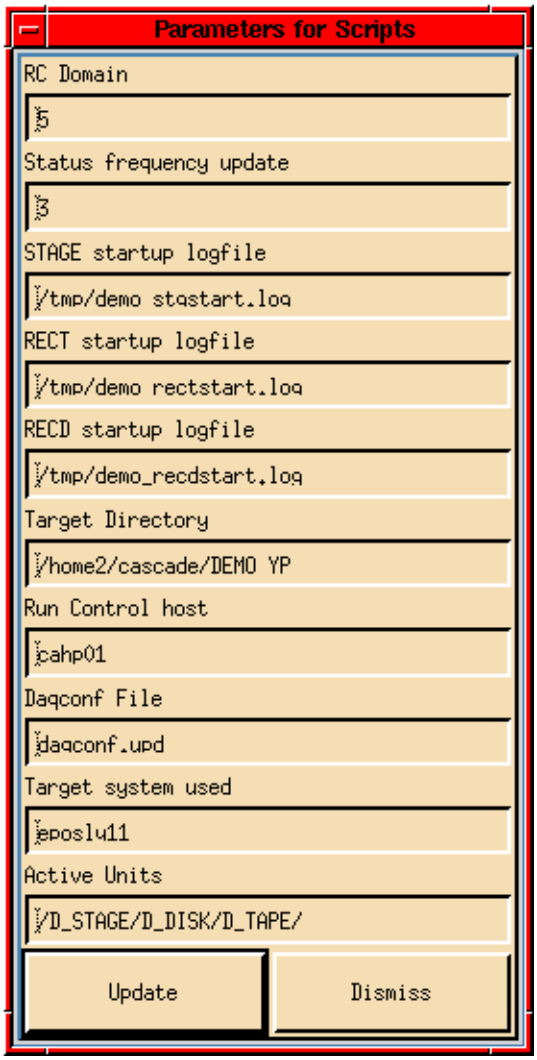
   This integer value is the number of seconds between 2 status messages update.
3. Startup Logfiles

   The full path of a  file (one per DAQ unit) containing a log of the startup events.).
4. Target Directory

   The full path of the application directory on the target system (Front-End).
5. Target System Used

   The hostname of the target system (Front-End).
6. Daqconf File

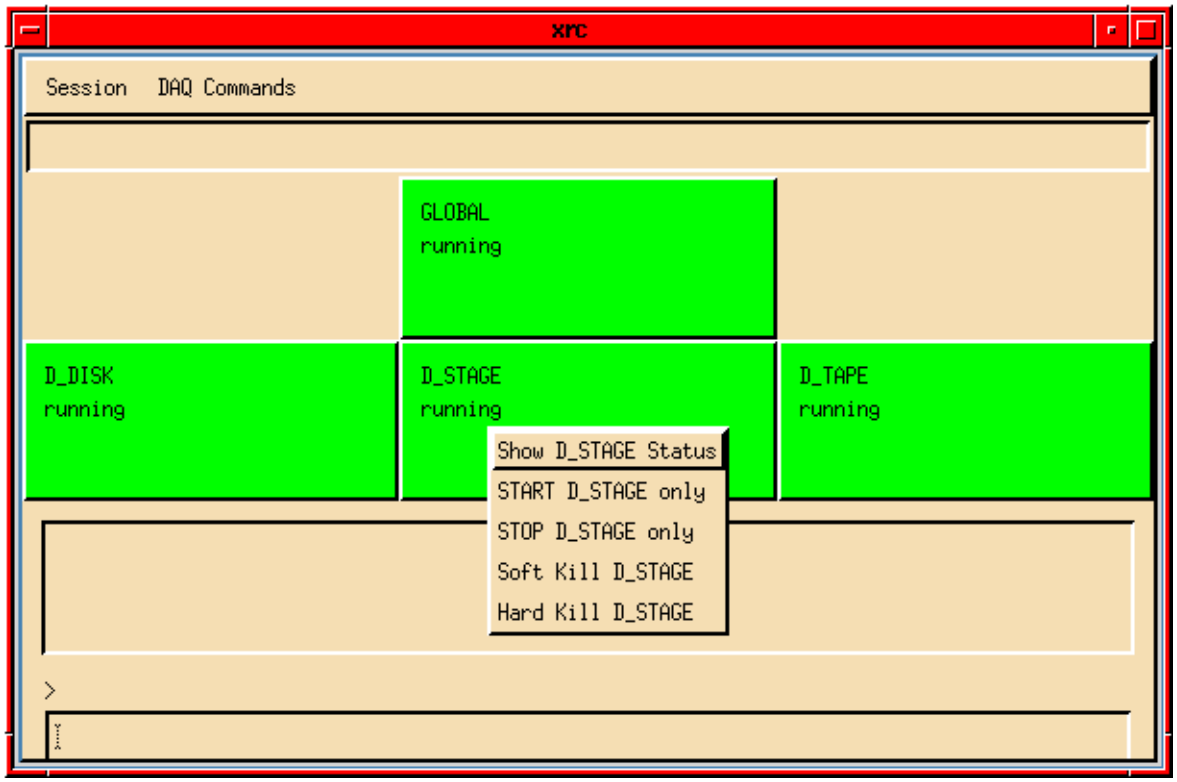
   The name of the CASCADE Configuration file common to all DAQ units.



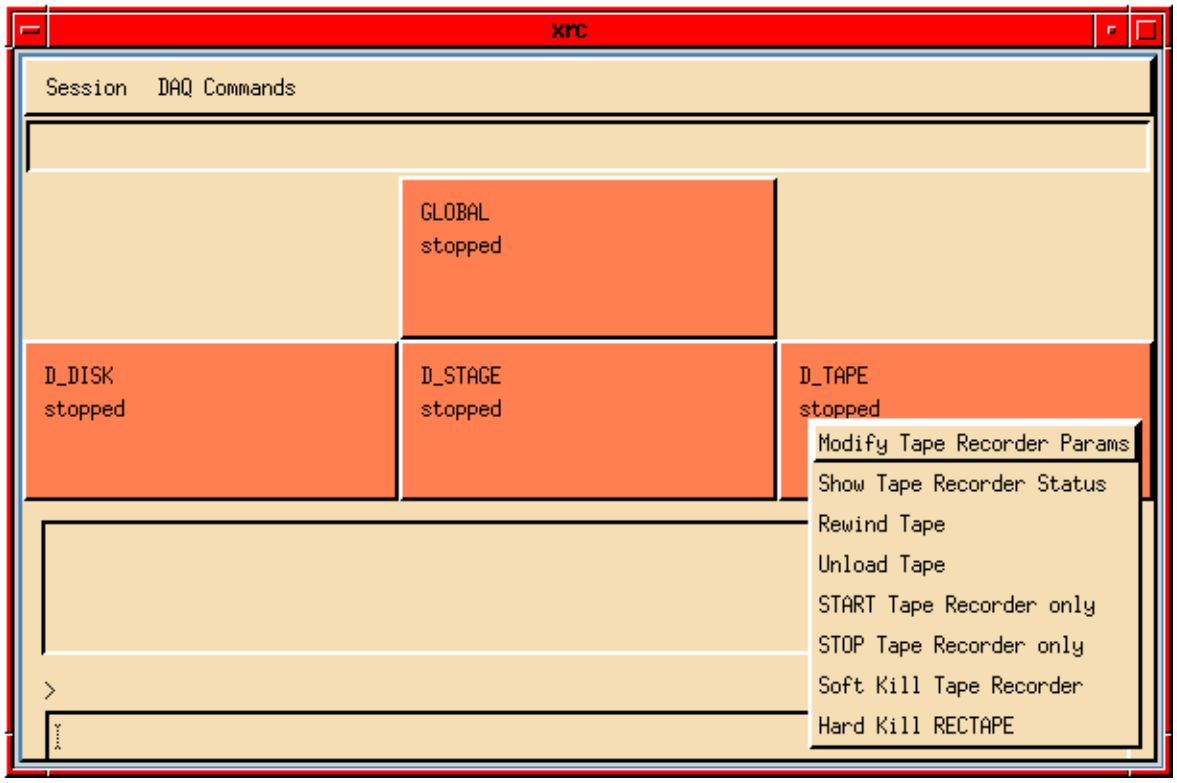

### 6.5.2 The individual stage pop-up menus

To activate a stage/recoder individual pop-up menu, the corresponding cell must be selected and MB3 should be clicked.

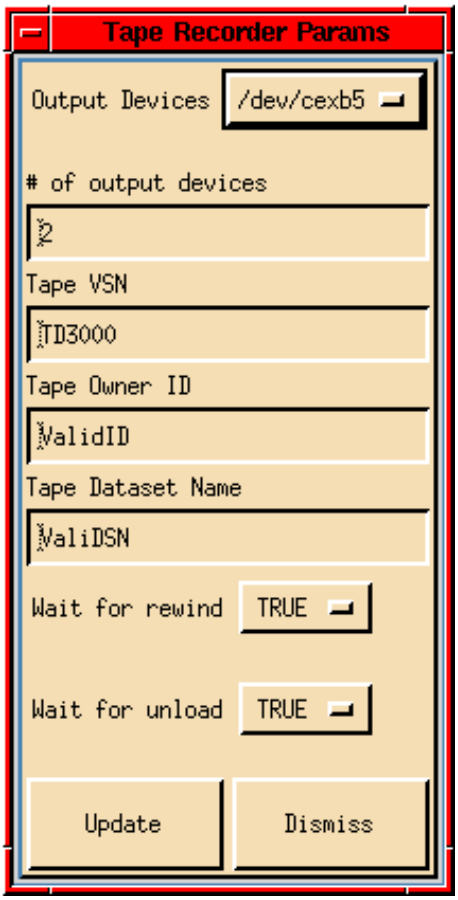
**a. STAGE pop-up menu:**




**b. RECORDER pop-up menu:**




**c. Recorder Parameters panel**




This option when selected presents a dialog box consisting in text entry fields for the following variables:

Output Devives

This option contains a list of the names of potentially used recording devices.

Number of Output Devices

An integer value representing the number of potentially used recording devices.

Tape VSN (Volume Serial Number)

This entry field contains a 6 character string that should match the VSN written in the prelabel of the loaded tape.

Tape Owner ID

This entry field contains a 10 character string which is the identification of the owner of the tape. This information is just written in the label of the loaded tape but is not processed.

Tape Dataset Name

This variable is not user editable and is updated by the recorder.

When the **Update** button is selected the parameters visible on the Tape recorder dialog box will be sent to the stages currently ENABLED when the next START Run command will be issued.

### d. REWIND Tape

When the `Rewind' option is selected, a rewind message is sent to the list of stages specified under the REWIND Option in the configuration file, in the order they appear. If a stage returns a bad status on Rewind action...

### e. UNLOAD Tape

When the `Unload' option is selected, an unload message is sent to the list of stages specified under the UNLOAD Option in the configuration file, in order they appear. If one or more stages is ABSENT the Start command is not sent.

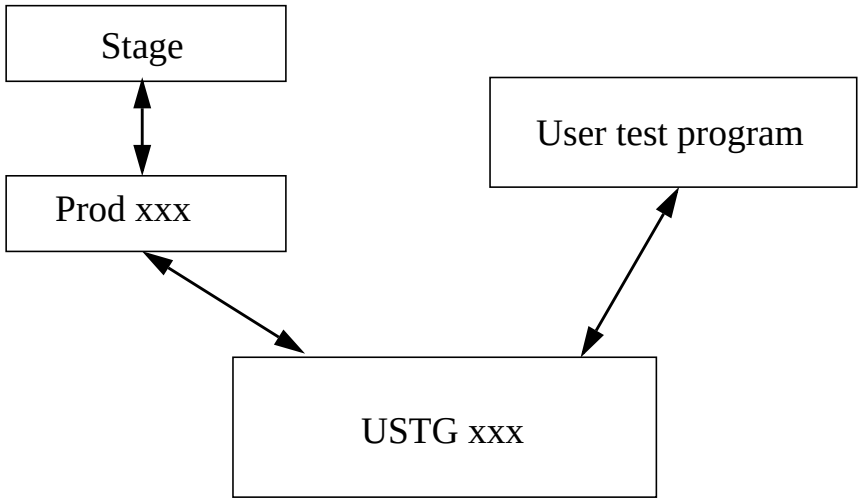
# 7 Event Production

## 7.1 Introduction

This chapter describes the library functions and the template files which are distributed with CASCADE in order to help the user with the development of the application dependent event production software.

The functions for reading event data are divided into two groups. Routines in the first group are called directly from the user independent part of the stage and have **names of the form Prod.....** Inversely, they call functions in the stage to reserve space for event data, validating event data etc. These latter functions **have names starting with STG**.

It is proposed that most of the experiment dependent code to handle event trigger and data readout should be implemented in the form of another set of functions callable from the routines in the first group. All the functions in the second group have **names starting with USTG**. The interface defined by the **USTG** functions provides a separation between code “close” to the stage and the application dependent readout code.and allows to develop test programs independently of the stage, see figure below. **It is strongly recommended that the user code be tested independently before integration in the complex environment of the stage where debugging is more difficult**.




## 7.2 Programming Considerations

To avoid certain problems in the user library, related to reentrancy, the user needs to have some knowledge of the structure of the stage program. The stage is logically divided into phases, implemented in the form of threads the execution of which is controlled by a CASCADE scheduler. The threads are activated by signals and usually

execute sequentially. A thread may, however, suspend itself due to a lack of resources e.g. of memory space in which case the scheduler performs a “context” switch and activates another thread. Since the threads are part of the same process the context switch does not include saving & restoring of global variables (under OS-9 variables addressed relative to A6). It is, therefore, essential that each thread - or more precisely each instance of a thread (petal) - does not share global variables with other parts (threads) of the stage. It should be pointed out that each instance of an input thread is identified by a port number as indicated by the presence of the port parameter in the user routines.

As far as the input part of the stage is concerned it is important to realise that the user library routines may be executed as parts of different threads or in other words that the user code is “shared” between the threads corresponding to the different input ports. When an input thread is suspended another instance of the input thread, corresponding to another port, may be scheduled due to the occurrence of a signal (different from the one which triggered the first thread) which implies that the user code is re-entered possibly resulting in some interference for example at level of shared global variables. This second thread may - or may not - be suspended depending on its space requirements.

However, these reentrancy problems are alleviated by the fact that input threads can only be suspended at two well-defined places: when calling *STG_GetSpace()* to obtain space for the event header, data and trailer and when calling *STG_DeclEvent()* which reserves space for an event descriptor. An input thread does not “loose” its global variables from the moment the thread execution is started by the scheduler and until the first call to *STG_GetSpace()*. Similarly, the execution is not interrupted between *STG_GetSpace()* and *STG_DeclEvent()* and from this point until the exit of the thread. As a consequence, a thread may loose its global variables when “crossing” these execution points but not “in between”. To avoid these problems, global variables should be given the dimension of the number of ports such that the threads do not share global variables.

However, for variables which are not defined explicitly by the user this may present a problem. As an example, if the FASTBUS library is called this latter may contain global data which the user is not aware of and cannot control. In that case calls to external libraries may have to be grouped such that they do not cross the “critical” points. In the case of NOMAD, data from FASTBUS are read into buffers declared in the user library. These buffers should, therefore, be dimensioned as the number of input ports or allocated dynamically and ALL FASTBUS calls performed before calling *STG_GetSpace()*. The unpacking, decoding and construction of LSC events which don't require access to FASTBUS could be done between the calls to *STG_GetSpace()* and *STG_DeclEvent()*.

To summarise:

- input threads may suspend themselves due to lack of space when calling *STG_GetSpace()* and *STG_DeclEvent()*;
- the user library code may be re-entered, potentially resulting in a “loss” of global variables by a thread (petal);
- to avoid this problem all user declared global variables should be dimensioned according to the number of input ports;

- all functions to external libraries which may contain global data should be grouped in noninterruptable sequences i.e either between or outside calls to *STG_GetSpace()* and *STG_DeclEvent()*;
- in some experiments it may be possible - depending on the read-out architecture and the triggering system - to serialise the input triggers at the hardware level. The ORed busy signal from the inputs may be employed to inhibit input signals until the current input thread has finished execution. In this case the deadtime may increase since the input threads cannot execute in parallel. In the case of NOMAD - for example - reading event data into local buffers and splitting of burst data will be serialised;
- if the physics input on different ports are not logically related it may be simpler to use several stages.

## 7.3 Event headers and event production templates

In CASCADE, the format and contents of event headers and trailers are user defined. The lengths are specified in the configuration file and the header and trailer data filled in the event production functions.

All the templates mentioned in this document are based on a common event header format. This is required for event building and also has the advantage of allowing systematic and general checks on the event data to be performed on the events as they flow through the stages. The event header format as defined in the template file eventheader.h is the following:

| Word # | Contents | Used by |
|---|---|---|
| 1 | total event size (header + data + trailer) | |
| 2 | event type (general) | |
| 3 | event number | |
| 4 | event marker (e.g. LSC or EVB) | |
| 5 | header size | |
| 6 | burst type (e.g. NU1,MUON,NU2,CALI) | event builder |
| 7 | event serial number within burst | event builder |
| 8 | number of events in burst | event builder |
| 9 | port number | |
| 10 | error code | |
| 11 | LSC crate number | |
| 12 | CRC | future CRC checks |

Depending on the type of stage, certain fields may not be meaningful.

## 7.4 User Routines - First Group

In this section we describe the functions in the first group of routines (see Section 7.1) which are called from the stage. These routines are part of a file with the generic name

`userprod_xxx.c`. This file is compiled in the makefile USR_makefile described in Chapter 4. Examples of such routines are distributed with CASCADE in userprod template files.

- `userprod_signal_sglev.c` is a simple example in which no external hardware is required. The event signals are generated by software (for example using the sch_send program) and the events are of random length with incremental data (1,2,3....). The event length is defined by the environment variables `CASMINEVSIZE` and `CASMAXEVSIZE` with default values 0 and 700 respectively. This template can only handle one user input port.
- `userprod_corbo_sglev.c` is an example where triggers are provided via a CORBO VME interrupt module with each trigger generating a single CASCADE event. A maximum of four input ports, each corresponding to one CORBO channel may be defined in the configuration file. The event data is incremental (1,2,3....) with a random length defined by the environment variables `CASMINEVSIZE` and `CASMAXEVSIZE` with default values 0 and 700 respectively.Header words one to five are filled with “standard” header information while header words 6-12 are filled with mostly dummy data.
- `userprod_camac_sglev.c` is an example where triggers are provided via a CAMAC Borer 1802 Dataway Display Module in the form of LAMs with each trigger generating a single CASCADE event. Only one input port is defined since the Borer 1802 can only generate a single LAM but the code is designed to handle more ports. The event data is incremental (1,2,3....) with a random length defined by the environment variables `CASMINEVSIZE` and `CASMAXEVSIZE` with default values 0 and 700 respectively.Header words one to five are filled with “standard” header information while eader words 6-12 are filled with mostly dummy data.

The functions in the first group which all have names starting with 'Prod' are called in the following sequence by the stage:

**Initialisation:**

ProdStageInit
ProdPortInit(port) for each port

**For each run:**

ProdStageStartRun
ProdPortStartRun(port) for each port
.
.
.
ProdPortStopRun(port) for each port
ProdPortStopRun

**For each event:**

ProdInput

## ProdStageInit

### Synopsis

```
#include userprod_intf.h

int ProdStageInit()
```

### Description

*ProdStageInit()* is called for global, port independent initialisation of the stage. It calls the function *USTG_StageInit()*, described below.

