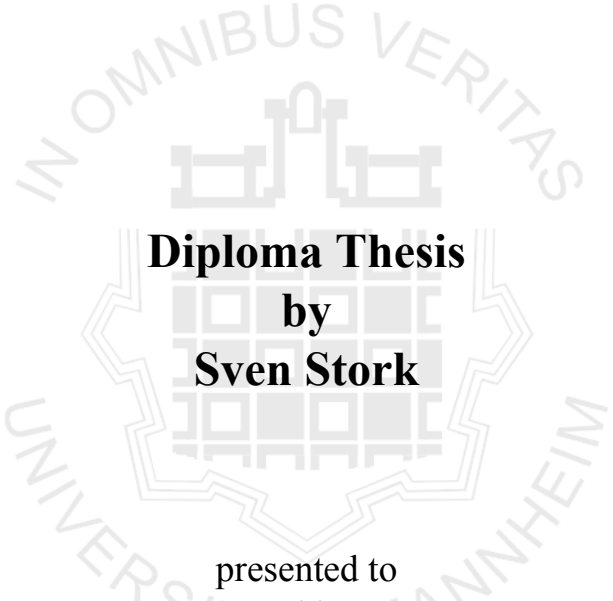
Design of an efficient Software Environment for a RDMA Network Interface Controller



Diploma Thesis
by
Sven Stork

presented to
Computer Architecture Group
Department of Computer Engineering
University of Mannheim

11 January 2006

Referee	: Prof. Dr.-Ing. Ulrich Brüning
Co - Referee	: Prof. Dr. K.-H. Brenner
Supervisor	: Dipl.-Inf. Mondrian Nüssle

Abstract

The goal of this thesis is to evaluate the requirements, the design and the implementation of an efficient software interface for the Extoll NIC. The resulting software is called Extoll Software Stack (ESS).

The design of the ESS should be efficient and exploit the maximum performance that will be offered by the Extoll NIC. The software interface should be clear and intuitive. An important point is the optimization of the software interface for an easy and efficient collaboration with already existing middle-wares, like MPI and DAPL.

The first part of the thesis gives an overview of the currently available communication interfaces. The next part evaluates the Extoll NIC hostport interface. Based on the determined hostport interface the ESS design will be presented. The last part of the thesis gives an overview of the current implementation and testing environment of the ESS.

Contents

List of Figures

List of Tables

List of Listings

Introduction

CHAPTER 1

In 1999, the Computer Architecture Group, University of Mannheim, started a research project called Atoll ([ATOLL]). The goal of the Atoll project was to develop a complete SAN for high performance computing (HPC). Besides the NIC a complete software stack, consisting of a management software and an application programming interface (API) called PALMS and a management Daemon, has been developed by Mondrian Nuessle ([NUS03]).

The Atoll SAN proved to be functional and efficient. Extended information about Atoll and Palms can be found in “Atoll Basics” on page 19 and “Atoll - PALMS” on page 7. For additional information about the Atoll design and the Atoll performance refer to [ATOLL99], [ATOLL00], [ATOLL02], [ATOLL03] and [RZY97]. During the development and usage phase of the Atoll SAN several drawbacks showed up caused by the design of the Atoll NIC (see “Atoll Basics” on page 19). The first issue was the limited number of hostports and therefore the limited number processes that were able to communicate via the Atoll simultaneously. The other problem of the Atoll design was the send/receive mechanism. In the case of the Atoll all data that is sent/received needs to be copied twice, one copy into/from a special send-/receive-buffer and another one between this buffer and the Atoll NIC. This approach has a fairly big impact on the communication performance especially in the case of big messages.

These drawbacks lead to the start of a new research project called Extoll. Extoll stands for “Extended Atoll”. The goals of Extoll are to use the Atoll architectural ideas and build a new SAN controller without the drawbacks of Atoll and additional features and improvements. The first goal of the Extoll project is to get rid of the limited supported number of hostports. The Myrinet SAN ([MYRICOM]) has a similar architecture like the Atoll SAN and therefore the same problem. If more hostports are requested than available the software starts to virtualize the hostports. This software virtualisation works in such a way that instead of triggering the hostport directly from user space, the processes call the device driver. The device driver then coordinate access of the different processes to the hostport. With this software virtualisation it is possible to support more hostports than physically available, but with the drawback of a big performance penalty for the indirect communication via the device driver. Therefore the Extoll NIC employs a new approach, which supports hardware virtualisation. With this hardware virtualisation it is possible to support a huge amount of processes with a far less and fixed amount of hardware resources but without any additional software penalty.