### Cross-references

*USTG_StageInit()*

## ProdPortInit

### Synopsis

```
#include userprod_intf.h

int ProdPortInit (int port, int signing)
```

### Parameters

| | | |
|---|---|---|
| port | IN | stage port number |
| signum | IN | number of the signal associated with event triggers |

### Description

This function is called once (per port) in the lifetime of the stage and should perform a global initialization of the readout system for the stage input port defined by port. This includes initialisation of the event triggers i.e. linking the occurrence of event triggers to the generation of the signal defined by signum. The application dependent initialisation is performed in the function *USTG_PortInit()*, described below.

### Cross-references

*USTG_PortInit()*

## ProdStageStartRun

### Synopsis

```
#include userprod_intf.h

int ProdStageStartRun (int run_number, int run_type)
```

### Parameters

| | | |
|---|---|---|
| run_number | IN | run number |
| run_type | IN | run type |

### Description

*ProdStageStartRun()* is called once per run with the run number and run type as parameters (as defined by the run control). It should return the value EVENT if the function has produced an event, otherwise NO_EVENT. *ProdStageStartRun()* calls *USTG_StageStartRun()*, described below.

### Returns

| | |
|---|---|
| [EVENT] | an event was produced (*STG_DeclEvent()* called) |
| [NO_EVENT] | no event has been produced |

### Cross-references

*USTG_StageStartRun()*

## ProdPortStartRun

### Synopsis

```
#include userprod_intf.h

int ProdPortStartRun (int port, int* trigStatus)
```

### Parameters

| | | |
|---|---|---|
| port | IN | stage port number |
| trigStatus | OUT | trigger status |

### Description

*ProdPortStartRun()* is called after *ProdStageStartRun()* each time a run is started and should perform any initialisation of the readout or trigger system for the stage input port defined by port. This function enables triggers and returns in trigStatus the value TRIG_ENABLED if successful and TRIG_DISABLED otherwise. *ProdPortStartRun()* calls *USTG_PortStartRun()*, described below.

### Returns

*ProdPortStartRun()* SHOULD NOT produce an event. It returns a dummy value

### Cross-references

*USTG_PortStartRun()*

## ProdPortStopRun

### Synopsis

```
#include userprod_intf.h

int ProdPortStopRun (int port, int* trigStatus)
```

### Parameters

| | | |
|---|---|---|
| port | IN | stage port number |
| trigStatus | OUT | trigger status |

### Description

*ProdPortStopRun()* is called each time a run is stopped and should perform any operation required on the readout or trigger system on the stage input port defined by port. It normally disables hardware triggers. If the last event in the run has been processed it should return the value `TRIG_DISABLED` to indicate that hardware as well as software triggers are disabled. If more events may be produced, the value `TR_HW_DISABLED` should be returned to indicate that more events will be produced. This latter case may happen if a stop run command is received when a burst of events are produced. *ProdPortStopRun()* should return the value [EVENT] if the function has produced an event, otherwise [NO_EVENT]. *ProdPortStopRun()* calls *USTG_PortStopRun()*, described below.

### Returns

| | |
|---|---|
| [EVENT] | an event was produced (*STG_DeclEvent()* called) |
| [NO_EVENT] | no event has been produced |

### Cross-references

*USTG_PortStopRun()*

## ProdStageStopRun

### Synopsis

```
#include userprod_intf.h

int ProdStageStopRun ()
```

### Description

This function is called after *ProdPortStopRun()* each time a run is stopped and should perform any port independent operation required on the readout or trigger system. It should return the value [EVENT] if the function has produced an event, otherwise [NO_EVENT]. This function calls *USTG_StageStopRun()*, described below.

### Returns

| | |
|---|---|
| [EVENT] | an event was produced (*STG_DeclEvent()* called) |
| [NO_EVENT] | no event has been produced |

### Cross-references

*USTG_StageStopRun()*

## ProdInput

### Synopsis

```
#include userprod_intf.h

int ProdInput (int port, int* trigStatus)
```

### Parameters

| | | |
|---|---|---|
| port | IN | stage port number |
| trigStatus | OUT | trigger status |

### Description

*ProdInput()* is called each time an event trigger occurs and is responsible for reading the event data. It typically executes the following sequence of operations:

- disable trigger
- analyse trigger (in a general sense, see *USTG_EvTrgAna()*)
- reserve space for event data in CASCADE buffer (*STG_GetSpace()*)
- transfer data into this buffer
- ' declare' event i.e. notify CASCADE that the event is ready (*STG_DeclEvent()*)

The variable trigStatus should return the trigger status as computed by *ProdInput()*. If triggers are still enabled, the value `TRIG_ENABLED` should be returned. If hardware triggers are disabled, but software triggered events may still be produced, the value `TR_HW_DISABLED` should be returned. If hardware as well as software triggers are disabled i.e. the last event has been produced, the value returned should be `TRIG_DISABLED`. *ProdInput()* should return the value [EVENT] if the function has produced an event, otherwise [NO_EVENT]. *ProdInput()* calls *USTG_InputEvent()*, described below.

### Returns

| | |
|---|---|
| [EVENT] | an event was produced (*STG_DeclEvent()* called) |
| [NO_EVENT] | no event has been produced |

### Cross-references

*USTG_EvTrgAna(), STG_GetSpace(), USTG_InputEvent(), STG_DeclEvent()*

### 7.4.1 CASCADE 'event' functions

These functions are called from *ProdInput()* to reserve space for events and to inject events into CASCADE.

## STG_GetSpace

### Synopsis

```
#include usin.h

int STG_GetSpace (int maxSize, int **dataAddress, int **hdAddress,int
**trAddress)
```

### Parameters

| | | |
|---|---|---|
| maxSize | IN | total number of 32-bit words to allocate for this event |
| dataAddress | OUT | pointer to the event data buffer |
| hdAddress | OUT | pointer to the event header buffer |
| trAddress | OUT | pointer to the event trailer buffer |

### Description

Requests CASCADE to allocate space for event header, trailer and event data:

- allocate maxSize 32-bit words to store event data and return the address of the allocated area in dataAddress.
- allocate hdSize 32-bit words to store event header and return the address of the allocated area in hdAddress
- allocate trSize 32-bit words to store event trailer and return the address of the allocated area in trAddress

### Cross-references

## STG_ReleaseSpace

### Synopsis

```
#include usin.h

int STG_ReleaseSpace (int maxSize, int *dataAddress, int
*hdAddress,int *trAddress)
```

### Parameters

| | | |
|---|---|---|
| maxSize | IN | total number of 32-bit words requested when calling STG_GetSpace |
| dataAddress | OUT | pointer to the event data buffer |
| hdAddress | OUT | pointer to the event header buffer |
| trAddress | OUT | pointer to the event trailer buffer |

### Description

Requests CASCADE to immediately release the space previously allocated with STG_GetSpace.

Note that this function should be called **only if STG_DeclEvent is not called** after space has been obtained by a call to STG_GetSpace (in other words if an event is not created). In the ’normal’ case where an event is declared, the allocated space is released automatically when the event is no longer used in the stage.

### Cross-references

*STG_GetSpace()*

## STG_DeclEvent

### Synopsis

```
#include usin.h

int STG_DeclEvent(int type, int port, int actualSize, int
requestedSize, int *dataAddress,int *hdAddress, int *trAddress)
```

### Parameters

| | | |
|---|---|---|
| type | IN | event type |
| port | IN | port number |
| actualSize | IN | actual event size |
| requestedSize | IN | max event size as requested by *STG_GetSpace()* |
| dataAddress | IN | pointer to the event data buffer |
| hdAddress | IN | pointer to the event header buffer |
| trAddress | IN | pointer to the event trailer buffer |

### Description

This function declares the event to CASCADE which:

- creates and initialises an event descriptor
- fills the event header with the event actualSize, type and source
- fills the event trailer with the event actualSize
- updates the stage event number
- fills the event descriptor
- adds the event descriptor to the stage input list

### Cross-references

*STG_GetSpace()*

## STG_SetEvAttr

### Synopsis

```
#include usin.h

int STG_SetEvAttr (void *evp, int outinhibit, int otherattr)
```

### Parameters

| | | |
|---|---|---|
| evp | IN | pointer to event (as returned by *STG_DeclEvent()*) |
| outinhibit | IN | output inhibit flag |
| otherattr | IN | output inhibit mask |

### Description

If outinhibit is equal to zero the event is sent to all the output ports (ie. not inhibited).

If outinhibit is different from zero, the event is sent to the ports defined by the parameter otherattr in the form of a bit mask. If bit #n in otherattr is equal to zero, output of the event is inhibited for port #n. If bit #n is equal to one, the event is output to port #n ( ie. the output inhibit is overridden). In other words, if otherattr is equal to zero, the event is not sent to any output port.The event is identified by evp as returned by a previous call to *STG_DeclEvent()*.

### Warning

The ports are numbered from zero. Port #n corresponds to output port #(n-1) in daqconf where ports are numbered starting from one.

### Comment

- This function is usually called immediately after *STG_DeclEvent()*.

### Cross-references

*STG_DeclEvent()*

## STG_GetRunState

### Synopsis

```
#include usin.h

int STG_GetRunState (int *runstate)
```

### Parameters

| | | |
|---|---|---|
| runstate | OUT | current run state |

### Description

The current state of the run is returned in runstate

### Cross-references

## STG_GetPortPriority

### Synopsis

```
#include usin.h

int STG_GetPortPriority (int port, int * priority)
```

### Parameters

| | | |
|---|---|---|
| port | IN | stage port number |
| priority | OUT | priority of thread associated with input port |

### Description

The current value of the priority of the input thread defined by port is returned in priority.

### Cross-references

## STG_SetPortPriority

### Synopsis

```
#include usin.h

int STG_SetPortPriority (int port, int priority)
```

### Parameters

| | | |
|---|---|---|
| port | IN | stage port number |
| priority | IN | priority of input thread |

### Description

This function allows the user to change the priority of a thread associated with an input port. It is normally used in case of potential overlapping of trigger bursts at two different input ports. In such a case, changing the relative priority of the threads associated with the ports guarantees that a burst is completely handled at the highest priority port before considering triggers at the other port, thus preventing event interleaving.

When the value of the priority is increased, the thread gets a lower priority. At present, all the input threads have a priority of 106. Since the input threads should always have less priority than the other threads of the stage, the user should never assign a priority value lower than 106 to any of the input ports.

### Comments

Check with a CASCADE expert before using this function!

### Cross-references

*STG_GetPortPriority()*

## Remark about event sizes

The idea is that the user will read the event data into a buffer allocated by the stage. This allocation is done by the function *STG_GetSpace()* which should be called by the user before reading the event. In some cases, the actual size of the event is unknown before the event is read and therefore the space allocation must be done for an upper limit that the event could not reach. The purpose of the function *USTG_MaxEventSize()* is to return this upper limit so that it can be passed to the function *STG_GetSpace()*.

Once the event has been read and its actual size is known, the actual size must be passed to the function *STG_DeclEvent()* so that it can be stored in the event descriptor to enable event consumers to handle the right amount of data words and forget about the remaining invalid words.

## 7.5 User Routines - Second Group

This section describes the routines which contain the major part of the user specific code to handle event triggers and event data. As previously pointed out, the names and parameters of these routines may be changed as a function of the application.

In addition, this second group of user routines contains one routine called *USTG_FZuserVec()* which, as opposed to the others, is not called from the stage input thread but from the dispatch thread and only if the stage is linked to a recorder. This routine, which is also application dependent, has been grouped with the others for reasons of convenience. *USTG_FZuserVec()* must fill the user vector part of the ZEBRA FZ header associated with every event. The format of the FZ user vector used in the examples distributed with CASCADE is described below. The name and the parameters of this routine should not be changed.

Examples of user routines are distributed with CASCADE in userprod template files. They are written in C and compiled in makefile, `USR_makefile` described in Chapter 4. They are descibed in more detail in section 7.4.

## 7.6 ZEBRA FZ User Vector used in the distributed templates

The ZEBRA FZ header includes an application dependent “user vector”. Specification and filling of this vector has to be done in the function *USTG_FZuserVec()*. All the event production templates distributed with CASCADE are based on the following user vector format:

| Word # | Contents |
|---|---|
| 1 | run number |
| 2 | event number |
| 3 | date |
| 4 | time |
| 5 | 5 |

| Word # | Contents |
|---|---|
| 6 | 6 |
| 7 | event type |
| 8 | 8 |
| 9 | 9 |
| 10 | 10 |
| 11 | 11 |
| 12 | 12 |
| 13 | 13 |
| 14 | 14 |
| 15 | 15 |
| 16 | 16 |
| 17 | 17 |
| 18 | 18 |
| 19 | 19 |
| 20 | 20 |

Full details on the ZEBRA FZ header itself can be found in RRR.

## USTG_StageInit

### Synopsis

```
int USTG_StageInit()
```

### Description

*USTG_StageInit()* is called by *ProdStageInit()* and should perform global stage initialisation.

### Cross-references

*ProdStageInit()*

## USTG_PortInit

### Synopsis

```
int USTG_PortInit (int port, int signum)
```

### Parameters

| | | |
|---|---|---|
| port | IN | stage port number |
| signum | IN | signal number associated with event triggers |

### Description

This function is called from *ProdPortInit()* and should initialise the event readout, including triggers for the specified port.

### Cross-references

*ProdPortInit()*

## USTG_StageStartRun

### Synopsis

```
int USTG_StageStartRun (int run_number, int run_type)
```

### Parameters

| | | |
|---|---|---|
| run_number | IN | run_number |
| run_type | IN | run_type |

### Description

*USTG_StageStartRun()* is called from *ProdStageStartRun()* and should perform the appropriate port independent initialisation for each run.

### Cross-references

*ProdStageStartRun()*

## USTG_PortStartRun

### Synopsis

```
int USTG_PortStartRun (int port)
```

### Parameters

| | | |
|---|---|---|
| port | IN | stage port number |

### Description

This function is called from *ProdPortStartRun()* and should perform port dependent initialisation on the port defined by port.

### Cross-references

*ProdPortStartRun()*

## USTG_StageStopRun

### Synopsis

```
int USTG_StageStopRun ()
```

### Description

This function is called from *ProdStageStopRun()* and should perform port independent operations on the readout or trigger system required when the run is stopped.

### Cross-references

*ProdStageStopRun()*

## USTG_PortStopRun

### Synopsis

```
int USTG_PortStopRun (int port)
```

### Parameters

| | | |
|---|---|---|
| port | IN | stage port number |

### Description

This function is called from *ProdPortStopRun()* and should perform port dependent actions required when a run is stopped.

### Cross-references

*ProdPortStopRun()*

## USTG_GetVICNo

### Synopsis

```
USTG_GetVICNo (int* VicCrateNo)
```

### Parameters

| | | |
|---|---|---|
| VicCrateNo | OUT | VIC crate number |

### Description

*USTG_GetVICNo()* returns the VIC crate number if a VIC interface (VIC8251 [1]) is present in the VME crate.The use of this function allows to develop 'generic' stages i.e. stages which may run on several LSCs or event builders. The LSC may be identified at run time and LSC specific code executed accordingly.

### Cross-references

## USTG_InputPortParam

### Synopsis

```
int USTG_InputPortParam (int port, int trigdevice, int trigchannel)
```

### Parameters

| | | |
|---|---|---|
| port | IN | stage input port number |
| trigdevice | IN | trigger device:= 0 for CORBO VME trigger module |
| trigchannel | IN | trigger channel: 0 to 3 for CORBO |

### Description

*USTG_InputPortParam()* is called by *ProdPortInit()* with parameters specifying the trigger device and channel as defined in the CASCADE configuration file. It is typically used to initialise variables describing the trigger configuration.

### Cross-references

*ProdPortInit()*

## USTG_InputEnbTrig

### Synopsis

```
int USTG_InputEnbTrig (int port)
```

### Parameters

| | | |
|---|---|---|
| port | IN | stage port number |

### Description

This function enables the event triggers for the specified port.

### Cross-references

*USTG_InputDsbTrig()*

## USTG_InputDsbTrig

### Synopsis

```
int USTG_InputDsbTrig (int port)
```

### Parameters

| | | |
|---|---|---|
| port | IN | stage port number |

### Description

This function disables the event triggers for the specified port.

### Cross-references

*USTG_InputEnbTrig()*

## USTG_EvTrgAna

### Synopsis

```
int USTG_EvTrgAna (int port)
```

### Parameters

| | | |
|---|---|---|
| port | IN | stage port number |

### Description

This function is called before space for event data is reserved and should analyse the event trigger. This “analysis” is strongly application dependent. It may consist in finding out if the trigger is “valid” i.e whether event data should be transferred for example based on reading trigger patterns or subsets of the event data. The function value returned could be a trigger “descriptor” code.

### Cross-references

## USTG_MaxEventSize

### Synopsis

```
#include usin.h

int USTG_MaxEventSize (int port)
```

### Parameters

| | | |
|---|---|---|
| port | IN | stage port number |

### Description

This function returns the maximum expected event size. This number is used to reserve space for the event data. In some cases it may be a theoretical upper limit. In other case it may be the actual event size as obtained for example by reading word count registers in the read-out system.

### Cross-references

## USTG_InputEvent

### Synopsis

```
int USTG_InputEvent (   int port, int *dataAddress, int maxSize,
                        int *type)
```

### Parameters

| | | |
|---|---|---|
| port | IN | stage port number |
| dataAddress | IN | pointer to the CASCADE event data buffer |
| maxSize | IN | maximum event size as obtained by calling *USTG_MaxEventSize()* |
| type | OUT | event type |

### Description

This function performs the actual reading of the event data into the buffer defined by dataAddress, at most maxSize bytes. It returns the event type.

### Cross-references

## USTG_FZuserVec

### Synopsis

```
int USTG_FZuserVec(  int *userVecAddr, int *hdAddress,
                     int *dataAddress, int *trAddress,
                     int *userVecLen)
```

### Parameters

| | | |
|---|---|---|
| userVecAddr | IN | pointer to the user vector part of the ZEBRA FZ header |
| hdAddress | IN | pointer to the event header |
| dataAddress | IN | pointer to the event data |
| trAddress | IN | pointer to the event trailer |
| userVecLen | OUT | user vector length |

### Description

This function must fill the user vector part of the ZEBRA FZ header of the event. The addresses of the event header, event data and event trailer are provided so that the event characteristics can be extracted and, if necessary, plugged into the user vector.

### Cross-references

# 8 Event building

## 8.1 Introduction

Event building is generally understood as the operation which consists in merging (sub)events originating from several (sub)detectors but corresponding to the same physics event. This simple concept has been somewhat extended before being implemented in the stage so that CASCADE can be used for a wider spectrum of applications. Facilities have also been implemented so that the event integrity can be checked and so that the building operation might be aborted after a time-out signal has been received. Event building in CASCADE has the following main features:

- The event type “components” necessary to build an event of that type may include dataless “condition” components such as the occurrence of control signals, timeouts, etc.
- One event type component may itself be an event type, built from its own other components, thus providing the possibility of having one or several hierarchies of event types (e.g. “bursts” made of “events” themselves made of “subevents”).
- Although building of “super” events, such as complete bursts, may be necessary before deciding on (and possibly marking in the event header) the validity of the individual events, one may or may not wish to deliver entire bursts to monitoring programs and/or to downstream stages.
- It may be desirable to reformat the subevents header and trailer before merging the subevents into the resulting “built” event.
- The event building operation may not be the same for (or may not even apply to) all the event types seen by the stage.