The other goal of the Extoll project is to improve the problem of the extra copies for the send receive mechanism. This problem is solved by adding support for RDMA operation to the Extoll NIC. This RDMA operations allow the direct transfer of the data into the device without the overhead via a special send/receive buffer. All currently available SANs, with the exception of the Quadrics QsNet ([QUADRICS]), support RDMA operation only between preregistered buffers. This means, before a RDMA operations can be started the source and the destination processes need to register the source/destination buffers on the NIC. Because the registration of memory is an expensive operation this approach can have a big impact on the performance. Like the Quadrics SAN the Extoll SAN will support RDMA operations without pre-registering of buffers that works with the virtual addresses of the user space process.

Besides the improvements of the Atoll SAN the Extoll SAN also introduces some new features. One of the new features is the support of a hardware barrier. Similar to the crossbar the barrier functionality is inside every NIC and works in a distributed fashion. The next new feature that has been introduced is the Ultra port mechanism. The Ultra port is a special communication mechanism that has been optimised for low latency.

This thesis is part of the Extoll project and has the goal to evaluate the software interface of the Extoll NIC and the design and implementation of the complete software stack. The evaluation includes the comparison of the currently existing software interfaces and their drawbacks. Based on the gained information of the drawbacks of the Atoll and the competitors the whole design of the hostport software interface is made.

The whole project follows the hardware-software co-design paradigm. That means that parallel to the hardware design the software is designed and implemented. The hardware-software co-design approach has the advantage that the overall development time is shorter than by a sequential design. Another advantage is that the software design process can have influences on the hardware design and vice versa.

1.1 Outline

- Chapter 2 “Communication Interfaces” gives an overview of the currently available communication interfaces and highlights the advantages and disadvantages of each presented communication interface.
- Chapter 3 “Extoll Hostport Interface” introduce the Extoll features that will be offered by the final NIC. The rest of this chapter will present the design space for possible software interfaces including all design decisions made.
- Chapter 4 “Design of the ESS” present the Extoll Software Stack design based on the design decisions of the hostport interface.
- Chapter 5 “Implementation of the ESS” will present detailed information about the current implementation.

1.2 Conventions

1.2.1 Definitions

For this thesis the following conventions and definitions are made:

Definition 1-1 : A **process** is a running instance of a program including all variables and states.

Definition 1-2 : A **thread** consists of a instruction counter, a register set and stack. A thread always runs inside a process and therefore has access to the whole address space of the process.

Definition 1-3 : A **virtual connection** is a logical connection between 2 processes that is described by the communication between these 2 processes.

Definition 1-4 : **Polling** means busy polling. When the user busy polls the CPU is non-stop reading and comparing from a certain address to check if a special condition has been reached.

Definition 1-5 : **Waiting** means that the process will not do a busy polling. If the required condition is not reached the whole process will go to sleep. When the required condition has been reached the process will be waked up again.

1.2.2 Decision Tree

Decision trees are used to visualize different aspects and options of a topic . A decision tree consists of the following objects (see Figure 1-1):

- **Topic**. A topic is the root of every decision tree. A topic describe the topic/problem that is presented by the corresponding decision tree.
- **Aspect**. Aspects are connected with orthogonal lines. Aspects are either child of the topic or of other aspects. All Aspects of the same parent describe a different aspect of the common parent and are therefore independent from each other.
- **Option**. Option are always connected with the corresponding aspect by straight lines. All options of an aspect represent alternative approaches for the same aspect. Therefore the options of an aspect are mutual exclusive.