It is possible to force stages to perform a building operation on the (sub)events collected from the input ports. This operation is done automatically by the stage on the basis of:

1. application-specific construction requirements which have to be set via appropriate parameters in the corresponding stage entry of the configuration file (a complete description of these parameters can be found in Chapter 13 of this document)
2. a number of application-dependent event building functions which have to be written and linked with the stage modules at system generation (these functions are automatically called by the stage to get information on the subevents and, if necessary, reformat them before performing the actual building operation)

Examples of configuration files and event building functions are distributed with CASCADE as template files (see Chapter 13 for the configuration file templates and see below for the event building functions).

## 8.2 userevb templates

An example of such application-dependent event building routines is distributed with CASCADE in the following template file. This file is compiled in makefile USR_makefile described in Chapter 4.

- **userevb_burst.c** is presently the only example provided. The UEVB_GetInfo routines extract the number of events in the burst and the event serial number within the burst from the LSC event header and return this information to the event builder. The *UEVB_FillHdTr()* function reduces the size of the LSC event headers from 12 to 10 and adjusts the header size and event size accordingly. Finally a “standard” global event header with mostly dummy data is constructed. Please note that this example is entirely based on the event header described in Section 7.3.

## 8.3 Application-dependent event building functions

These functions are written in C and compiled in makefile, USR_makefile described in Chapter 4. The functions are described below.

## UEVB_Init

### Synopsis

```
#include evb.h

int UEVB_Init(void)
```

### Description

This function is called automatically during the initialisation phase of stages which have to perform event building (according to their entry in the configuration file). It allows the user to perform any application specific initialisation related to event building.

### Cross-references

## UEVB_GetMinInfo

### Synopsis

```
#include evb.h

 void UEVB_GetMinInfo(   BLC_EventDescr *ed, char *type, int *evsn,
                         int *nbevb)
```

### Parameters

| | | |
|---|---|---|
| ed | IN | subevent descriptor |
| type | OUT | subevent type |
| evsn | OUT | serial number of event within burst |
| nbevb | OUT | number of events in burst |

### Description

This function extracts from the header of the specified (sub)event, the minimum information required by the event builder to decide whether this event needs an event building operation or not. This decision is based on the type of the event which is returned as a string of characters (of which only the first four characters will be used). If type is equal to a type specified in the event builder part of DAQCONF event building operation will be performed. The two last parameters are only relevant in that case.

### Comments

If the “standard” header is used, the parameters type, evsn and nbevb are found in words 6,7,8 of the header.

### Cross-references

## UEVB_GetMoreInfo

### Synopsis

```
#include evb.h

 int UEVB_GetMoreInfo(   BLC_EventDescr *ed, int RdScalerFlag,
                         int *evsn, *nbevb)
```

### Parameters

| | | |
|---|---|---|
| ed | IN | subevent descriptor |
| RdScalerFlag | IN | read scaler flag |
| evsn | OUT | serial number of event within burst (dummy) |
| nbevb | OUT | number of events in burst |

### Description

This function is called by the event builder once per burst, for the first subevent of the first event (but after *UEVB_GetMinInfo*). If RdScalerFlag is TRUE, the function read the scaler associated with this type of burst and compares it with the total number of events found in the (sub)event header. If the numbers are equal, the function returns a value of 0, else a value of -1. The parameter nbevb is the total number of events; in case that the comparison mentioned before fails, the largest of the two numbers should be returned.

### Returns

| | |
|---|---|
| dummy | if RdScalerFlag is FALSE |
| 0 | if RdScalerFlag is TRUE and the scaler value associated with this type of burst is equal to the total number of events found in the (sub)event header |
| -1 | if RdScalerFlag is TRUE and the scaler value associated with this type of burst is NOT equal to the total number of events found in the (sub)event header |

### Comments

This function is somewhat specific to NOMAD.

### Cross-references

## UEVB_FillHdTr

### Synopsis

```
#include evb.h

 void UEVB_FillHdTr(     BLC_EventDescr *ed, int burst_status)
```

### Parameters

| | | |
|---|---|---|
| ed | IN | event descriptor |
| burst_status | IN | status of burst |

### Description

This function is called at the end of a burst for EACH event in the burst at a point in the event building phase where event descriptors are created but the final events NOT YET assembled. It allows to adjust the header and trailer of the subevents, to build the (global) event headers and to mark events with a status (GOOD/BAD_SPILL...). The parameter ed is an event descriptor which contains a pointer to a linked list of subevents thus allowing the routine to access the headers and trailers of the subevents. The parameter burst_status is the global status of the burst as defined by the event builder based on certain consistency criteria. It is the user's responsibility to insert this status at the appropriate place in the event header.

### Cross-references

# 9 Event Monitoring

## 9.1 Introduction

In CASCADE two schemes for event monitoring are provided, **local** and **remote**. A **local** monitoring program runs in the same processor as the stage and retrieves events via shared memory. A **remote** monitoring program runs on another processor and retrieves events via a network connection from a ‘front- end’ processor where a stage is executing. Local monitoring programs are running under OS-9 or LYNXOS on VME processors while remote monitoring programs normally will run on UNIX workstations. In this chapter we describe local monitoring, however, since remote monitoring is built on top of local monitoring, this chapter is also relevant to remote monitoring.

With local monitoring, the synchronisation and message exchanges between the monitoring programs and the stage are performed by means of signals and named pipes (see below) while access to event data is via shared memory. An event is made available to the monitoring program by means of its header, data and trailer addresses. The sampling mechanism does not automatically copy the event into a user buffer. Each event delivered to an monitoring program is held in the stage data buffer until it is explicitly released. This data buffer area is locked as long as the monitoring program holds the event. As a consequence, when the processing of an event is relatively long (e.g. in the case of an event display program), it may be better to copy the event in a local array of the monitoring program and to release it immediately.

There are two modes of sampling: “on request” and “fixed” sampling. When using “on request” sampling, the stage sends to the monitoring program as many events as possible, without blocking the data taking for lack of space in the data buffer. In “fixed” sampling, the monitoring program specifies a percentage of events that it wants to receive and it is then guaranteed to be given at least the specified amount of events. The monitoring program will receive more events if possible (i.e. provided this does not block the data taking by filling up the data buffer). For both modes of sampling, the monitoring program may specify one or more event type(s) that it wants to receive. The sampling mechanism will discard events whose event types don’t match the one(s) requested. All the events with event type greater than `MAX_NORMAL_EVENT_TYPE`[1] are always reserved and unconditionally given to the monitoring programs.

NB

The “fixed” sampling mode is not fully debugged (and will never be) and not reliable and, therefore, should not be used.

## 9.2 Monitoring program templates

An example of a monitoring program and the corresponding makefile is distributed with CASCADE and can be found in the online/templates directory with the names `mp_valid`.c and `SH_makefile_demo` respectively. They are automatically copied

1. This symbol is defined in the BLC package and set to 1000 in the current version.

(be careful since many other files are copied at the same time) into the current directory (as **`mp_demo.c`** and **`SH_makefile_demo`**) by invoking the script **copy_templates_demo**.

The mp_demo.c program is the MP program which is used to "validate" CASCADE. It is distributed with the intention of serving as a starting point and example for the development of user application MP programs. The program can run in both local and remote mode (see section on remote monitoring) and is very useful as a general purpose MP for printing events flowing through the stage. The local version is built as follows :

Under OS-9: **make -b -f=SH_makefile_demo mp_demo**

Under LYNXOS: **gmake -f SH_makefile_demo mp_demo**

The program may run in interactive or automatic mode. The interactive mode allows to analyse (print) the events one after the other as decided by the user while the automatic mode is intended for continuous analysis at the highest possible speed. The program has the following parameters :

mp_demo <stage name> [dummy percent no_events no_words_dmp print_rate event_type1 event_type2 ..... event_typeN]

where

| | |
|---|---|
| stage name | name of the stage |
| dummy | this parameter is dummy in **local** mode |
| percent | percentage of events which should be seen by the MP - **must be 0 !** |
| no_events | number of events to be analysed by the MP |
| no_words_dmp | number of event words to be dumped in case of error |
| print_rate | every <print_rate> events a line is printed on the terminal |
| event_type1 .... event_typeN | defines one or more event types to be seen by the MP (0 means all) |

If all the parameters are given on the command line the program works in automatic mode. If the program is started with only the first parameters :

mp_demo <stage name> in local mode

then the parameters are defined interactively :

**`Interactive mode`** : a positive answer makes the program asking for next event after each event processed, while a negative answer makes the program continuously sampling until the number of events to sample is reached.

**`Dump all events`** **:** if the answer is yes, then one has to input the number of words to print for each event

**`Next event type to sample`**: this question is repeated until a -1 is entered. A zero value means that all event types are requested. A non zero value specifies the event type requested.

**`Enter percent of events to sample`**: a zero value means that the mode of sampling is “on request”. **At present only this mode works reliably**.

**`Number of events to sample`**: specifies the number of events that the monitoring program wants to receive before exiting.

### 9.2.1 Monitoring Program Structure

The structure of the monitoring program is slightly different depending on whether it copies the data to a local user buffer or not. The distributed templates don’t copy; as a consequence they don’t explicitly release the event by calling *sh_release_event()*, since the last event processed is automatically released when requesting the next one. Their structure is thus the following (this is pseudo-code only):

```
call sh_mconnect
while (not_enough_events_seen) {
  call sh_request_event
  call sh_wait
  call sh_get_data
  process event
}
call sh_disconnect
exit
```

If the user chooses to make a copy of the event in a local buffer, the sequence changes in the following way (this is pseudo-code only):

```
call sh_mconnect
while (not_enough_events_seen) {
  call sh_request_event
  call sh_wait
  call sh_get_data
  copy the event in the local buffer
  call sh_release_event
  process the event
}
```

```
call sh_disconnect

exit
```

The monitoring program does not have to do continuous polling to discover whether there is an event available for it in the data buffer of a stage. Instead it receives the notification via an asynchronous signal, whose value is returned via the *sh_wait* routine.

## 9.3 Error Testing In A Monitoring Program

The errors returned by the sampling routines are defined in the header file `sh_errors.h`.

Their numeric value are the following:

| | |
|---|---|
| SUCCESS | 0 |
| SERVICE_NOT_FOUND | 1 |
| ALLOCATE_FAILED | 2 |
| SIGACTION_FAILED | 3 |
| BAD_ARGUMENT | 4 |
| SEND_FAILED | 5 |
| RECEIVE_FAILED | 6 |
| RUN_STOPPED | 7 |
| PATH_FAILED | 8 |
| OPEN_FAILED | 9 |
| CLOSE_FAILED | 10 |
| STAGE_ABSENT | 11 |
| PIPE_NOT_CREATED | 12 |

In the distributed templates, the routine *sh_message()* is called in case of errors from the sampling routines and the string describing the error returned by this routine is then printed (see below).

## 9.4 Exception Handler In The Monitoring Program

An exception handler has been implemented in such a way that the monitoring program always calls the *sh_disconnect()* routine before exiting. This is needed to make the necessary cleanup inside the stage code.

The trapped errors in the OS9 systems are defined in the file `trapdefs.d`:

| | | |
|---|---|---|
| BUSERR | 102 | bus error |
| ADRERR | 103 | address error |
| ILLINS | 104 | illegal instruction |
| ZERERR | 105 | zero divide |

| | | |
|---|---|---|
| CHKERR | 106 | CHK instruction |
| TRAPV | 107 | TRAPV instruction |
| PRIVERR | 108 | privilege violation |

The trapped errors in the UNIX systems are:

| | |
|---|---|
| SIGFPE | Arithmetic exception |
| SIGILL | Illegal instruction |
| SIGSEGV | Invalid memory reference |
| SIGTRAP | trace trap |

The value of the trapped error is printed before exiting the monitoring program. The two special signals SIGINT (Control-c) and SIGQUIT (Quit special character typed) are trapped on both systems.

If the stage crashes, the signal SIGPIPE (broken pipe) is sent from the stage (really from the exit handler of the phase dealing with sampling) to all the monitoring programs connected to that stage. In this case the user routine *ush_trap()* is called within each monitoring program.

## 9.5 Debugging Tools

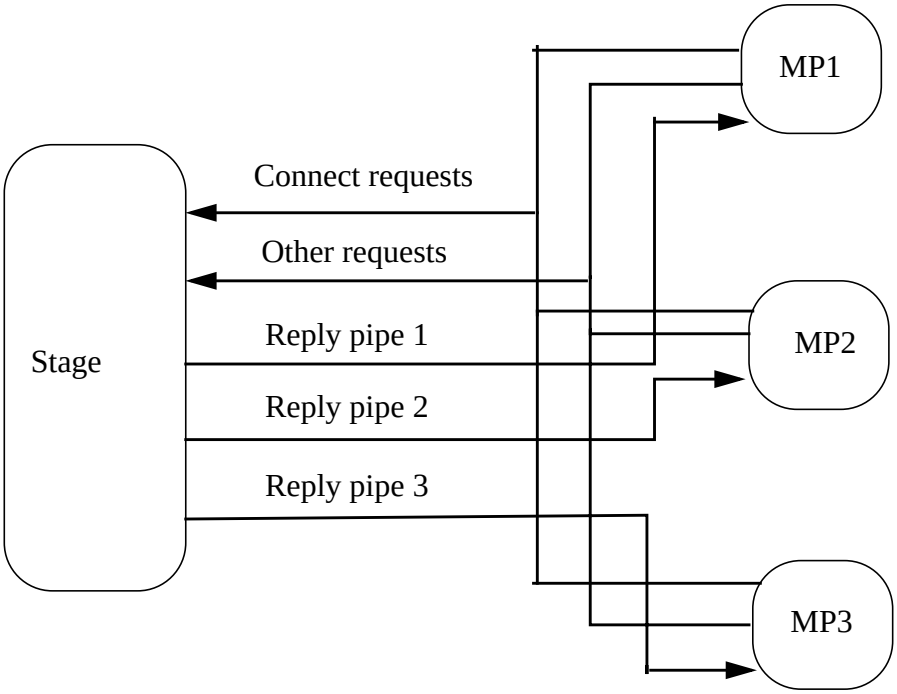
A debugging program `st_dump` is available as a separate process to display on the screen the contents of each sampling table entry in the sampling list. This gives the status of all active monitoring programs at the moment at which the program is run. This program is for specialists and a knowledge of the internals of the MP package is required to understand the output of this program.

To run it, type:

***st_dump <stage_name>***

Make sure the display auto-wrap option of the window in which the process is run is ON if you don’t want to have the lines of the dump truncated.

## 9.6 Pipes




The MPs and the stage exchange messages via named pipes. Requests from the MPs are divided into two groups, **connect** requests and **other** requests.

There is one common 'well-known' connect pipe with name of the type :

Under OS-9: **fifo$CSH$<stage_name>$0**

Under LYNXOS: **/tmp/fifo$CSH$<stage_name>$0**

When the sh_mconnect function is called a connect message is sent via this pipe.

Similarly, there is one common 'well-known' pipe with a name of the type :

Under OS-9: **fifo$OSH$<stage_name>$0**

Under LYNXOS: **/tmp/fifo$OSH$<stage_name>$0**

All other requests from the monitoring programs are sent via this pipe

The stage replies to an MP by writing to one of the **reply** pipes. There is one reply pipe per MP with a conventional name of the type

Under OS-9: **fifo$OSH$<stage_name>$<pid>**

Under LYNXOS: **/tmp/fifo$OSH$<stage_name>$<pid>**

where <pid> is the pid of the MP. The reply pipe is created by the MP at connect time and opened (only) in the connect code of the sh_request thread.

The status of these pipes can be checked with the following command :

Under OS-9: **dir -e /pipe**

Under LYNXOS: **ls -l /tmp/fifo***

## 9.7 Sampling Routines

From version 3_00 the local sampling routines, described below, are available for both OS9 and LYNXOS. To run monitoring programs under UNIX on the back-end workstations, see the section on remote monitoring.

## sh_mconnect

### Synopsis

```
#include “mp.h”

int sh_mconnect(  char* stage_name,
                  int no_event_types,
                  int* m_event_types,
                  int percent,
                  int* event_signal)
```

### Parameters

| | | |
|---|---|---|
| char* | stage_name | name of the stage |
| int | no_event_types | number of entries in array pointed to by *m_event_types* |
| int* | m_event_types | events types |
| int | percent | percentage of events to be seen |
| int* | event_signal | signal to be raised when an event is ready |

### Description

The monitoring program makes itself known to the stage and declares its sampling criteria.

The *stage_name* argument is the command line argument specifying the stage name passed to the monitoring program when it is invoked.

The *no_event_type*s argument specifies the number of event types requested.

The *m_event_types* is a pointer to an array of no_event_types integers: each element of the array specifies a requested event type. If the first element of the array is zero, any type of event will be given to the monitoring program.

The *percent* argument specifies that the monitoring program wants to receive a percentage of events of the specified event type(s). If non zero, the sampling will be “fixed” and the monitoring program will receive the requested amount of events. If percent is zero, the sampling will be “on request”.

NB. In version 2.06 this parameter must be equal to zero.

The *event_signal* argument returns the signal used by the stage to notify the monitoring program that an event is available for it, so that the monitoring program can test after the call to *sh_wait()* if it is waiting for more than one signal.

## Returns

Upon successful completion, the function shall return a value of zero. Otherwise, one of the following values will be returned:

| | |
|---|---|
| [ALLOCATE_FAILED] | Dynamic allocation of the *event_signal* to be used afterwards by the stage to notify the monitoring program that an event is available for it failed. |
| [SERVICE_NOT_FOUND] | The stage name given as parameter to the monitoring program is not found in the CASCADE dictionary created by the stage on the machine where the monitoring program runs. Most probably, a wrong stage name has been typed when the monitoring program was invoked. |
| [SIGACTION_FAILED] | The call to block the occurrence of the signal number indicated by *event_signal* failed. |
| [SEND_FAILED] | Sending the connect message to the stage through the request pipe failed. |
| [RECEIVE_FAILED] | Receiving the acknowledge to the connect message from the stage through the reply pipe failed. |
| [BAD_ARGUMENT] | The *m_event_types* argument is bad: a zero event type (which means all event types) has been specified among other non zero event types or an event type larger than MAX_NORMAL_EVENT_TYPE has been requested. |
| [OPEN_FAILED] | Opening one of the request pipes failed. The corresponding IPC package error is printed. |
| [PATH_FAILED] | Building the name of one of the request pipes failed. The corresponding IPC package error is printed. |

## Related Functions

See *sh_disconnect()*.

## sh_declare_signal

### Synopsis

```
#include “mp.h”

 int sh_declare_signal (   int user_signal)
```

### Description

The monitoring program declares signals (other than the event signal returned by *sh_mconnect()*) for which it wants to wait.The routine may be called many times (one for each signal that the user wants to declare).

The *user_signal* argument specifies the signal number that the user wants to wait for. After the call to *sh_wait()*, the monitoring program can check which signal is raised.

### Returns

Upon successful completion, the function shall return a value of zero. Otherwise, one of the following values will be returned:

| | |
|---|---|
| [BAD_ARGUMENT] | The given argument is bad: the signal has already been declared in a previous call or it is the same signal as *event_signal* returned by *sh_mconnect()*. |

## sh_request_event

### Synopsis

```
#include “mp.h”

 int sh_request_event()
```

### Description

The monitoring program makes a request for a new event and implicitly releases the last event processed, if needed (i.e. if there is one event still being processed by the monitoring program and if a call to *sh_release_event()* has not been explicitly made before).