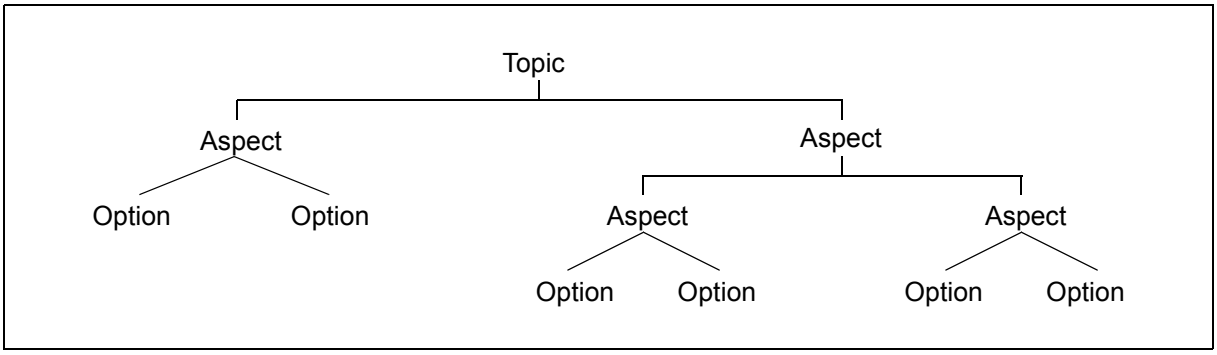


Figure 1-1 : Decision Tree Syntax

1.2.3 UML

This thesis uses UML diagrams for visualisation of different aspects. The UML diagrams that are used in this thesis are valid for the UML version 2.0 standard ([FOWLER], [UML]).

Communication Interfaces

CHAPTER 2

This chapter introduces the most commonly used communication interfaces nowadays. The basics of the corresponding interconnect network will be described as much as necessary to understand the corresponding API. The design of an API must fulfil different goals. The first is the efficiency. To reach the best performance on a high performance hardware the software must not introduce a high performance penalty by an inefficient usage of the available resources. On the other side the software layer should abstract the current hardware and offer an easy to use interface to the upper layer. These interfaces that are presented are the BSD Sockets, Atoll PALMS, Myrinet MX, Quadrics, DAPL and MPI.

2.1 BSD Socket API

The BSD socket API ([STE98]) was introduced in 1983 with the release of the 4.2 BSD system. The goal of the BSD socket API was to create a generic interface for accessing computer networks. Today the BSD socket API is the de-facto standard for network programming. The BSD API was originally designed for IP networks. Today, the BSD socket API supports a broad range of different computer networks and protocols.