### Returns

Upon successful completion, the function shall return a value of zero. Otherwise, the following value will be returned:

| | |
|---|---|
| [SEND_FAILED] | Sending the request message to the stage through the request pipe failed... |

## sh_wait

### Synopsis

```
#include “mp.h”

 int sh_wait(   int* signal)
```

### Description

The monitoring program waits for an asynchronous signal. The routine suspends the caller until a signal is delivered to inform the process of the occurrence of an event.

The *signal* parameter returns the signal number which woke-up the monitoring program. Two cases need to be distinguished:

- when *signal* is *event_signal* (as returned by *sh_mconnect()*) the monitoring program can find in the reply pipe the address of an event descriptor for an event of the type it requested; the monitoring program should thus call *sh_get_data()* to actually get this event;
- when *signal* is one of the user signals declared in *sh_declare_signal()* the monitoring program should act accordingly.

### Returns

Upon successful completion, the function shall return a value of zero.

## sh_get_data

### Synopsis

```
#include “mp.h”

 int sh_get_data (   int *size,
                     int *hsize,
                     int *tsize,
                     int *type,
                     int **ev,
                     int **hev,
                     int **tev)
```

### Description

The monitoring program gets an event from the event descriptor address found in the reply pipe.

The *size*, *hsize* and *tsize* arguments respectively return the event data size, the event header size and the event trailer size.

The *type* argument returns the event type.

The *ev, hev* and *tev* arguments respectively return pointers to the event data, the event header and the event trailer.

### Returns

Upon successful completion, the function shall return a value of zero. Otherwise, the following value will be returned:

| | |
|---|---|
| [RECEIVE_FAILED] | Receiving the event descriptor from the stage through the reply pipe failed. |
| [RUN_STOPPED] | Notification of run stopping/stopped received when trying to get the next event. |

## sh_release_event

### Synopsis

```
#include “mp.h”

 int sh_release_event(  )
```

### Description

The monitoring program releases the last event received (after finishing processing it or after having copied it in a local buffer). After this call, the stage can remove this event from its buffer if space is required.

### Returns

Upon successful completion, the function shall return a value of zero. Otherwise, the following value will be returned:

| | |
|---|---|
| [SEND_FAILED] | Sending the release message to the stage through the request pipe failed. |

## sh_disconnect

### Synopsis

```
#include “mp.h”

int sh_disconnect()
```

### Description

The monitoring program leaves the sampling.

The call to this routine is mandatory before exiting the monitoring program. It allows to delete the corresponding entry in the stage sampling table list, so that the stage knows that it does not any more have to reserve events for this monitoring program.

### Returns

Upon successful completion, the function shall return a value of zero. Otherwise, the following value will be returned:

| | |
|---|---|
| [SEND_FAILED] | Sending the disconnect message to the stage through the request pipe failed. |
| [CLOSE_FAILED] | Closing one of the request pipes failed. The corresponding IPC package error is printed. |
| [PATH_FAILED] | Building the name of one of the request pipes failed. The corresponding IPC package error is printed. |

## sh_message

### Synopsis

```
#include “mp.h”

 int sh_message (  int status, char *error_message)
```

### Description

It is recommended to test the status returned from all the sampling routines. In case of failure (status different from zero), one can call this routine to get a meaningful text error message from the returned integer status.

It is up to the monitoring program to decide what to do with the message (simply print it or call Emu or insert it in a log file, etc).

The *status* parameter specifies the status returned by one of the sampling routines.

The *error_message* string returns the text message corresponding to given status.

### Returns

Upon successful completion, the function shall return a value of zero.

The message “Unknown error” is returned if a status value is given that does not correspond to any known error.

### 9.7.1 FORTRAN Interface

As explained in the introduction, with version 2.06 of CASCADE local monitoring is only available under OS-9. As a consequence, a FORTRAN interface is not available. For remote monitoring, a set of FORTRAN interface functions have been implemented as explained in the section on remote monitoring.

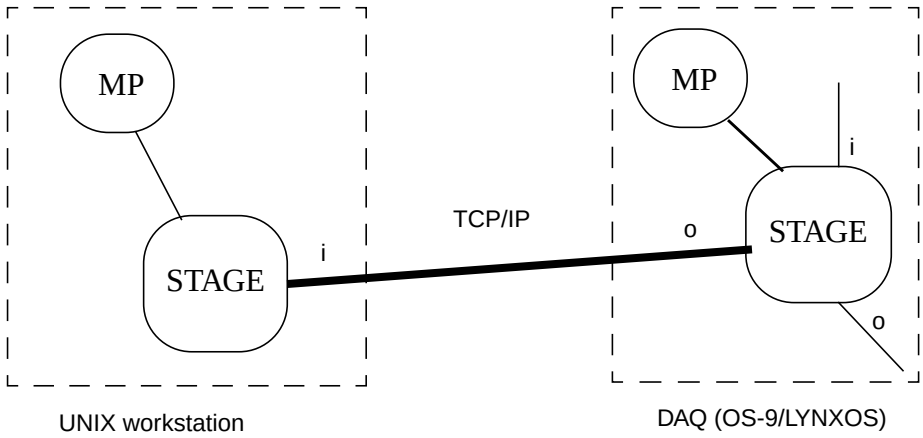
# 10 Remote Monitoring Facility

## 10.1 Introduction

Typical CASCADE data-acquisition configurations consist of multi-crate VME systems running OS-9 or LYNXOS linked to a number of UNIX workstations. An important task of the UNIX workstations is to monitor and analyse the data collected by the VME systems. A basic requirement is, therefore, that the VME systems (LSCs and EVBs) should be able to provide a high rate of events to the workstations.

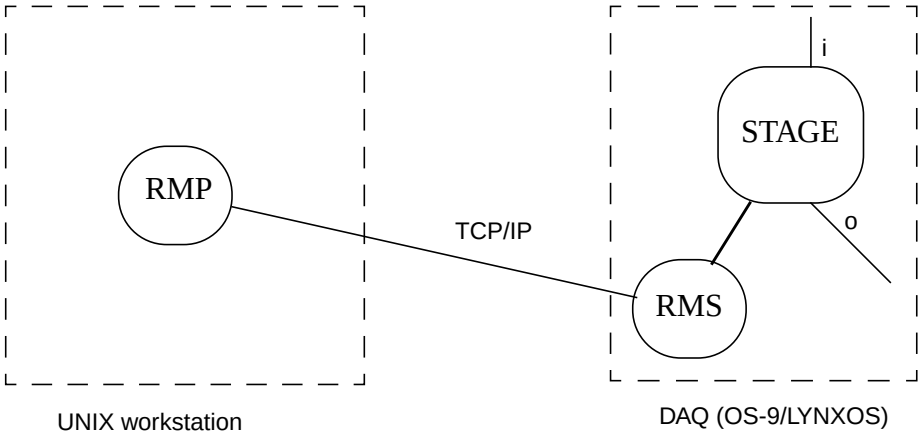
In this chapter we describe a scheme called **remote** monitoring which addresses this problem. We use the term **remote** to distinguish the scheme from the 'usual' method of **local** monitoring in CASCADE. By **local** we understand that the monitoring programs are running on the same processor as a CASCADE stage and extract events from the stage via a protocol based on pipes and shared memory, as seen in figure 4 and described in Chapter 9.

**Figure 4** Monitoring in local mode




The remote monitoring is based on a client/server approach as illustrated in figure 5. A monitoring client program (RMP) on a UNIX workstation connects to a monitoring program server (RM server) on a processor where a stage is running, usually an OS-9 or LYNXOS system. The RMP requests events from the server which - using the 'local' monitoring functions - extracts events from the stage and returns event data and status to the client where the event is then available for analysis. It is seen that in this scheme the RMP connects to a remote RM server instead of a local stage to acquire events.

**Figure 5** Monitoring in remote mode




In figure 6 is shown a generalised version of figure 5 where a number of monitoring programs on UNIX workstations get events from remote OS-9/LYNXOS systems. The RM servers are created dynamically when a client opens a link and will disappear when the client disconnects (or crashes). **Monitoring programs may be started on any workstation with an IP connection to the target system.**

The scheme proposed here is very similar to the one already used for remote PAW under OS-9. The basic difference is that remote monitoring transports **events** to the remote workstation instead of **histograms**.

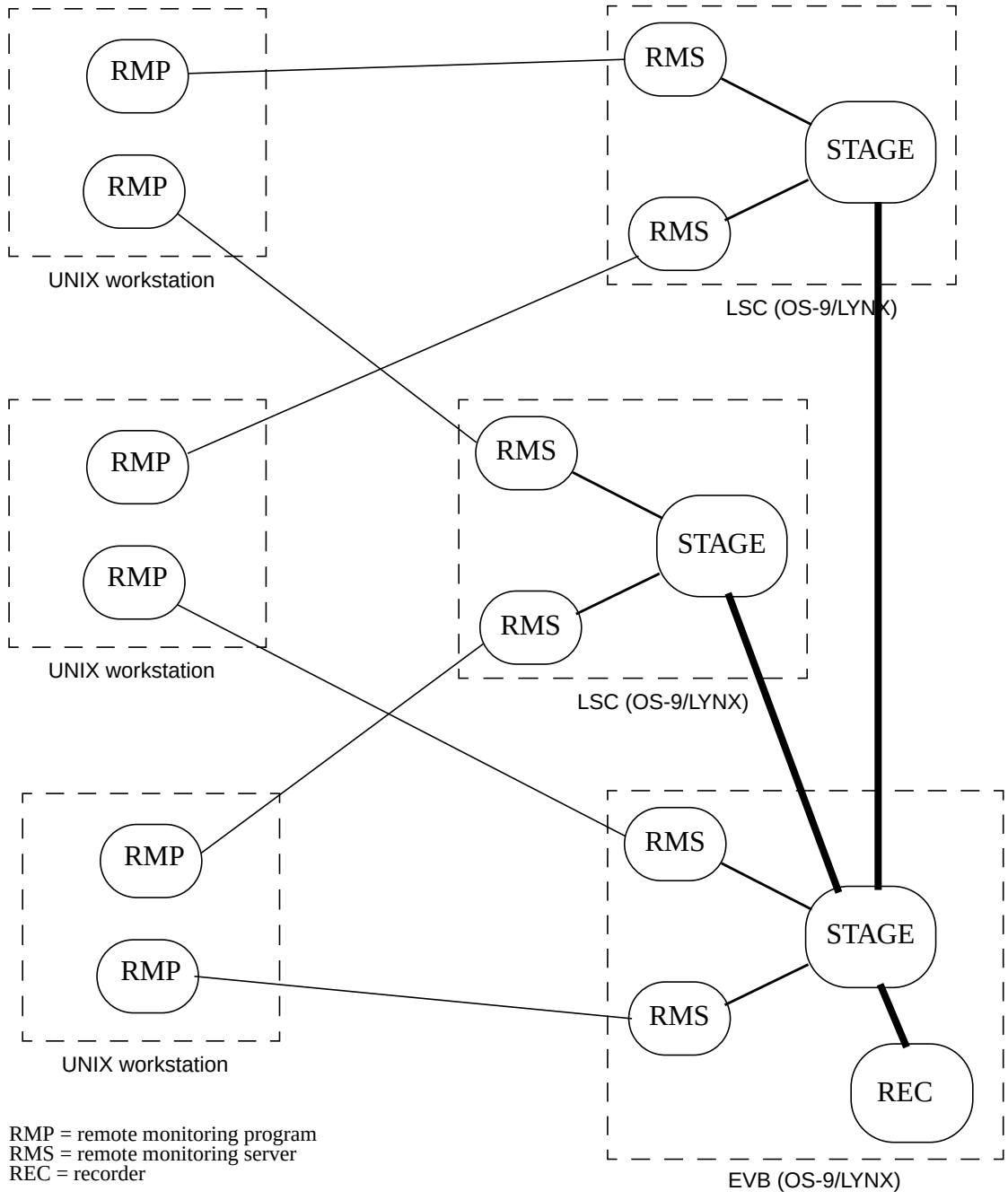
Remote monitoring is essentially a system which maps client monitoring onto equivalent server functions across the network via TCP/IP. It is based on existing software elements: the local monitoring functions (Chapter 9), the NIC library [19] and the TCP daemon mechanism under OS-9 [16] and LYNXOS. The qualitatively new element is the mapping of the functions across the network.

The remote monitoring scheme has the following features:

- monitoring programs are loosely coupled to CASCADE:
  1. the RMPs may run on any UNIX station independently of whether a stage is present
  2. monitoring does not interfere -via a local stage- with the data flow. Monitoring programs can misbehave (crash, exit) without affecting the stages under OS-9 or LYNXOS.
  3. the remote monitoring software package is independent of the CASCADE “internal” structures. It is written in C and is available independently of CASCADE releases.
- monitoring programs can run in remote and local mode almost without changes. By loading different libraries the user switches between the two modes. However, certain functions have different meanings in the two modes.
- the total number of stages in a remote monitoring configuration is smaller than in the equivalent local monitoring configuration (in figure 6 three stages are replaced by six RM servers). Run control is therefore simplified.

- A correct ending of the monitoring connection to the stage is guaranteed even if the RMP crashes or exits without disconnecting previously. The RM server will take care of closing the connection properly.

**Figure 6** Example of remote monitoring configuration




## 10.2 Remote monitoring functions

The remote monitoring library (libRM.a) available to the implementer of remote monitoring programs consists of a set of functions with the same names and syntax as those described in Chapter 9, providing a high degree of portability between the two modes. These functions typically transmit the parameters to the RM server, request

some action and retrieve data and status via the network. Some functions need some reinterpretation; *sh_wait()*, for example, does not make sense remotely but is included in the library to provide compatibility.

Below is given a short description of the remote monitoring function. The synopsis and parameters of the *sh* routines are the same as those described in Chapter 9 to which we refer for details.

## RM_init

### Synopsis

```
#include rmp.h

int RM_init (char* hostname)
```

### Parameters

| | | |
|---|---|---|
| hostname | IN | IP name of the remote host where a stage is running |

### Description

A link request is sent to the TCP daemon (inetd) on the system identified by **hostname**. The daemon forks an RM server which will then listen for requests on the socket inherited from the daemon

### Returns

| | |
|---|---|
| [RM_OK] | The TCP connection has been established |

### Comment

This function is specific to the remote monitoring and it should be called before any other in order to establish the TCP connection between the RMP and the RM server. In case of error this function prints an error message and exits.

## sh_wait_timeout

### Synopsis

```
#include rmp.h

int sh_wait_timeout (int timeout)
```

### Parameters

| | | |
|---|---|---|
| timeout | IN | timeout in milliseconds |

### Description

This function is equivalent to *sh_wait()* but including a timeout mechanism that allows the RMP to wait for a stage reply for a specified timeout instead of forever. A timeout value less or equal than zero is interpreted as infinite timeout i.e. wait forever.

### Returns

| | |
|---|---|
| [RM_OK] | a stage reply arrived |
| [SH_TIMEOUT] | the timeout expired without reply |
| [SELECT_ERROR] | error when calling the *select* system call |

### Comment

In case of SH_TIMEOUT is returned, it's up to the RMP either to retry (with the same or a different timeout value) or to give up. This function is not implemented yet in local monitoring, *sh_wait()* is used instead.

## sh_mconnect

### Description

This function establishes a link with the remote stage. A connection request message is sent to the RM server created by a previous call to RM_init. The RM server executes the *sh_mconnect()* function with the parameters retrieved from the request message and returns the status of the operation to the RMP.

### Returns

This function returns exactly the same value as the local one but a few new error codes related to the network have been added:

| | |
|---|---|
| [RM_OK] | connection established with the remote stage |
| [CONN_CLOSED] | network connection closed by peer |
| [SEND_ERROR] | error when sending request |
| [RECEIVE_ERROR] | error when receiving the reply |
| [WRONG_REPLY] | unexpected reply (protocol failed) |

## sh_request_event

### Description

This function sends an event request to the RM server which attempts to get an event from the stage by calling *sh_request_event()*, *sh_wait()* and *sh_get_data()*. The server may block in *sh_wait()* if no events are available. The event data (header + data + trailer) is copied into a local buffer in the RM server and transmitted back to the RMP with status information.

### Returns

This function returns one of these values:

| | |
|---|---|
| [RM_OK] | request sent successfully |
| [CONN_CLOSED] | network connection closed by peer |
| [SEND_ERROR] | error when receiving the reply |

### Comment

This function is mapped onto three consecutive *sh* calls in the RM server in order to get an event from the remote stage. The result of these local calls, either an event or status is passed to the RMP when *sh_get_data()* is called.

## sh_wait

### Description

This function is dummy and provided for compatibility with local monitoring (see the local version of *sh_wait*()).

## sh_get_data

### Description

The function retrieves an event from the remote stage. The event data (header + data + trailer) is read into a local buffer in the RM library.

### Returns

This function returns either *RM_OK* or an error happened during the sequence of getting an event from the stage: three local calls (*sh_request_event()*, *sh_wait()* and *sh_get_data())* plus the event transmission:

| | |
|---|---|
| [RM_OK] | event available |
| [CONN_CLOSED] | network connection closed by peer |
| [RECEIVE_ERROR] | error when receiving the reply |
| [WRONG_REPLY] | unexpected reply (protocol failed) |
| [WRONG_SIGNAL] | unexpected signal received by the RM server |
| [WRONG_SIZE] | event header, data or trailer size <= 0 |
| [WRONG_POINTER] | event header, data or trailer pointer is NULL |
| [CALLOC_SERVER] | Not enough memory for the event in the RM server |
| [CALLOC_CLIENT] | Not enough memory for the event in the RMP |

### Comment

Both the server and the client libraries have a buffer of 10Kbytes to store an event. If the event size is greater then dynamic memory allocation is required (*calloc()* is used). This memory handling is completely transparent.

## sh_release_event

### Description

A release event request is sent to the RM server which executes a local *sh_release_event()* and returns the status of the operation.

### Returns

This function returns exactly the same value as the local one but a few new error codes related to the network have been added:

| | |
|---|---|
| [RM_OK] | event released |
| [CONN_CLOSED] | network connection closed by peer |
| [SEND_ERROR] | error when sending request |
| [RECEIVE_ERROR] | error when receiving the reply |
| [WRONG_REPLY] | unexpected reply (protocol failed) |

## sh_disconnect

### Description

A disconnect request is sent to the RM server which executes a local *sh_disconnect()* and returns the status of the operation.

### Returns

This function returns exactly the same value as the local one but a few new error codes related to the network have been added:

| | |
|---|---|
| [RM_OK] | disconnection done |
| [CONN_CLOSED] | network connection closed by peer |
| [SEND_ERROR] | error when sending request |
| [RECEIVE_ERROR] | error when receiving the reply |
| [WRONG_REPLY] | unexpected reply (protocol failed) |

## sh_message

### Description

This function sends an error request to the RM server which executes a local *sh_message()* and returns the error message to the RMP.