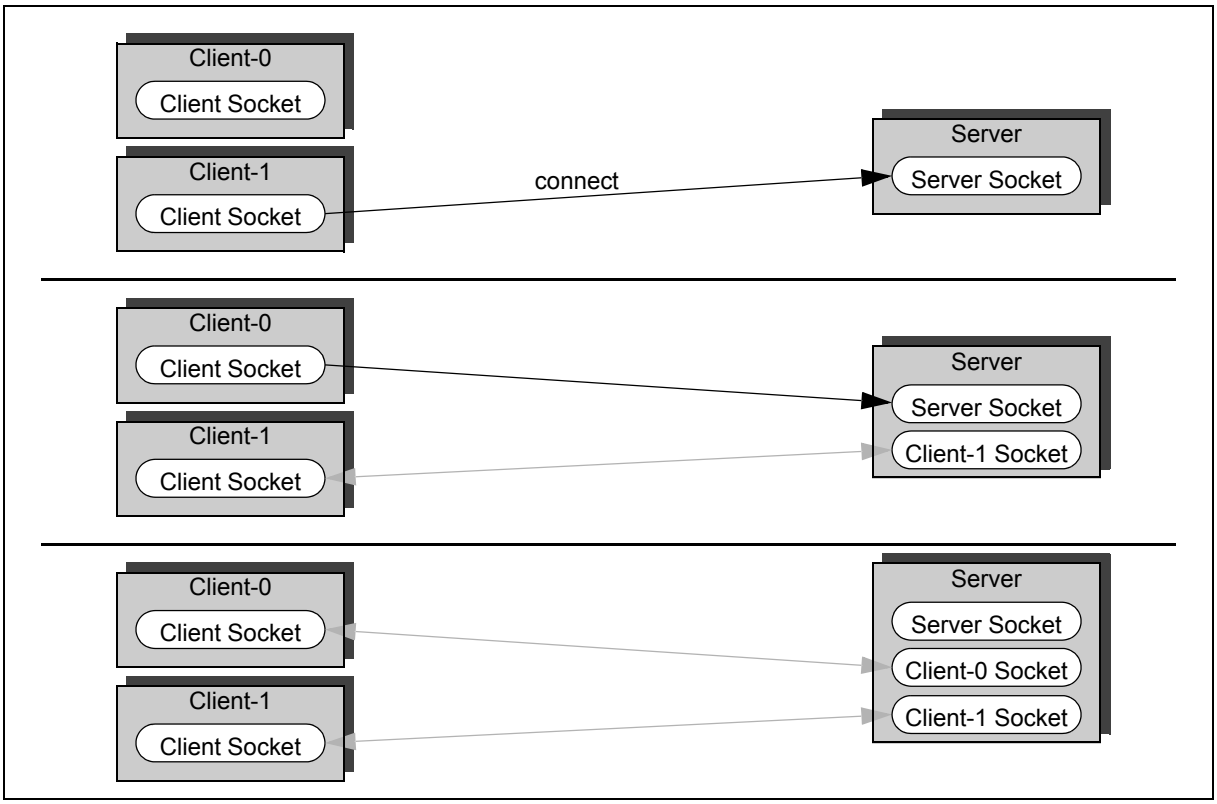


Figure 2-1 : Client/Server Sockets

A socket is the central object of the BSD socket API. A socket is an abstract object that describes a connection between two processes. A socket is a tuple of endpoint information. One endpoint information describes the local endpoint of the communication and the other endpoint information the remote endpoint of the connection. A socket communication is build upon the server/client principle. The server side of the communication creates a new socket. The socket is transformed in to a server socket by binding the socket to a certain address. The server process is now able to wait on the server socket for incoming connections. The client side needs the address information of the server socket to establish a connection. The client side first creates a new socket and then connects to the server side. After the server side accepted the connection, a socket for the corresponding client will be created that represents the server endpoint of the connection. The server side is now able to accept more connections via the server-socket and/or communicate with the client via the returned socket (see Figure 2-1). Figure 2-2 visualize the the server/client principle in more details and shows the involved function calls.

Each socket connection is full duplex, this means that each side is able to read/write data at any time. From the user point of view a socket behaves like an ordinary file descriptor. Therefore the communication can be realised with the normal POSIX ([POSIX]) file operation functions. It is furthermore possible to use a socket with the select, poll and or epoll call to realise a non-busy wait. And as a kind of file descriptor a socket is shared amongst all threads of a process.