### Returns

This function returns exactly the same value as the local one but a few new error codes related to the network have been added:

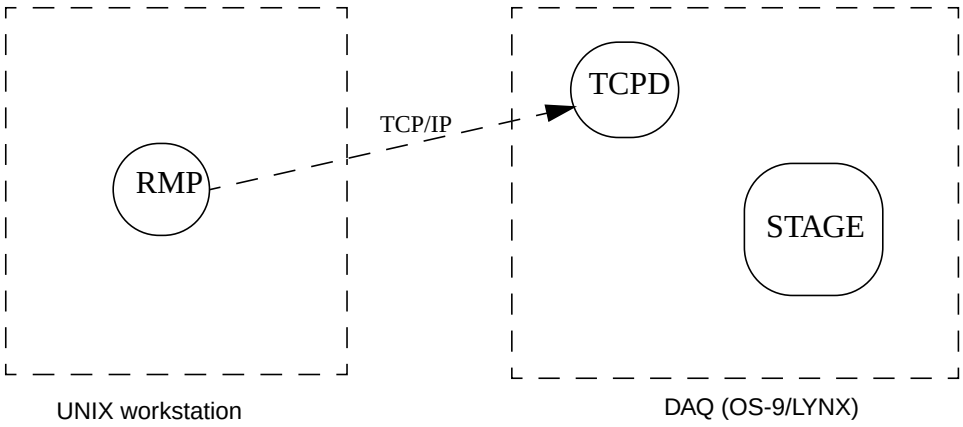
| | |
|---|---|
| [RM_OK] | error message available |
| [CONN_CLOSED] | network connection closed by peer |
| [SEND_ERROR] | error when sending request |
| [RECEIVE_ERROR] | error when receiving the reply |

## 10.3 Clients and servers

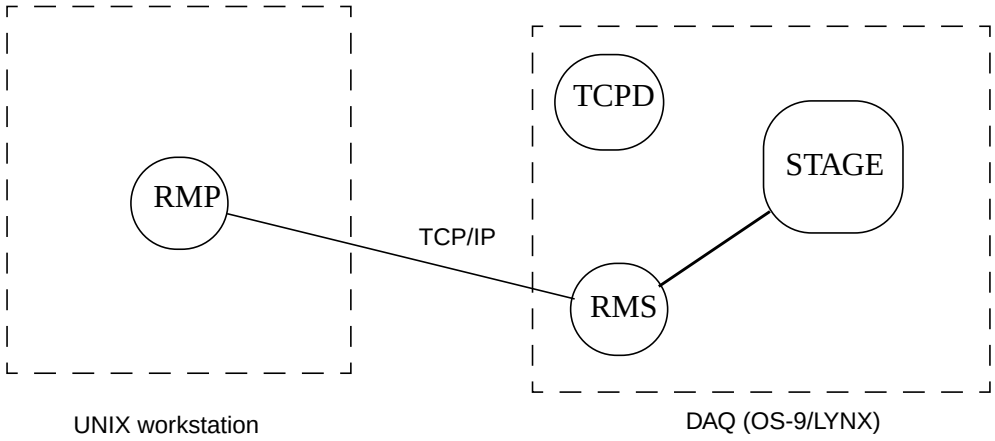
The RM servers in OS-9 and LYNXOS are created dynamically using a technique which is already employed for the PAW servers and rsh servers under OS-9. For LYNXOS, it is based on the `inetd` daemon. For OS-9 it is based on a program `tcp_daemon` [16] similar to the `inetd` daemon of UNIX. The tcp daemon is associated with an IP service called casc-rmp for remote monitoring defined in the `services` file.

The tcp daemon listens to requests on the port defined by the `casc-rmp` service. When a RMP requests a link to a RM server, the tcp daemon creates a socket defining a connection to the client and then forks a RM server which inherits the socket and therefore the connection to the RMP, see figure 7. A client/server pair is now established and the next request from the client will be directed to the forked RM server.

**Figure 7** Tcp daemon and remote monitoring servers




a) RMP sends a tcp connection request




b) TCPD forks a RMS who inherits the connection

TCPD = tcp daemon
RMP = remote monitoring program
RMS = remote monitoring server

The RMPs are very similar to normal CASCADE monitoring programs but have a distinct part to establish a connection with a remote RM server. The client checks whether a service *casc-rmp* is defined in the IP services file and retrieves the corresponding port number. If the service is not defined a 'hardwired' port number is used. A connect request is then sent to the server with the request data specifying the name of the remote stage and the sampling parameters. The server tries to connect to the stage and returns the status of the operation to the client. If there are no errors the client will now proceed with sending requests for events. The server, in turn, requests events from the stage and waits if no events are available. When this is the case the event (header + data + trailer) is copied into a local buffer and is then transmitted with the status of the operation.

## 10.4 Installation on OS-9

To set up the remote monitoring facility an IP service has to be added to the services file and a few system programs have to be installed:

- **remote monitoring IP service**

The network service file: /`os9/system/etc/services`

needs to be modified to include the following service (see Network services and daemons in Chapter 2):

```
casc-rmp7734/tcp #CASCADE Remote monitoring server
```

- **mp_server**

This program is found in `/os9/online/cmds` on the ECP/FEX OS-9 cluster and it must be conveniently loaded[1] into memory before using remote monitoring.

- **tcp_daemon**

This program is described in [16] and can be found in `/os9/online/cmds` on the ECP/FEX OS-9 cluster. It should be started as follows

```
tcp_daemon casc-rmp mp_server -f /dd/tmp/rmp.log <>>>/nil &
```

tcp_daemon uses the IP service *casc-rmp* and will fork program mp_server when a request is received. The mp_server program will write log messages in file `/dd/tmp/rmp.log_pid` where pid is the pid of mp_server. The logfile is optional. All standard and error I/O is redirected to `/nil` and tcp_daemon executes in background

To summarise, a startup file for the remote monitoring may look like:

```
load -d /os9/online/cmds/tcp_daemon

tcp_daemon casc-rmp mp_server -f /dd/tmp/rmp.log <>>>/nil &
```

## 10.5 Installation on LYNXOS

- The network service:

  ```
  casc-rmp7734/tcp #CASCADE Remote monitoring server
  ```

  needs to be declared in the /`etc/services` network service file

- The following daemon entry needs to be done in the file '/etc/inetd.conf':

  ```
  casc-rmp stream tcp nowait cascade
  /usr/local/online/bin/mp_server mp_server -f
  /tmp/rmp.log
  ```

1. This step is automatically done by the CASCADE starting script files.

**Warning**:

Since under Unix and therefore LYNXOS, a process can send signals only to processes with the same ownership and since the mp_server and the stage processes send signals to each other, it is mandatory under LYNXOS that stages have the same owner has the one declared for mp_server in the inetd.conf entry mentioned above.

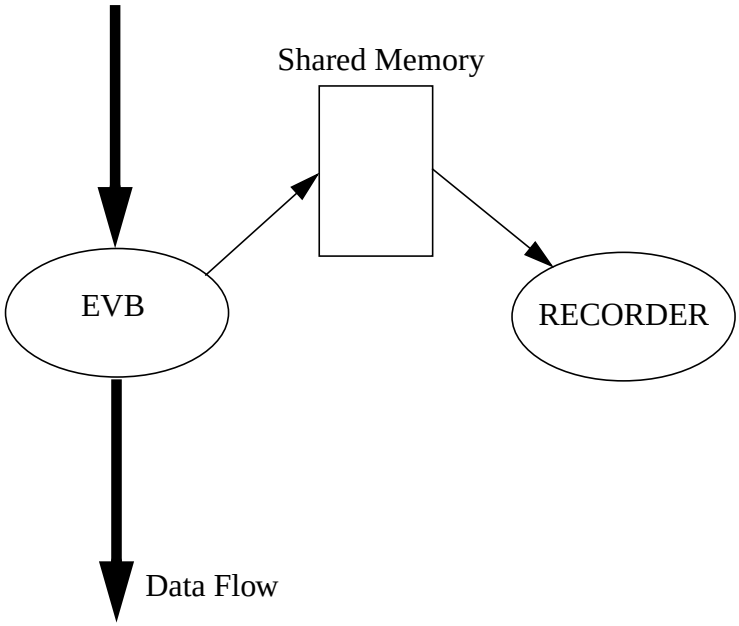
# 11 Data Recording

## 11.1 Introduction

Recording is done by a special type of stage called a recorder which has for unique task the storage of events on a physical device. Recorders must run in the same CPU as the stage which feeds them. Recorders have only one input and have no output to other stages. The interface and the protocol used to communicate with a recorder are the same as for inter-stage communications but the inter-stage link is based on shared memory.

During the run, the stage prepares events and signals the recorder every time an event is available in the shared memory. When signaled, the recorder retrieves the event, splits it into fixed blocks (records) and stores them on the medium.

**Figure 8** Data recording logical flow




At present, recording can be done on a dummy device (for test purposes), on disk files (locally or via NFS) and, under OS-9, it can be done either on the IBM3480-compatible STK4280 cartridge device[13], on the EXABYTE [14] and on the DLT, Digital Linear Tape device. Remote disk recording using a client-server approach is also available to record data on a Unix machine accessed via the network.

Three recorder processes called respectively recorder, xarecorder and dltrecorder exist. All three support the four types of recording (respectively called dummy, disk, tape recording and remote disk). An environment variable is used to define the desired recording type.the dummy, tape, local disk and remote disk recording. They only differ by the tape device each of them supports.

**recorder** must be used to record on the STK4280 cartridge device.

**xarecorder** must be used to record on the Exabyte device.

**dltrecorder** must be used to record on the DLT device.

## 11.2 Event formatting and packing

In CASCADE, event recording is always done according to the ZEBRA FZ format specification. For efficiency reasons, the formatting operation is split between the recorder and itsfeeder stage. As foreseen by the ZEBRA FZ specification, events can (but do not need to) be packed into the recording device fixed length physical records.

By default, CASCADE does not pack events, but packing can be activated by setting the environment variable **FZ_EV_PACKING** to any non-zero value **in the feeder stage environment**. The stage retrieves this variable from the environment and the recorder gets it from the feeder stage shared segment.

In the non-packing mode, every event uses the smallest integer number of physical records (at least one) which can accomodate the event including its FZ envelope. Padding words are appended to the 'enveloped' event so that the whole lot fits exactly in an integer number of physical records. In the packing mode, every event is appended to the tail of the previous one in the current physical record and no padding words are inserted. This mode is of course much less greedy in space on the recording device.

## 11.3 Starting the Recorder

The command line for the recorder(s) is

[xa/dlt]recorder <rec_name> [ <configuration_file> ]

The optional second parameter is a full path name for the configuration file. If it is not specified, the default name "./daqconf" is used.

The recorder is configurable via a number of environment variables. All these environment variables should be set via the stage booting script files (see Chapter 5) and not directly using setenv commands.

The following variables apply to the four types of recording:

- RECORDERTYPE

  The type of the recorder to be used must be set using the environment variable RECORDERTYPE with one of the following keywords:

  | | |
  |---|---|
  | dummy | for test purposes |
  | disk | for local or NFS file |
  | tape | for SUMMIT, EXABYTE or DLT |
  | network | for remote disk |

  It is NOT case sensitive. Any other string will generate a error message: “Unknown type of recorder”.

- ERRORS_TO_EMU

  Reporting of errors is done using either printf or EMH_SysMsg calls as set by the environment variable ERRORS_TO_EMU:

  0 - use printf (default)

  1 - use EMU

  NB: if EMU is chosen, be sure that emu is started (otherwise the recorder process will be stopped by a full pipe).
- MAX_BLOCKS_PER_FILE

  Maximum number of blocks (records) per file (or tape). When this number is reached either on a file or on tape, the end-of-run sequence is triggered.
- REC_DISABLE_DEBUG

  If set, this variable will prevent the recorder to write debug messages to its logfile. The default is REC_DISABLE_DEBUG=0 (ie debug messages are enabled)
- REC_STOP_ON_ERROR

  In case of write error during a run, the recorder tries to close the tape/file and goes to the error state. By default, it will send an End Of Run request to the run control, and all the system will be stopped. If REC_STOP_ON_ERROR is set to a non zero value, the recorder will remain in the NRC_ERROR state and will not stop the whole system.

There are other environment variables depending on the type of recording that has been selected.

## 11.4 The dummy recorder

The recorder can be configured as dummy for test purposes. In this case, it just prints a few lines every time an event is retreived from the shared memory (the first 40 and last 16 bytes of each record), which is convenient to check that some records are produced and that they have the expected format.

If used during a test with a real data flow, this "chattering" recorder will quickly fill a logfile and create some problems, except if you use the following :

- REC_MUTE

  If set to non-zero, the dummy recorder will not print anything in the logfile. It just consumes the events and counts them. The events are not printed nor written anywhere.

## 11.5 Tape Recorder for SUMMIT

It has not been possible to produce a unique recorder executable module handling EXABYTE, SUMMIT and DLT recording because of uncompatible SCSI libraries. Therefore there is a different executable for every device. Tape recording on SUMMIT is handled by the OS9 module **recorder**.

The tape recorder saves the records taken from the shared memory on a tape. The run control is used to set the tape parameters, eventually the label parameters and allows the user to pilot the tape (rewind, unload it). The principle is that every parameter is set before trying to start a run. The run is actually correctly started after a check of the recording parameters.

The recorder allows to write either labelled or unlabelled tapes depending on the selection done by environment variables. It also provides tape information which can be saved into a data base for tape administration.

### 11.5.1 Environment Variables for SUMMIT Tape Recording

Before running the recorder process in tape mode, the following environment variables have to be defined:

- `REC_LABEL`

  Setting this variable to non zero enables the tape labelling.

- `UNLOAD_AT_EOR`

  Used to produce tapes containing only one file. If this variable is set to non-zero, a stop run sequence (either because of end of tape, some error or user stop command) implies an automatic unload.

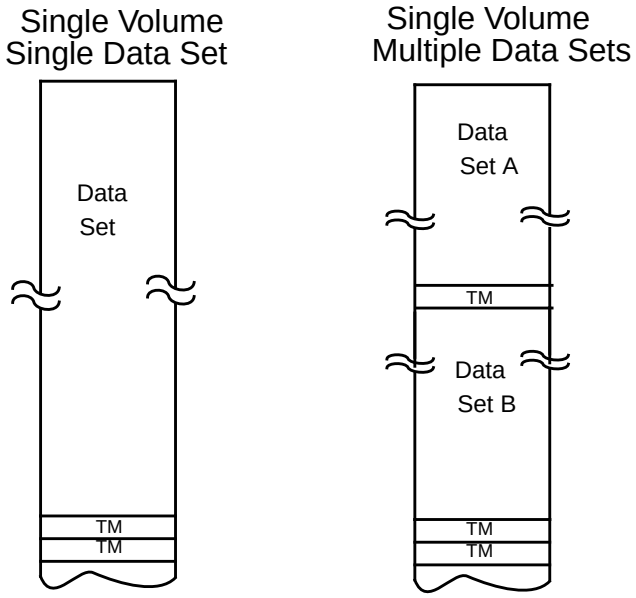
### 11.5.2 Unlabelled Tapes

If the environment variable REC_LABEL is set to zero (or unset), the recorder will produces unlabelled tapes.

**1.** Organisation for unlabelled tapes:

Only single volume are allowed (i.e. a file cannot extend on more than one tape). It is possible to produce either single data set or multiple data set tapes. As shown in Figure 9, each data set is followed by a tape mark, and the last data set is followed by two tape marks.

**Figure 9** Unlabelled tapes




**2.** Starting a run:

At start of run, the tape has to be positioned either at beginning of tape (BOT) if this is a new tape or if it has been rewound, or after two tape marks if this is not the first run to be recorded on this tape.

a. If the tape is at beginning: -BOT-
   Nothing is to be done, the tape is correctly positioned to perform recording.
b. If the tape already contains data, it is positioned after two tape marks:
   -[DATA]-TM-TM-
   The run will be recorded after the first TM, overwriting the second TM.
c. In every other case, the tape is in an abnormal position and the run can not start correctly.

**3.** Ending a run

At end of run, two filemarks are written to signify end of data (EOD)

**a.** In case of a single run, the tape will now look like this: -BOT-[DATA]-TM-TM-

**b.** In case of multiple runs, we get the following:
-BOT-[DATA1]-TM-[DATA2]-TM-....-[DATAn]-TM-TM-

### 11.5.3 Labelled tapes

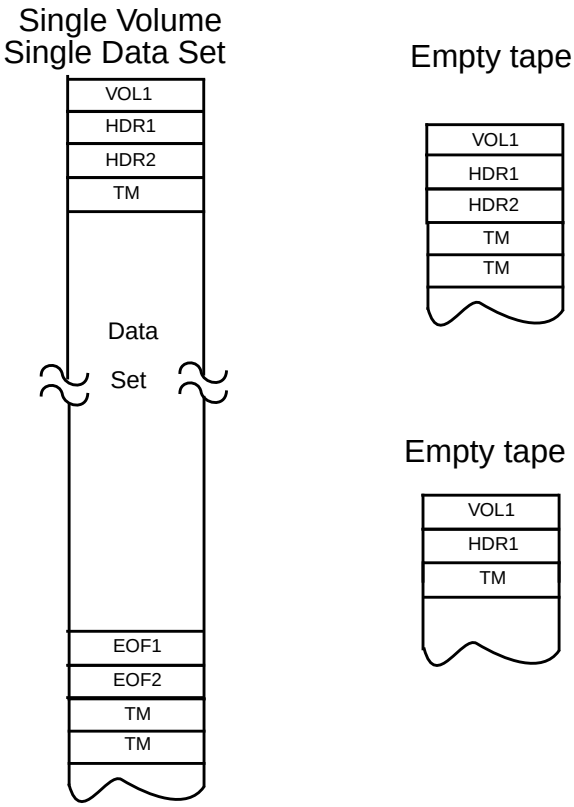
The recorder uses a CERN written portable labelling package [25] to handle tape labels. This option is activated by means of the environment variable REC_LABEL. When this variable is set to a non zero value, the recorder produces IBM standard labelled tapes.

**1.** Organisation for labelled tapes:

As for unlabelled tapes, only single volume tapes are allowed (i.e. a file cannot extend on more than one tape). Moreover, it is possible to produce only single data

sets for labelled tapes. The data set is preceded by header labels and followed by trailer labels.

**Figure 10** Labelled tapes




**2.** Starting a run

At start of run, the tape must be positioned at beginning and empty (i.e. containing only prelabels). If the volume serial number matches what has been transmitted by the run-control, the run is started, else an error is returned. The prelabels are then overwritten by real labels and the data can be written to the tape (note: compilation flags can modify this philosophy).

**3.** Ending a run

At end of run, the data set is closed and the trailers and two tape marks are written. The next operation should be unload tape, to mount the next tape. If the environment variable `REC_UNLOAD_AT_EOR` is set to a non zero value, the unload operation is automatically performed when the run stops (either by a user manual stop command, or because the tape is full or if an error occured) and the Volume Serial Number is incremented (with Old Run Control).

**4.** Labels Description

The labels are 80-character long records in EBCDIC of one of the following types:

| **Label Id** | **Label Description:** |
|---|---|
| VOL1 | Volume label |
| HDR1 and HDR2 | Data set header labels |
| EOV1 and EOV2 | Data set trailer labels (end of volume) |
| EOF1 and EOF2 | Data set trailer labels (end of data set) |

| Label Id | Label Description: |
|---|---|
| UHLn | User header labels (unlimited number permitted) |
| UTLn | User trailer labels (unlimited number permitted) |

As only single volume data set are to be handled, and no user labels are foreseen, we only need VOL1, HDR1, HDR2, EOF1, EOF2. The format of the label is:

| Field Name | Position - Length | Example |
|---|---|---|
| **VOL1:** | | |
| Label Id | 0-3 | VOL |
| Label Number | 3-1 | 1 |
| Volume Serial Number | 4-6 | TD3000 |
| Accessibility | 10-1 | <space> = not protected |
| Owner Id | 41-10 | NOMAD |
| **HDR1/EOV1/EOF1:** | | |
| Label Id | 0-3 | HDR (or EOV or EOF) |
| Label Number | 3-1 | 1 |
| File Id | 4-17 | run# or PRELABEL |
| Set Id | 21-6 | TD3000 (same as VSN) |
| File Section Number | 27-4 | 0001 (single volume) |
| File Sequence Number | 31-4 | 0001 (single data set) |
| Creation Date | 41-6 | byyddd |
| Expiration Date | 47-6 | byyddd |
| Accessibility | 53-1 | <space> = not protected |
| Block Count | 54-6 | (for the current set) |
| **HDR2/EOV2/EOF2:** | | |
| Label Id | 0-3 | HDR (or EOV or EOF) |
| Label Number | 3-1 | 2 |
| Record Format | 4-1 | U = Undefined |
| Block Length | 5-5 | 3600 |
| Record Length | 10-5 | 0 (should be 3600?) |
| Buffer Offset | 50-2 | 00 |

**5.** Detection of End Of Tape

The recorder should never reach the physical end of tape. In fact, it sends a request to stop the run as soon as MAX_BLOCKS_PER_FILE records are written (see 11.2 Configuring the recorder). This request takes some time to be seen by all the DAQ system components, and then the buffers are flushed, which means that more than MAX_BLOCKS_PER_FILE records will be actually written. There is no guaranty

that it will not reach the logical end of tape if the value of this environment variable is not set properly (if it is too high). If this happened, the recording would be stopped and the records left would be lost.