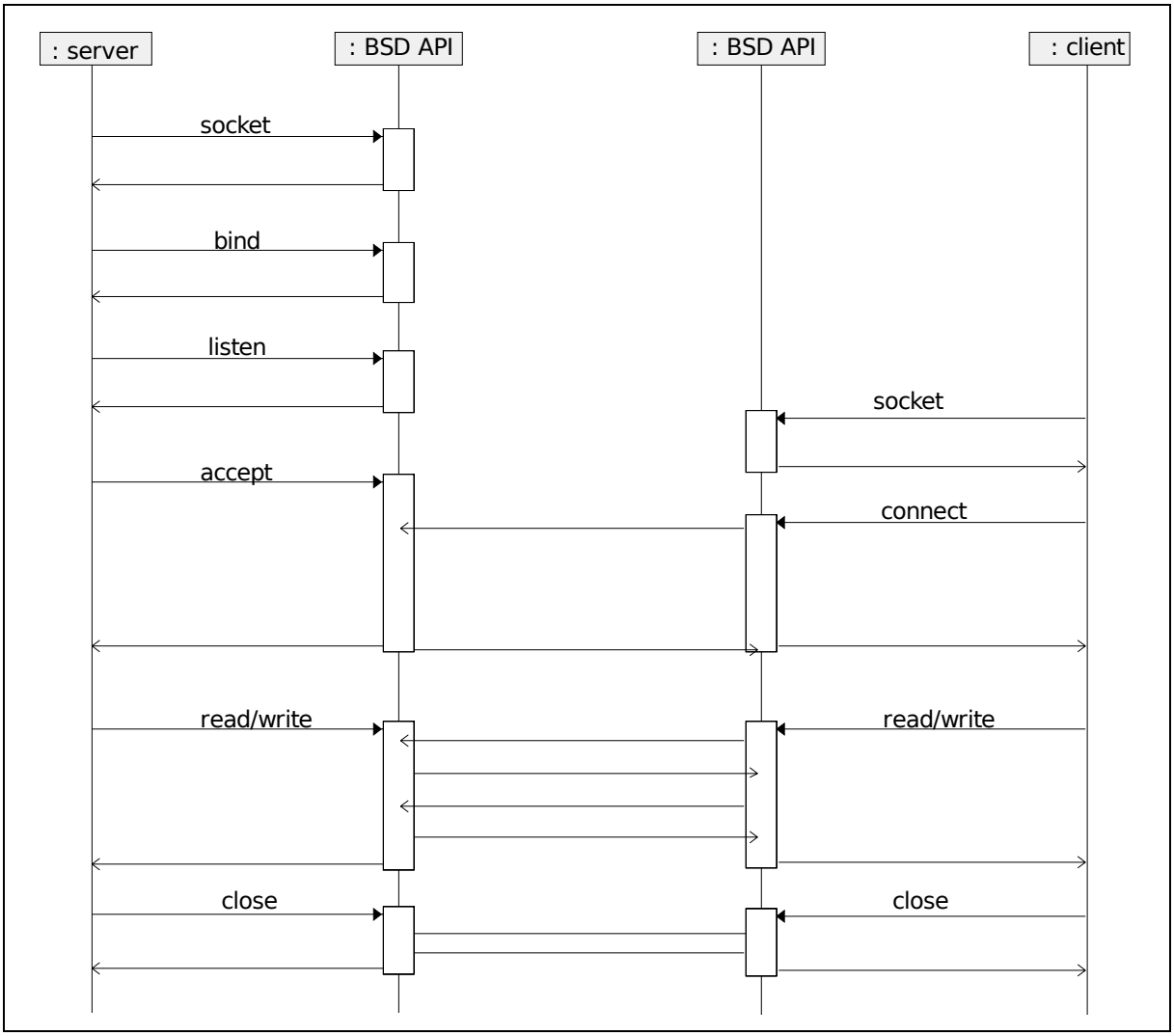


Figure 2-2 : BSD connection setup

2.2 Atoll - PALMS

The Atoll-PALMS is the standard API for the Atoll NIC, which offers a user-level communication mechanism. A description of the basic structure of the Atoll NIC can be found in “Atoll Basics” on page 19. The first versions of the Atoll-PALMS were inspired by the BSD-Interface to allow the users a smooth transition to this new API. During the implementation of a MPICH2 port which was based on the first version of the Atoll-PALMS several drawbacks of this interface showed up. Therefore in the second version of the Atoll-PALMS a much lower-level interface has been offered to the upper layer. This new interface is more complex than the interface of the first version but offers more control to the upper layer which is necessary to realise the maximum performance. For backwards portability the Atoll-PALMS interface of the first version has been emulated on the functions of the second version (see Figure 2-3).

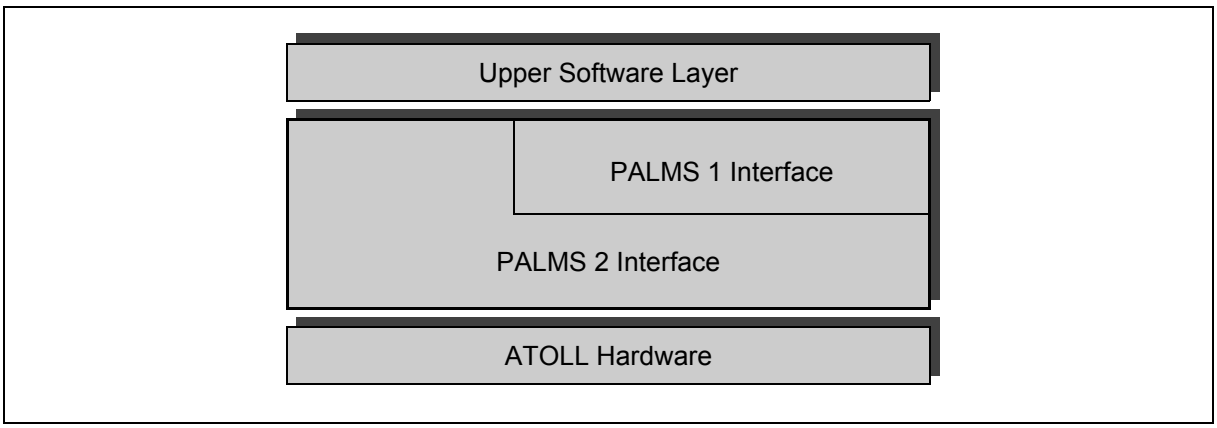


Figure 2-3 : PALMS layout

The Atoll NIC uses a message based communication mechanism. The idea of this system is to exchange messages between Atoll hostports. To identify a hostport in the network each hostport must have an identifier that is unique across the whole network. After a hostport has been opened by a process on the local Atoll NIC, the process is able to send messages to all other opened hostports in the network. To send a message to a certain hostport, the origin hostport process needs the unique hostport identifier of the target hostport. If the target hostport ID is known, the process can create a connection between the local hostport and the target hostport. The Atoll connections are light-weight connections that do not need to exchange information for an establishment of a new connection. This is possible because the Atoll SAN use source-path routing, and creation of a connection mainly consists of a routing table look up. Because the Atoll connections are only unidirectional, both sides need to connect to the corresponding other hostport. The first version of the PALMS-API offered only simple send and receive functions which expect a pointer to a buffer that contains the whole data. These functions automatically performed all necessary tasks to send data including fragmenting of the data. This simple interface had the disadvantage that if several non continuous data buffers need to be send/received, the small data pieces need to be packed into one continuous buffer for the PALMS (see Figure 2-4 (a)). Because of this all data has to be copied twice, first into the temporary buffer for the PALMS and then a second time from the temporary buffer into the send data buffer of the hostport.

To avoid a second copy operation the second version of PALMS introduces direct access to the send/receive buffers. This principle is visualised in Figure 2-4 (b).