After the last records are written on tape, the data set is properly closed (trailer labels and two tape marks), the tape is automatically unloaded and the next one is mounted (the label vsn is automatically incremented by the recorder and sent to the Old Run Control, ; the New Run Control computes the run parameters itself).

## 11.6 Tape recorder for DLT

It has not been possible to produce a unique recorder executable module handling DLT, EXABYTE and SUMMIT recording because of uncompatible SCSI libraries. Therefore there is a different executable for every device. Tape recording on DLT is handled by the OS9 module **dltrecorder**.

The DLT recorder works in the same way as the SUMMIT recorder and it uses the same environment variables.

## 11.7 Tape recorder for EXABYTE

It has not been possible to produce a unique recorder executable module handling EXABYTE, SUMMIT and DLT recording because of uncompatible SCSI libraries. Therefore there is a different executable for every device. Tape recording on EXABYTE is handled by the OS9 module **xarecorder**.

The EXABYTE recorder works globally in the same way as the SUMMIT recorder except for the following :

### 11.7.1 Environment variables for EXABYTE tape recorder

The EXABYTE uses the same `REC_LABEL` and `REC_UNLOAD_AT_EOR` variables as the SUMMIT and added variables to determine the type of EXABYTE recorder and the density.

- `REC_EXA_TYPE` : 8200 or 8500 (def : 8500)

  defines the type of EXABYTE to be used.

  note : 8200 may no longer be supported.
- `REC_EXA_DENS` : 8200 or 8500 (def : 8500)

  if a 8500 EXABYTE is used, it can be configured to write either in its normal density (8500) or emulating the 8200 density. This variable is irrelevant for others than 8500 EXABYTE.

### 11.7.2 Recording session

Everything works as when using a SUMMIT except for the 8200 EXABYTE. This (old) device does not provide the facility which allows to know the current position on the tape. Thus the `MAX_BLOCKS_PER_FILE` limit to a tape cannot be handled.

With a 8200 EXABYTE, a run is roughly stopped when the logical end of tape is reached.

## 11.8 Remote disk recorder

Remote recording can be handled by the process recorder or dltrecorder or xarecorder. Any of these modules can be configured as a remote disk recorder by setting RECORDERTYPE=network. It allows to write CASCADE event data to a remote disk, typically from a front-end VME system to a disk on a UNIX workstation.This functionality is also provided, in principle, by recording via NFS, but measurements have shown that this is unacceptably slow (about 10 kbytes/sec). The remote disk recording described below bypasses the NFS file protocol.

Remote disk recording is using a TCP/IP client-server approach. The client is a recorder attached to a stage and the server responds to requests from the client to write records to disk.

When a run is started - and disk recording is selected - a remote disk server process is started (forked by an inet daemon). A start-of-run message is sent to the server which opens a file in a directory to be defined directly via the human interface. A positive acknowledge is returned if no errors conditions are found.

The client then proceeds to send records to the server which writes them onto disk. At end-of-run the file is closed as well as the connection to the client and the server exits. The end-of-run condition may be triggered by a user command or by the client when a maximum number of records is reached or possibly by the server if errors occur eg. lack of disk space.Before exiting, the server starts, in background, the execution of a user script the name of which has to be declared via the human interface. This informs the user that the data file is ready for further processing.
The format for this call is :
<USER_SCRIPT_NAME> <DATA_FILE_NAME> > /tmp/eor_script.log 2>&1 &

The file are written into a directory (defined via the run control human interface) with fixed names :

RUN_(no).fz

where no is the CASCADE run number.

### 11.8.1 Environment variables for remote disk recording

The execution of the recorder in remote disk mode is controlled via a set of environment variables which are most conveniently defined in the recorder start script file. The start.recdisk_valid file in the /templates directory provides an example.

- DISK_MIN_FREE

  Minimum disk free space in Mbytes. The server checks regularly how much space is left on the disk. If there is less than the number of Mbytes specified by DSIK_MIN_FREE, the end-of-run sequence is triggered.

- `DISK_HOST`

  The IP name of the host.

  Note : The TCP port number to be used for the connection is extracted from the /etc/services file that must define the service casc-drec (ex : casc-drec 7733/tcp). The inet daemon listens on this port number for requests to fork a server.

### 11.8.2 Setting up the disk server

The disk server is distributed with CASCADE in the online/bin directory. It has to be defined in two UNIX network configuration files (as on the client side):

- '/etc/services' for example :

  ```
  casc-drec      7733/tcp
  ```

  The port number should be the same as the one defined in the recorder environment on the client side.

- '/etc/inetd.conf' for example (under SunOS) :

  ```
  casc-drec stream tcp nowait cascade /usr/local/online/bin/rec_server rec_server
  ```

  This allows multiple instances of rec_server running with the uid of user cascade and invoked with no parameters.

## 11.9 Logging of tape information

This feature makes sense for labelled tapes only. An EMU message is generated using *EMH_SysMsg()* for each Start/Stop operation. This trace message is to be routed to a logfile that will be decoded and injected into a database so that the tapes could be easily read back.

The database message is made of:

- the operation name: START or STOP
- the tape volume serial number: label_vsn
- the run number
- the volume number: (always 1 because the recorder produces only single volume tapes)
- the date and time of the operation,
- the number of records written to tape for the current data set (0 if the operation is START),
- the file nb (always 1 if single data set tapes are produced, or the sequence number of the data set on the tape)
- the status of the system (always OK!).

All these different fields are separated by the separator ^. The message begins and ends with two separators ^.

Example:

if the run #17 has been started with the tape TD3005 and stopped after recording a few records (2274) two messages will be generated:

```
^^START^label_vsn=TD3005^run_number=17^volume_number=1^date=94-03-29
11:22:52^nb_records=0^file_nb=1^status=OK^^
```

and then

```
^^STOP^label_vsn=TD3005^run_number=17^volume_number=1^date=94-03-29
11:23:17^nb_records=2274^file_nb=1^status=OK^^
```

# 12 Error and Message Handling and Reporting

## 12.1 Introduction

In the context of CASCADE, a number of facilities are available to handle messages originating from both the application specific modules and the CASCADE package itself. Whether the message's contents refer to an error, a warning or anything else does not make any difference in the message facilities provided in CASCADE. These facilities include:

- message preparation outside the application code
- message injection from OS-9, LYNXOS and UNIX systems
- selective message routing at run time
- message transport across heterogeneous operating system platforms
- support for a variety of message destination types.

These facilities have been implemented using a number of packages which are mentioned below. This chapter provides most of the information required to use the system, while making references to other packages documentation whenever necessary.

- The central unit - the error message transfer utility to be used - is EMU. EMU, written in ADA, has originally been designed for the MODEL data acquisition system running under VAX/VMS and has been in use on that platform for a few years [20]. It has since been rewritten in C in order to run on UNIX, LYNXOS and on OS-9 [21][1]. EMU allows for message preparation outside the application code and for a selective message routing at run time.
- EMN [22], the TCP/IP network connection for EMU, makes the use of EMU transparent in a distributed system. Two or more UNIX, LYNXOS or OS-9 systems running EMU can be connected via EMN client and server processes.
- In order to allow CASCADE system messages as well as CASCADE user messages to be handled by EMU and EMN without interference some guidelines have been specified. Additional functions have been provided to facilitate the correct implementation of these guidelines.
- The ED package provides facilities to display EMU error messages in Motif windows or to print them on the standard output device.
- The various steps on setting up the EMU, EMN and ED packages are listed.

## 12.2 General Flow of Cascade Error and Message Handling

The phases which appear in the error and message handling in CASCADE are shown in Figure 11 and Figure 12. They are:

**a.** An error may be detected in a CASCADE process, originating in one of the CASCADE packages, in a CASCADE independent library (i.e. HW, system) or in

1. The Posix version, written in C, is named EMUX in the referenced documentation

the user code. This error may be identified by a number or by a text string. An error message follows from here on the same path as any information message to be sent.

**b.** The function *EMH_UsrMsg()* transforms the text and the parameters into an EMU message, adds CASCADE specific information and calls the necessary EMU injection routines.

**c.** The message enters the EMU system. The message text is, if available there, retrieved from the EMU decoder message file. In a distributed configuration it is routed according to the local and remote routing setup for EMU and the EMU network configuration EMN.

**d.** The EMU message is retrieved by one or more final destination(s), for example a log file or a user written process. EmuDisplay [23], a program using Motif windows, is available to assist in grouping and displaying messages on the screen.

**Figure 11** Example of CASCADE Error and Message Handling

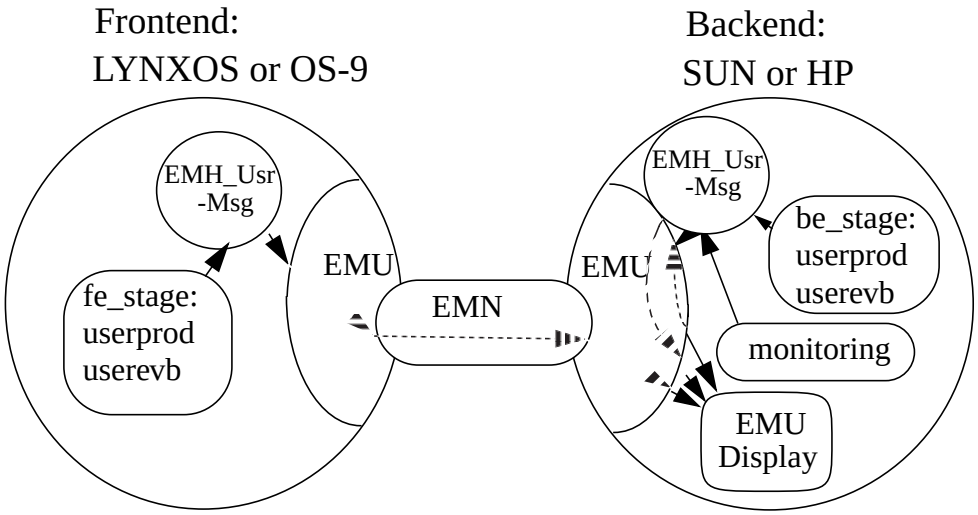
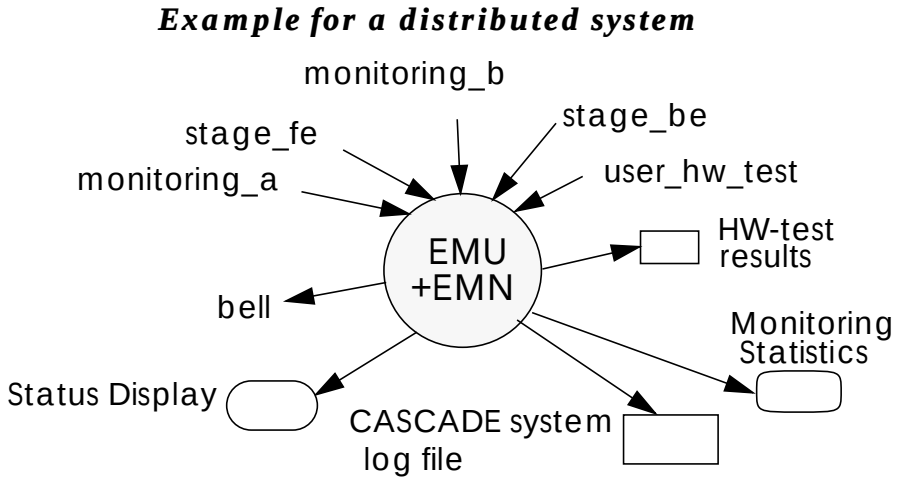



**Figure 12** Error and Message reporting in CASCADE - an overview



## 12.3 Error Message Utility EMU in CASCADE

### 12.3.1 Overview

**EMU [ps]** is a general purpose error messaging utility which runs on all parts of the CASCADE data acquisition system. EMU can be conveniently separated into two main parts: the EMU kernel for local message decoding and routing and the networking layer EMN for transport of EMU messages between machines.

### 12.3.2 The EMU kernel

This consists of an EMU router process (emurout) and EMU decoder process (emudeco) and a library of user-callable EMU initialisation and message injection functions. User programs always inject messages into the emu2deco FIFO on the local machine via the library functions. emudeco reads messages from the emu2deco FIFO, decodes them according to a set of rules contained in a map file (deco.map) and injects the decoded messages into the emu2rout FIFO. emurout reads decoded messages from the emu2rout pipe and distributes them to a set of destinations according to a set of rules contained in a map file (rout.map). Destinations may be other FIFOs, logfiles, or consoles.

### 12.3.3 The EMN network layer

The EMN networking scheme used in CASCADE uses a client/server approach. The network setup for each EMN client system is described in the file emunet.info. A client process emunet_clnt connects to the emunet_serv service on a remote machine. It sends all messages which it receives in a private FIFO - declared in the local rout.map file - via a TCP/IP socket to the remote service which transfers them to the remote EMU system. In this way a subset of all messages seen by the local emurout can be sent to the remote system.

## 12.4 Message streams and Severity in CASCADE error message handling

### 12.4.1 Message Streams

In the EMU system used in the context of CASCADE we distinguish between

- CASCADE system messages sent by CASCADE system implementers,
- CASCADE user messages sent by user code integrated into CASCADE (i.e. userprod, userevb, monitoring) and
- CASCADE independent messages

This separation provides a first division into message streams. To allow for the corresponding definition for the routing of Messages in the EMU system setup the following **EMU properties** must be used.

- CASCADE system messages: **CASCADE_SYS**

- CASCADE user messages: **CASCADE_USR**
- CASCADE independent messages: use any property name but **NOT** CASCADE*, in particular **NOT** CASCADE_SYS and not CASCADE_USR

Users of the EMU system not connected in any way to CASCADE form the third EMU client group within the CASCADE EMU system. He/she must not use an identifier starting with 'CASCADE'.

The user has the option to use messages having been specified by him/her in the EMU decoder message file or to send their EMU message and its associated property directly to EMU without specifying them in the EMU decoder message file. During the development phase it may be practical to use the method of direct EMU message injection, whereby the entire text of the message is passed via the injection routines to EMU. If the messages are defined in the decoder message file an 'EMU system manager' must coordinate the setup and update of this file. For a library package of more general use it is recommended to define the EMU messages in the decoder file under the package identifier. Those messages have to be defined only once to the message decoder file but can be used by a number of programs. The package identifier has to be added to the call when injecting a message.

### 12.4.2 Severity

PROPERTIES are used primarily for the routing or filtering of EMU messages but also for message identification by the receiver. This can be considered as another level of message stream separation. A set of pre-defined properties has been setup primarily to be used for CASCADE system and CASCADE user messages. One of the following PROPERTIES shall be associated to a message:

**FATAL - ERROR - WARNING - INFO - SUCCESS**

Additional properties may be added to the EMU message.

The severity of an error or a message may depend on the context of its occurrence. Therefore the severity is added at runtime to the message.

Note: The implementation of EMU for UNIX, LYNXOS and OS-9 is case sensitive.

### 12.4.3 Reserved names for CASCADE

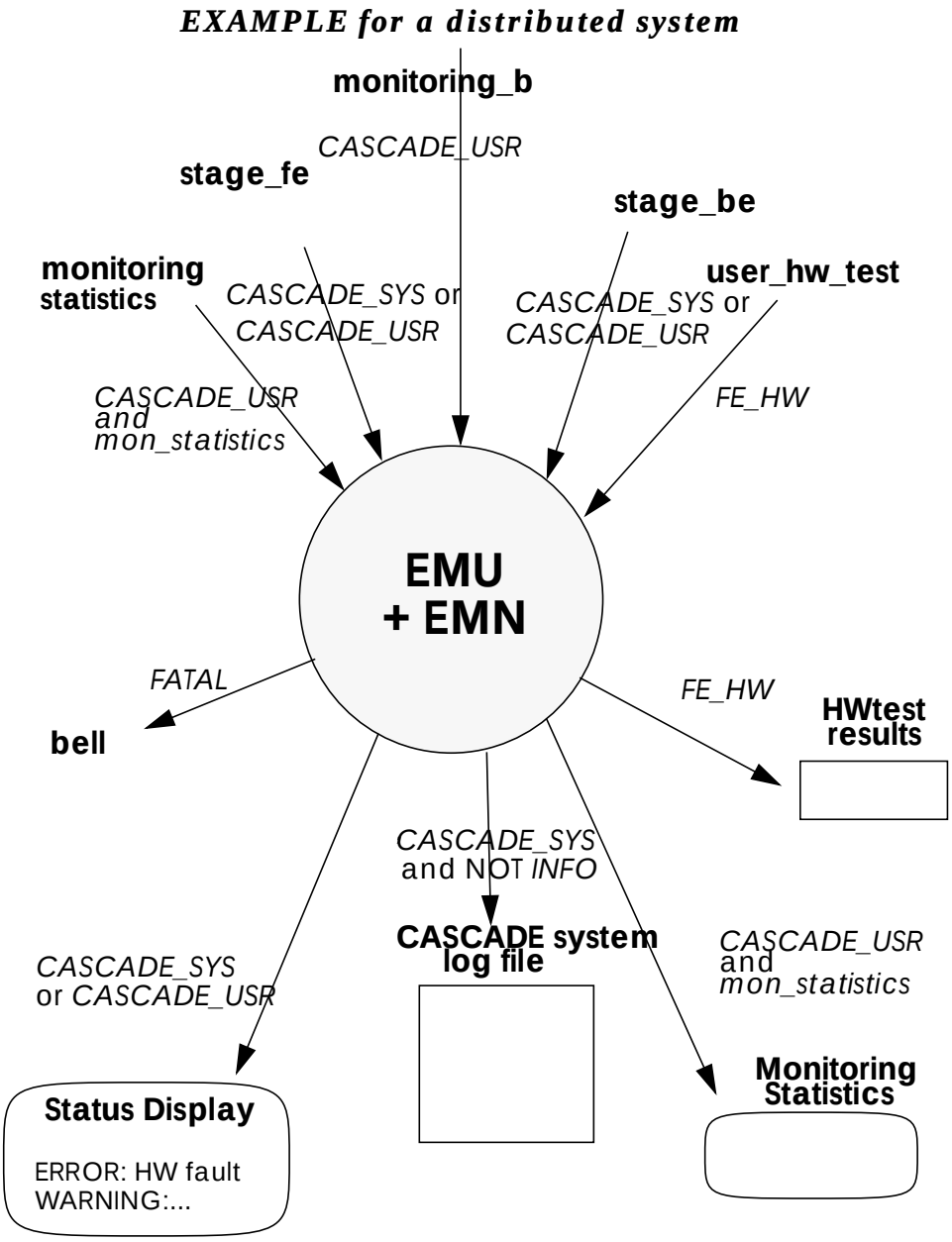
The properties mentioned in Section 12.4.1 and Section 12.4.2 are reserved names within the CASCADE EMU setup. This list of reserved properties is not exhaustive; others may be added at a later time. Properties with the name CASCADE* must not be defined by a CASCADE user or CASCADE independent user. Once they have been defined by the CASCADE group it is up to this group to give permission to the user for using it, depending on the actual purpose of the property.

Capitals shall be used for the definition of EMU keywords defined by the CASCADE group.

### 12.4.4 Example

Figure 13 illustrates the use of CASCADE message streams and severities for the example shown in Figure 11.

**Figure 13** Using CASCADE error streams and severities for CASCADE error and message handling




## 12.5 Message injection from CASCADE into EMU

A function is provided to serve as an interface between the message sender and EMU. This avoids having to call a number of EMU injection routines per message and adds **automatically** the CASCADE error stream property **CASCADE_USR** for CASCADE user message and **CASCADE_SYS** for CASCADE system messages. The CASCADE independent user must not use any of these routines.

## EMH_UsrMsg and EMH_SysMsg

### Synopsis

```
#include emh.h
void EMH_SysMsg(int sw, int sev, char *msgid, char *fmt,...)
```

### Parameters

| Type | Name | Description |
|---|---|---|
| int | sw | output direction switch |
| enum | sev | CASCADE EMU message severity |
| char | msgid[] | message name id |
| char | fmt[] | message parameter formats |
| p1,p2 p3,... | | optional message parameters |

### Description

EMH_...Msg transforms the passed parameters into a sequence of EMU calls if the switch is set to EMU or otherwise into a printf statement.

**sw** is a switch to indicate to EMH_...Msg whether it should use EMU or another output mechanism. Allowed options are:

EMH_EMU: transform into calls to EMU injection routines (report via EMU)

EMH_STDOUT : report using printf (stdout)

EMH_STDERR : report using fprintf (stderr)

EMH_NULL : do not report the information

A combination of these symbols can be used by oring the options with the exception of EMH_NULL .

**sev**

defines an EMU message property corresponding to a severity via an enumerated type. One of the following severities must be used:

FATAL, ERROR, WARNING, INFO, SUCCESS

**msgid** is a name to identify a message. If this message id is defined in the EMU message file then the message text is retrieved from there. Otherwise the message text must be passed as part of the parameters in the call to EMH_...Msg.

**fmt** is a printf like format string which defines the number and the type of the arguments to follow. Allowed are a subset of the printf formats and a number of EMU specific formats. Text string can be present in the fmt parameter within double quotes. Currently allowed formats are:

%d: decimal integer (int sw, int sev, char msgid[], char fmt[],p1,p2 p3,.....)

%f: floating point number

%s: string

%x: unsigned hexadecimal integer

%r: EMU property

%k: EMU package identifier

other types may be added. Text can be present in the fmt parameter.

**p1, p2, p3**...are a variable number of arguments of possible different types including EMU specific types as described in **fmt.**The number of arguments must be equal to the number of formats included in **fmt.**

## Examples

1. The message identifier “Mess1” is not present in the message file (direct emu call):

```
EMH_SysMsg(EMH_EMU,FATAL,”Mess1”,“and without additional prop: voila %d”,1234);
```

   generates this EMU message:

```
MESS‘Mess1‘PROG‘demo‘MACH‘sunom4‘INST‘7830‘DATE‘1994/02/03‘TIME‘15:35:27‘PROP‘CASCADE
_SYS‘PROP‘FATAL‘TEXT‘and without additional prop: voila 1234‘EOR ‘‘
```

2. The message identifier “Mess2” is not present in the message file:

```
EMH_SysMsg(EMH_EMU,ERROR,”Mess2”,“Hello: %d, with the prop:%p“,123,”PROPI”);
```

   generates this EMUmessage:

```
MESS‘Mess2‘PROG‘demo‘MACH‘sunom4‘INST‘7830‘DATE‘1994/02/03‘TIME‘15:35:27‘PROP‘CASCADE
_SYS‘PROP‘ERROR‘PROP‘PROPI‘TEXT‘Hello : 123, with the prop: ‘EOR ‘‘
```

   Text can be present in the format string. It is concatenated with the next EMH_xxxMsg parameter to become ONE emu parameter.

3. With the following message file installed:

```
-- begin message file
PROG demo
   MESS colours
     TEXT“who’s afraid of”
     TEXT“, “
     TEXT“ and “
-- end message file
```

   - a call like:

```
EMH_SysMsg(EMH_EMU,ERROR,”colors”,“%s%s%s”,”red”,”blue”,”yellow”)
```

     will send to emu the string “who’s afraid of red, blue and yellow”
   - while a call like:

```
EMH_SysMsg(EMH_EMU,ERROR,”colors”,“light%sdark %s%s”,”red”,”blue”, yellow”)
```

parameters sent to EMU by EMH_xxxMsg:

1st parameter: light red - 2nd parameter:dark blue - 3rd parameter: yellow

will send to EMU the string “who’s afraid of light red, dark blue and yellow”

## 12.6 EMU Message Decoding

The CASCADE system specific file cascade_emu.deco is being provided by CASCADE. This file must not be modified by the user. It must be concatenated with the experiment specific decoder message file in order to produce a single input file for the EMU message decoder ecdeco. The decoder output file deco.map is then used as the input file when starting emudeco.

## 12.7 EMU Message Routing

The file emu.rout must be built and compiled by ecrout, which produces the output file rout.map. This file is used as the input file when starting emurout. This procedure must be followed for each system involved.

It is recommended to rout all messages of type CASCADE_SYS and CASCADE_USR message stream to a central log file and to a central display.

In order to do so the rout file for a *front end* system has to include:

```
remote_system: FIFO:=’/tmp/remote_system’;
```

*and*

```
remote_system =>
(PROP = CASCADE_SYS OR PROP=CASCADE_USR);
```

In order to do so the rout file for the *central system* has to include:

```
-- declare destinations
    cascade_pipe:= FIFO:=’/tmp/cascade_error_message’;
    cascade_error_log:= LOGFILE:=’/tmp/cascade_error_log’;

-- define synonyms
    CASCADE:= (PROP = CASCADE_SYS OR PROP = CASCADE_USR);

-- routing selection
    cascade_pipe => CASCADE;
    cascade_error_log => CASCADE;
```

## 12.8 Example

An example is explained in detail. The necessary EMU and EMN setup is described as well as the different steps from the message injection up to the message destination.

### 12.8.1 Installation

The EMU related products are included in the CASCADE system. They normally are started at system startup and therefore available for general use.

In the normal case the compilers ecdeco and ecrout are installed. The processes emudeco and emurout must be active on the connected systems. The process emunet_clnt must be active on the client system and the process emunet_serv on the server system.

The files deco.map, rout.map and emunet.info must be available under the corresponding directories. If the decoder description file or the router description file changes than it must be recompiled with ecdeco or ecrout respectively. Then emudeco and emurout must be stopped and started with the new map files. If the file emunet.info is modified then the emunet_clnt must be stopped and started with the new version.The script **emuStart** can also be used to start up the corresponding processes. It is described in the chapter on building a CASCADE system.

There is also a guide available to help in installing all the EMU related products and files [24].

### 12.8.2 Example Configuration

An example similar to the one shown in Figure 11 is used:

LYNXOS - TCP/IP machine name: essosly03

SUN - UNIX - TCP/IP machine name: suds01

In this example messages with their message stream property may be sent from

- userprod and/or userevb from the fe-stage: CASCADE_USR
- the CASCADE system code from the fe-stage: CASCADE_SYS
- userprod and/or userevb from the be-stage on the SUN: CASCADE_USR
- the CASCADE system code from the be-stage on the SUN: CASCADE_SYS
- a monitoring program on the SUN: CASCADE_USR

As message destination we define the EventDisplay, a log file and a process triggering a bell.

### 12.8.3 EMN setup

As further explained in the installation guide [24] the EMN client emunet_clnt must in this case be installed on the LYNXOS system - the sending system - and the EMN server emunet_serv must be installed on the destination system, the UNIX system. The file **emunet.info** must be available on the client side. The first line lists a pipe name. This pipe name is the EMU destination pipe on the local system, here LYNXOS, as defined in the local rout.map file. The second line contains the TCP/IP machine name of the network destination:

/tmp/remote_sun

suds01

This information is used by the emunet_clnt on the LYNXOS system to send the messages which appear in the pipe /tmp/remote_sun over TCP/IP to the emunet_clnt on the remote system suds01 There they are automatically injected into the EMU system on suds01 in order to be processes in the standard EMU way.

### 12.8.4 The EMU Message file

- The message file would be:

-- CASCADE EMU message file -- example1 D.B. 01.03.94

```
PROGRAM fe_stage
   MESSAGE open_file
     PROPERTY DATABASE
     TEXT “received error”
     TEXT “when opening file”
   MESSAGE start_run
     PROPERTY CONTROL
     TEXT “Run”
     TEXT “started on My_OS9 ”
   MESSAGE stop_run
     PROPERTY CONTROL
     TEXT “Run”
     TEXT “stopped on My_OS9”
   MESSAGE no_trigg
     PROPERTY HW
     TEXT “ No trigger since 10 sec.”
     PROGRAM be_stage
   MESSAGE open_file
     PROPERTY DATABASE
     TEXT “received error”
     TEXT “when opening file”
   MESSAGE start_run
     PROPERTY CONTROL
     TEXT “Run”
     TEXT “started on My_SUN ”
   MESSAGE stop_run
     PROPERTY CONTROL
     TEXT “Run”
     TEXT “stopped on My_SUN”
PROGRAM monitoring_be
   MESSAGE good_events
     PROPERTY MONIT
     TEXT “Fantastic event received”
```

- This message file can be used for both the LYNXOS and for the SUN system. However only the messages belonging to the program stage_fe are necessary for the LYNXOS system and the messages for the programs stage_be and monitoring_be are necessary for the SUN.

- CASCADE_SYS or CASCADE_USR is set by the interface routines to the EMU injection EMH_SysMsg or EMH_UsrMsg respectively.
- If the same message is to be used by different programs for the same purpose then one can specify it under the name of a package instead the program name. In this case the package name must be added to the EMU injection routines or to EMH_...Msg.

### 12.8.5 The EMU Router file

Note: In this example more properties than actually used for the routing have been assigned to some messages for demonstration purposes.

- Router file for the LYNXOS system:

```
-- declare destinations
remote_sun: FIFO:=’/tmp/remote_sun’;
local_log :=LOGFILE:=’/dd/tmp/hw_local_error_log’;
-- define synonyms
CASCADE:= (PROP = CASCADE_SYS OR PROP = CASCADE_USR) ;
-- routing selection
remote_sun => CASCADE;
local_log => (PROP = HW);
```

- Router file for the SUN system

```
-- declare destinations:
alarm_pipe: FIFO :=’/tmp/alarm_message’;
cascade_pipe :FIFO:=’/tmp/cascade_error_message’;
monitoring_pipe :FIFO:=’/tmp/monitoring_message’;
message_log_file :LOGFILE:=’/tmp/cascade_error_log’;
bell :=EXTERNAL:= ’/tmp/emu_bell’ ;
-- define synonyms
CASCADE:= (PROP = CASCADE_SYS OR PROP = CASCADE_USR);
-- routing selection
alarm_pipe => (PROP = ALARM);
cascade_pipe => CASCADE;
monitoring_pipe => (PROP = MONIT);
message_log_file => (PROP = ‘ALL”);
bell => (PROP = FATAL);
```

### 12.8.6 Message injection

Making use of the defined message in the decoder files the message injection to EMU via the EMH_UsrMsg and EMH_SysMsg routines can look as follows:

- for the user part of the fe_stage:

```
EMH_UsrMsg(EMH_EMU,INFO,”start_run”,”%d”,run_num)
```

EMH_UsrMsg(EMH_EMU,INFO,”stop_run”,”%d”,run_num)

EMH_UsrMsg(EMH_EMU,FATAL,”no_trigg”)

- for the system part of the fe_stage:

  EMH_SysMsg(EMH_EMU,
  ERROR,”open_file”,”%r%x%s”,”MY_FE”,my_error,
  ”database.txt”) /* can add property MY_FE */

- for the user part of the be_stage::

  EMH_UsrMsg(EMH_EMU,INFO,”start_run”,”%d”,run_num)

  EMH_UsrMsg(EMH_EMU,INFO,”stop_run”,”%d”,run_num)

- for the system part of the be-stage:

  EMH_SysMsg (EMH_EMU,
  ERROR,”open_file”,”%r%x%s”,”MY_BE”,my_error,
  ”database.txt”) /* can add property MY_BE */

- for the monitoring program:

  EMH_UsrMsg(EMH_EMU,SUCCESS,”good_events”)

### 12.8.7 EmuDisplay and logfile

- EmuDisplay must be connected to ‘/tmp/cascade_error_message’. Then all messages flagged with CASCADE_SYS or CASCADE_USR are received and can be displayed in groups according to other criteria, for example the various severities, machine of origin, package or others [23].
- In the rout file it has been defined that all messages from all programs must be added to the log file /tmp/cascade_error_log. Whenever necessary this file can be examined manually or by a dedicated tool.

  In this example the log file may contain raw message like:

  MESS‘open_file‘PROG‘fe_stage‘MACH‘sunom4‘INST‘5360‘DATE‘1994/02/10‘
  TIME‘11:22:45‘PROP‘CASCADE_SYS‘PROP‘ERROR’PROP’DATABASE‘TEX
  T‘received error 24 when opening file database.txt‘EOR ‘‘

  MESS‘start_run‘PROG‘fe_stage‘MACH‘sunom4‘INST‘5360‘DATE‘1994/02/
  10‘TIME‘10:12:32‘PROP‘CASCADE_USR‘PROP‘INFO’PROP’DATABASE‘PR
  OP’MY_FE’TEXT‘received error 24 when opening file database.txt‘EOR ‘‘

# 13 Configuration Files

The purpose of the configuration file is to specify the topology of the application data acquisition system in terms of CASCADE elements (stage and inter-stage links) as well as the characteristics of each of these elements (mapping the logical representation to the physical implementation). The configuration file can have any name (default is **`daqconf`**). When they start, all stages read it and initialise their internal structures accordingly.

## 13.1 Overall File Structure

The configuration file is an ASCII file. It contains system global parameters and as many stage entries as there are stages in the configuration. The inter-stage links are not declared via dedicated entries but they are specified via the input and output port entries of the stages they are connecting together.

< system global parameters section >

< first stage entry>

.

.

< last stage entry>

Comments can be added anywhere in a configuration file. A comment starts with an exclamation mark (!) and terminates at the end of the same line.

## 13.2 Global System Parameters

A user event header and a user event trailer are created (respectively at the front and at the end of the event) whenever an event is created in a stage. The sizes of these two areas are application specific but are the same for all stages of the configuration. Application specific event production functions supply the user with the addresses of these areas so that they can be filled. Events resulting from an event building operation contain contiguous (sub)events, each of them with its own header and trailer, which are themselves encapsulated with a user header and a user trailer for the newly built event. An application specific event building function permits to reduce, if desired, the subevents header and trailer prior to the assembly of the full event.

HEADER_SIZE = hd_size

TRAILER_SIZE = tr_size

where:

**hd_size** is the number of 32 bit words to allocate for the user header

**tr_size** is the number of 32 bit words to allocate for the user trailer

## 13.3 Stage Entry

A stage entry contains two mandatory sections: the stage global parameters section and the stage input ports section. Depending on the location and on the function of a stage in the system, its stage entry may include one or two additional sections which are the stage output port section and the stage event type section. A stage entry has the following format:

```
STAGE_NAME = name
   < stage global parameters section >
   < stage input ports section >

   [stage output port section]
   [stage event type section]
END_STAGE
```

where:

**name** is the name of the stage

### 13.3.1 Global Stage Parameters Section

This section contains the main characteristics of the stage and it has the following format:

```
CONSTRUCTION_TYPE = ctype
STAGE_SEGMENT_SIZE = seg_size
NB_INPUTS = nb_in
NB_OUTPUTS = nb_out
```

where:

**ctype** should be set to EVB is the stage has to perform event building and to DUMMY otherwise

**seg_size** is the amount of space (in bytes) to be reserved for the stage shared segment. This shared segment is used for the stage internal structures and also for event buffering and event building (if ctype has been set to EVB). Therefore the shared segment should have a minimum size of 100000 bytes plus three times the size of the largest event type to build. It is strongly recommended to give stages a reasonably large buffer space as a function of the expected event rate and event sizes. By default the stage shared segment will be located in the system (CPU) memory. However, if the stage has one of its outputs connected to an other stage via a VICbus connection, the shared segment will be located in the VIC RFM memory.

**nb_in** is the number of input ports of the stage

**nb_out** is the number of output ports of the stage

### 13.3.2 Stage Input Ports Section

This section contains the characteristics of all the input ports of the stage. Every input port should be described by one input entry which has the following format:

```
INPUTi [= name]
   CONNECTED_STAGE = csname
   CONNECTED_STAGE_PORT = port - 1
   < inter-stage link characteristics >
END_INPUT
```

where:

**i** is the stage input port number (starting from 0)

**name** is an optional input port identifier name

**csname** is the name of the stage connected to this input port

**port** is the output port number used for this link in the connected stage (Note that, at present, it is the value **port-1** which has to be assigned to CONNECTED_STAGE_PORT)

The inter-stage link characteristics depend on the type of link connected to this input port (inter-stage link or detector/stage link). Full details of the inter-stage characteristics to be specified are given below for all the types of links supported at present.

### 13.3.3 Stage Output Ports Section

This section contains the characteristics of all the output ports of the stage. Every output port should be described by one output entry which has the following format

```
OUTPUTj
   CONNECTED_STAGE = csname
   TRANSFER_MODE = mode
   < inter-stage link characteristics >
END_OUTPUT
```

where:

**j** is the stage output port number (starting from 0)

**csname** is the name of the stage connected to this output port

The TRANSFER_MODE characteristic is optional. **mode** can be set to either ALL or IF_FREE. Ports with a transfer mode set to ALL (default mode) get, by definition, all events but with the inconvenience that an event is not considered for output as long as the previous event has not been output and acknowledged by all the output ports with a transfer mode set to ALL.

Ports set to the IF_FREE mode get events if, by the time they are considered for output, they are not busy to output a previous event. This mode of operation has the advantage that slow ports which don't need to see all events (e.g output to a backend stage with no output and 0% monitoring) do not slow down the overall

acquisition. The overall system throughput in such a case is limited by the speed of the slowest output port set to transfer ALL events.

The inter-stage link characteristics depend on the type of link connected to this output port (inter-stage link or detector/stage link). Full details of the inter-stage characteristics to be specified are given below for all the types of links supported at present.

## 13.4 Inter-stage Links

The inter-stage links are specified via the input and output port entries of the stages they are connecting together. Since the specification parameters vary significantly from one type of link to an other, the inter-stage link characteristics to be specified in the input and output port entries of the connected stages are described below as a function of the inter-stage link type.

### 13.4.1 VICbus links

The following characteristics have to be specified in the output port of the upstream stage:

FORMAT = MEM_LINK

VIC_NAME = device_name

VIC_CRATE = crate

where:

**device_name** is the name of the local device to use to access VICbus

**crate** is the VIC crate number of the partner stage (0 if the partner is the event builder)

The following characteristics have to be specified in the input port of the downstream stage:

TYPE = MEM_LINK

VIC_NAME = device_name

VIC_CRATE = crate

where:

**device_name** is the name of the local device to use to access VICbus

**crate** is the VIC crate number of the partner stage

### 13.4.2 Network Links

The following characteristics have to be specified in the output port of the upstream stage:

FORMAT = NET_LINK
CONNECTED_STAGE_ADDR = partner_ip

where:

**partner_ip** is the ip number of the partner stage system

The following characteristics have to be specified in the input port of the downstream stage:

TYPE = NET_LINK
CONNECTED_STAGE_ADDR = partner_ip
TCP_PORT = 6001

where:

**partner_ip** is the ip number of the partner stage system

### 13.4.3 Stage to Recorder Shared Memory links

The shared memory link is at present reserved to stage to recorder links.

The following characteristics have to be specified in the output port entry of the stage:

FORMAT = ZEBRA
REC_SIZE = size

where:

**size** is the physical record size in 32 bit words

The following characteristic has to be specified in the input port of the recorder:

TYPE = STAGE

### 13.4.4 Detector to Front-end Stage links

Entries corresponding to stage input ports directly connected to electronics have to be modified as follows:

CONNECTED_STAGE = NONE
TYPE = USER

to indicate that the input has to be handled by application specific event production functions

## 13.5 Additional Stage Information Related to Event Building

The stage entry of an event builder stage differs from other stage entries essentially by the contents of its stage global parameters section and by the fact that it includes an additional section called the stage event type section.

The stage global parameters section of an event builder stage should have the CONSTRUCTION_TYPE parameter set to EVB. It should also include an additional parameter called NB_EVTYPES which should be set to the number of event types for which an event building operation is necessary.

### 13.5.1 The stage event type section

Every type of events resulting from an event building operation needs an EVTYPE entry where the parameters associated with the building operation are specified.

```
EVTYPEk = etname
    NB_COMPONENTS = nc
    PACKING = pk
    < first component entry >
    < last component entry >
END_EVTYPE
```

where:

**k** is the event type number (starting from 1)

**etname** is the event type name

**nc** is the number of component types involved in the building operation. In this context a component type means either the type of subevent originating from an input port or a type previously declared by an EVTYPE entry.

**pk** specifies how events of this type have to be delivered to the rest of the stage (ENABLE indicates that components should be concatenated into a single event whereas DISABLE means that components should be passed as a series of individual events)

A component entry has the following format:

```
COMPONENTx = name presence
[FACTOR_COMPONENTx = fc]
```

where:

**x** is the component number (starting from 1)

**name** is the name of the component origin (input port or other event type)

**presence** has to be set to PRESENT or ABSENT depending whether the building operation requires this component to be PRESENT or ABSENT.

FACTOR_COMPONENT is an optional characteristic which is only necessary in case the building operation of this event type requires several instances of component x. In that case, **fc** is:

either the required number of instances

or the keyword USER to indicate that the required number of instances will be obtained at run time from a user function.

or the keyword CONTROL to indicate that this component is used only as an event building condition and will not result into any (sub)event data.

## 13.6 Configuration File Templates

The configuration file templates are examples of configuration files which may serve as starting points for developments of user applications. Two simple as well as a more complex examples are provided. The main purpose of the validation configuration file is to allow the CASCADE developers to perform a test or “validation” of a CASCADE release.

The templates provided as part of the distribution kit are available in the Cascade ’online/templates’ directory under the names `camac.daqconf_demo`, `corbo.daqconf_demo` and `valid.daqconf`.

When copied automatically in a user directory by the `copy_templates_demo script`, the two demo configuration files are renamed to: `camac.daqconf` and `corbo.daqconf`

- `camac.daqconf`

  A configuration with a stage triggered by a CAMAC module and output to tape and disk, in Zebra format, via a tape recorder and a disk recorder stage.
- `corbo.daqconf`

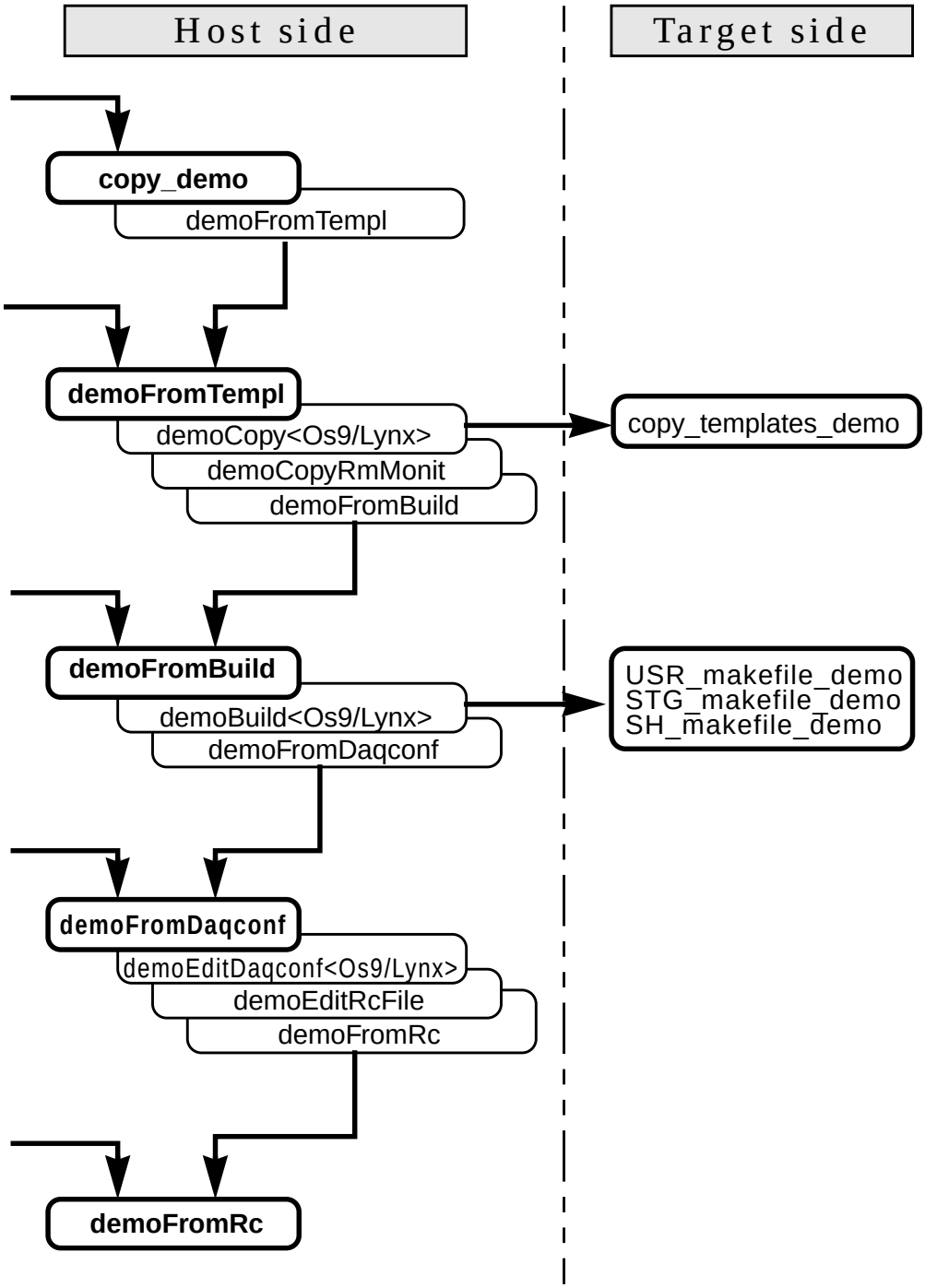
  A configuration with a stage triggered by a single CORBO channel and output to tape and disk, in Zebra format, via a tape recorder and a disk recorder stage.
- `valid.daqconf`

  This is the configuration file for the validation of a CASCADE release. It is a special and complex example describing a subset of the CASCADE configuration of the NOMAD data acquisition system. In a certain sense, it is not a template since it is not supposed to be developed further by the user. It can however be used as an example for large applications using event building.

# A The demo scripts suite

A suite of scripts files is distributed to automate the process of building and executing the Cascade demo and to help users to develop their own application.

The **copy_demo** script does the whole set of operations starting from scratch and getting the demo running at the end. It calls in sequence a number of other scripts. The following diagram shows this sequence and which intermediate entry points exits if only part of the procedure needs to be redone.



The demo scripts automatically:

- copy the necessary files in user work directories (on both the front-end and back-end systems)
- build the various processes (stage, monitoring programs)
- adapt the scripts: start.stage, start.rectape, start.recdisk to the application characteristics: machine names, stage names,
- generate the application configuration file (daqconf.upd)
- generate the input file for the run control mSQL data base (democonfig)
- launch execution of the run control.

# References

The following documents provide additional information.


**[1]** Creative Electronic Systems SA, **VIC 8251F User's Manual**, version 0.1, Geneva, Switzerland.

**[2]** Hughes Technologies Pty Ltd, **Mini SQL A lightweight database Engine** , Release 1.0.11, Jan 1996.

**[3]** Vande Vyvre, P., **Buffer and List Handling Classes**, CERN ECP/DS, 1992.

**[4]** Creative Electronic Systems SA, **FIC8234 User's Manual**, version 0.5, Geneva, Switzerland.

**[5]** J.O. Petersen, **CORBO user routines**, ECP/DS 93-11.

**[6]** J.O. Petersen, **A simple, general purpose interrupt handler for OS-9**, ECP/DS 93-12.

**[7]** Creative Electronic Systems SA, **CBD8210 CAMAC branch driver User's Manual**, Geneva, Switzerland.

**[8]** Creative Electronic Systems SA, **VCC 2117/A CAMAC Crate Controller User's Manual**, version 1.1, Geneva, Switzerland.

**[9]** Creative Electronic Systems SA, **FVSBI 9210 FASTBUS-to-VSB Interface User's Manual**, Geneva, Switzerland.

**[10]** Creative Electronic Systems SA, **FVSBI FASTBUS Library**, Geneva, Switzerland.

**[11]** Creative Electronic Systems SA, **RCB8047 CORBO VME Read-Out Control Board User's Manual**, Geneva, Switzerland.

**[12]** Meijers, F., Petersen, J., **A VME intercrate message system based on VICbus and OS-9**, CERN ECP/DS, 1992.

**[13]** Petersen, J., **A 'High-Level' Driver for the STORAGETEK STK4280 (SUMMIT)**, CERN ECP/DS OS9SCSI2, August 1991.

**[14]** Petersen, J., **A 'High-Level' Driver for the Exabyte**, CERN ECP/DS OS9SCSI5, August 1991.

**[15]** Saravia, A., **Standard Remote Shell for OS9**, OPAL Note 15/ONLINE-0474.

**[16]** Segal, B., **TCP/IP package for remote-PAW**, February 1990.

**[17]** Scharff-Hansen, P., **Use of TCPAW on OS9**, February 1990.

**[18]** Segal, B., **Installation and use of the TCPAW package**, Octobre 1990.

**[19]** Saravia, A., **The Network Inter-Stage Communication package**, ECP/DS 93/25

**[20]** Burkimsher, P. C., **EMU, the MODEL Error Message Utility**, version 2.1 December 1990

**[21]** Meijers, F., **EMUX: Error Message Utility for OS-9 and POSIX**, [html] **[**ps**]** version 2.0 January 1992

**[22]** Burckhart, D., **A TCP/IP network connection for EMU: Emunet**, [frame] **[**ps**]** CERN ECP/DS 93-27 V01.0, July 1993.

**[23]** Werner, P., **Emu Display facilities**, [frame] **[**ps**]** CERN ECP/DS 93-30

**[24]** Burckhart, D., P. Werner, **Installation guide for the Error Message Utility products EMU, Emunet and EmuDisplay on UNIX and OS9**, [frame**] [**ps**]** CERN ECP/DS 93-28 V01.0

**[25]** Gallot, Y., **A Portable Labelling Package**, CERN DD/OC OS9SCSI3, June 1990.

UserGuideREF.doc (Reference: [1] Creative Electronic Systems SA, VIC 8251F User's Manual, version 0.1,) 3
UserGuideREF.doc (Reference: [2] Hughes Technologies Pty Ltd, Mini SQL A lightweight database Engine ,) 6
UserGuideREF.doc (Reference: [17] P.C. Burkimsher, EMU, the MODEL Error Message Utility, version 2.1) 6
UserGuideREF.doc (Reference: [18] F. Meijers, EMUX: Error Message Utility for OS-9 and POSIX, version 2.0) 6
UserGuideREF.doc (Reference: [19] D. Burckhart, A TCP/IP network connection for EMU: Emunet, CERN ECP/) 6
UserGuideREF.doc (Reference: [20] P. Werner, Emu Display facilities, CERN ECP/DS 93-30) 6
UserGuideREF.doc (Reference: [4] Creative Electronic Systems SA, FIC8234 User's Manual, version 0.5, Geneva,) 7
UserGuideREF.doc (Reference: [7] Creative Electronic Systems SA, CBD8210 CAMAC branch driver User's) 7
UserGuideREF.doc (Reference: [8] Creative Electronic Systems SA, VCC 2117/A CAMAC Crate Controller) 7
UserGuideREF.doc (Reference: [9] Creative Electronic Systems SA, FVSBI 9210 FASTBUS-to-VSB Interface) 7
UserGuideREF.doc (Reference: [10] Creative Electronic Systems SA, FVSBI FASTBUS Library, Geneva, Switzer) 7
UserGuideREF.doc (Reference: [11] Creative Electronic Systems SA, RCB8047 CORBO VME Read-Out Control) 7
UserGuideREF.doc (Reference: [1] Creative Electronic Systems SA, VIC 8251F User's Manual, version 0.1,) 7
UserGuideREF.doc (Reference: [12] Meijers, F., Petersen, J., A VME intercrate message system based on VICbus) 7
UserGuideREF.doc (Reference: [13] Petersen, J., A 'High-Level' Driver for the STORAGETEK STK4280 (SUM) 7
UserGuideREF.doc (Reference: [14] Petersen, J., A 'High-Level' Driver for the Exabyte, CERN ECP/DS) 7
getsoft.frame (Head_1: 2.2 Which software to download where) 9
getsoft.frame (Head_1: 2.3 Setting up the directory infrastructure necessary to host CASCADE) 9
UserGuideREF.doc (Reference: [2] Hughes Technologies Pty Ltd, Mini SQL A lightweight database Engine ,) 10
UserGuideREF.doc (Reference: [15] A. Saravia, Standard Remote Shell for OS9, OPAL Note 15/ONLINE-0474.) 10
UserGuideREF.doc (Reference: [19] P. Scharff-Hansen, Use of TCPAW on OS9, February 1990.) 11
UserGuideREF.doc (Reference: [20] B. Segal, Installation and use of the TCPAW package, Octobre 1990.) 11
demoappendix.frame (Chapter: 3 Required Layered Products) 21
producer.frame (Head_1: 7.3 Event headers and event production templates) 23
overview.frame (Head_2: 1.2.2 The Inter-Stage Link) 23
lmonitor.frame (Chapter: 9 Event monitoring) 25
rmonitor.frame (Chapter: 10 Remote monitoring facility) 25

daqconf.frame (Chapter: 14 Configuration File) 25
runcontrol.frame (Head_1: 12.3 The CASCADE control file) 25
errhand.frame (Chapter: 13 Error and Message Handling and Reporting) 25
startappl.frame (Chapter: 5 Loading and Starting a CASCADE System) 25
runcontrol.frame (Chapter: 12 Run control) 27
UserGuideREF.doc (Reference: [15] A. Saravia, Standard Remote Shell for OS9, OPAL Note 15/ON-LINE-0474.) 27
buildappl.frame (Chapter: 4 Building a CASCADE System) 27
recorder.frame (Chapter: 11 Data Recording) 30
recorder.frame (Chapter: 11 Data Recording) 30
recorder.frame (Chapter: 11 Data Recording) 30
recorder.frame (Chapter: 11 Data Recording) 36
buildappl.frame (Chapter: 4 Building a CASCADE System) 48
buildappl.frame (Chapter: 4 Building a CASCADE System) 64
UserGuideREF.doc (Reference: [1] Creative Electronic Systems SA, VIC 8251F User's Manual, version 0.1,) 72
daqconf.frame (Chapter: 14 Configuration File) 81
daqconf.frame (Chapter: 14 Configuration File) 81
buildappl.frame (Chapter: 4 Building a CASCADE System) 81
producer.frame (Head_1: 7.3 Event headers and event production templates) 82
buildappl.frame (Chapter: 4 Building a CASCADE System) 82
lmonitor.frame (Chapter: 9 Event monitoring) 105
lmonitor.frame (Chapter: 9 Event monitoring) 106
UserGuideREF.doc (Reference: [18] A.Saravia, The Network Inter-Stage Communication package, ECP/DS 93/25) 106
UserGuideREF.doc (Reference: [17] Ben Segal, TCP/IP package for remote-PAW, February 1990.) 106
lmonitor.frame (Chapter: 9 Event monitoring) 107
lmonitor.frame (Chapter: 9 Event monitoring) 108
lmonitor.frame (Function: sh_wait) 110
lmonitor.frame (Function: sh_wait) 110
lmonitor.frame (Function: sh_mconnect) 111
lmonitor.frame (Function: sh_request_event) 112
lmonitor.frame (Function: sh_wait) 112
lmonitor.frame (Function: sh_get_data) 112
lmonitor.frame (Function: sh_wait) 112
lmonitor.frame (Function: sh_get_data) 112
lmonitor.frame (Function: sh_wait) 113
lmonitor.frame (Function: sh_request_event) 114
lmonitor.frame (Function: sh_wait) 114
lmonitor.frame (Function: sh_get_data) 114
lmonitor.frame (Function: sh_release_event) 115
lmonitor.frame (Function: sh_disconnect) 116
lmonitor.frame (Function: sh_message) 117
UserGuideREF.doc (Reference: [17] Ben Segal, TCP/IP package for remote-PAW, February 1990.) 118
infrastructure.frame (Chapter: 3 Required Layered Products) 120
UserGuideREF.doc (Reference: [17] Ben Segal, TCP/IP package for remote-PAW, February 1990.) 120
UserGuideREF.doc (Reference: [13] Petersen, J., A 'High-Level' Driver for the STORAGETEK STK4280 (SUM) 123
UserGuideREF.doc (Reference: [14] Petersen, J., A 'High-Level' Driver for the Exabyte, CERN ECP/DS) 123

startappl.frame (Chapter: 5 Loading and Starting a CASCADE System) 124
UserGuideREF.doc (Reference: [24] Gallot, Y., A Portable Labelling Package, CERN DD/OC OS9SCSI3, June) 127
UserGuideREF.doc (Reference: [17] P.C. Burkimsher, EMU, the MODEL Error Message Utility, version 2.1) 135
UserGuideREF.doc (Reference: [18] F. Meijers, EMUX: Error Message Utility for OS-9 and POSIX, version 2.0) 135
UserGuideREF.doc (Reference: [19] D. Burckhart, A TCP/IP network connection for EMU: Emunet, CERN ECP/) 135
UserGuideREF.doc (Reference: [20] P. Werner, Emu Display facilities, CERN ECP/DS 93-30) 136
UserGuideREF.doc (Reference: [26] D. Burckhart, P. Werner, Installation guide for the Error Message Utility) 143
UserGuideREF.doc (Reference: [26] D. Burckhart, P. Werner, Installation guide for the Error Message Utility) 143
UserGuideREF.doc (Reference: [20] P. Werner, Emu Display facilities, CERN ECP/DS 93-30) 146