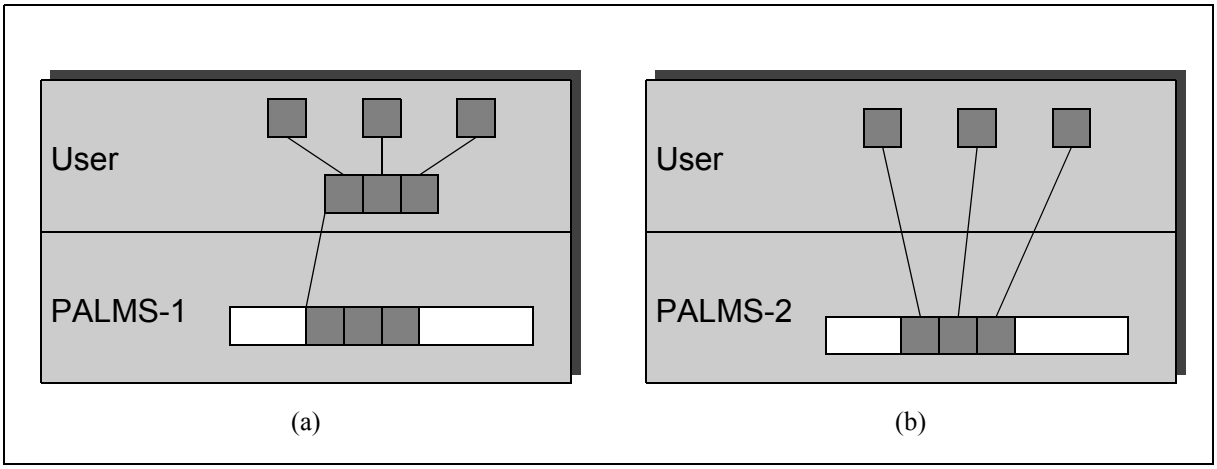


Figure 2-4 : PALMS-1 and PALMS-2 Send/Receive Mechanisms

As described in "Atoll Basics" on page 19 an active notification mechanism is not supported by Atoll. Therefore polling is the only possibility to detect new data. PALMS offers a special function to poll until new data is available. Since this blocking behaviour is discouraged on many situations the PALMS also offers a non-blocking function which checks if a new message is available or not.

2.3 Myrinet - MX

Myrinet is a proprietary cluster interconnect build by Myricom ([MYRICOM]). After Ethernet Myrinet is the most used interconnect for clusters ([TOP500]). Myrinet Express ([MX]) is the most current API for the Myrinet SANs. MX consists of a new API and a firmware. MX allows the user to bypass the operating system and use user-level communication for lower latency.

The central object of the MX API is the endpoint. An endpoint is a virtualization of an NIC. An MX endpoint offers the user a way to use the hardware. An MX endpoint has a similar meaning to MX as a hostport to PALMS. Unlike the Atoll SAN MX supports more than 4 endpoints and an MX endpoint is tagged by an user defined integer value that is used for filtering tasks. The current amount of endpoints that is directly supported by the hardware depends on the current adapter version. If more endpoints are requested than available, then MX will transparently switch from a user-level communication to traditional communication via kernel traps into the device driver. In this traditional communication mode one endpoint in hardware will be shared amongst several processes by multiplexing.

To send data from the local endpoint to a target endpoint the corresponding target address of the remote endpoint must be created. The creation of the remote target address is based on the following information:

- The local endpoint that belongs to the process.
- The remote NIC ID.
- The remote endpoint ID.
- The remote endpoint filter value.

The resulting endpoint address is only valid for this special connection from this local endpoint to the remote endpoint and cannot be used with any other endpoint. The endpoint address represents a unidirectional connection and therefore it can be compared with the Atoll connection. Because the endpoint address is only unidirectional both sides need to create an endpoint address for the corresponding other side to exchange data in both directions.

The whole communication scheme of the MX API is designed to be fully asynchronous. Fully asynchronous means that the initiation of a communication operations is strictly separated by the completion of the operation. For each initiated communication operation an MX request is created and returned to the user. The MX request enables the user to track the status of the initiated operation. MX supports some non-blocking functions to query the request status which allows the user to poll on a request. Besides this non-blocking functions MX also supports functions to wait on a request until a certain state is reached. This wait function will suspend the process until this state is reached.

MX supports message passing communication. For each send operation the user must supply a special matching value that is used to mach the send and receive operation. To receive the message on the remote side a receive needs to be posted with the same matching value. MX guarantees that all send/receive operations will match in-order. It is required to post a receive before the send operation is started. If there is no matching receive MX supports ways to handle this unexpected messages by buffering them until the corresponding receive will be posted.

In the current version 1.0 of MX the support for one-sided communication and collective operations (barrier, broadcast, ...) is missing. The missing functionality will be added in a later version. In addition to the communication via the Myrinet NIC the MX API realizes the communication between processes on the same node via shared memory mechanism. The switch between the NIC and the shared-memory communication is completely transparent from the user point of view. The MX API is available in a thread-safe version and a not thread-safe version. If thread-safety is not required by the upper layer the non-thread-safe library can be used which offers a slightly better latency than the thread-safe version.

2.4 Quadrics - Elan Library

QsNet is a proprietary interconnect solution developed by Quadrics ([QUADRICS]). The core of each QsNet adapter is the “Elan” network processor. To access the functionality offered by a QsNet adapter Quadrics supports two libraries ([ELAN]). Figure 2-5 gives an overview of the quadrics software stack. The Elan3 library offers a low-level interface to a QsNet adapter while the Elan library uses the functionality offered by the Elan3 library to build a high-level interface with the following communication modes:

- Direct memory access.
- Message passing.
- Queue based communication.
- Put/Get based communication.

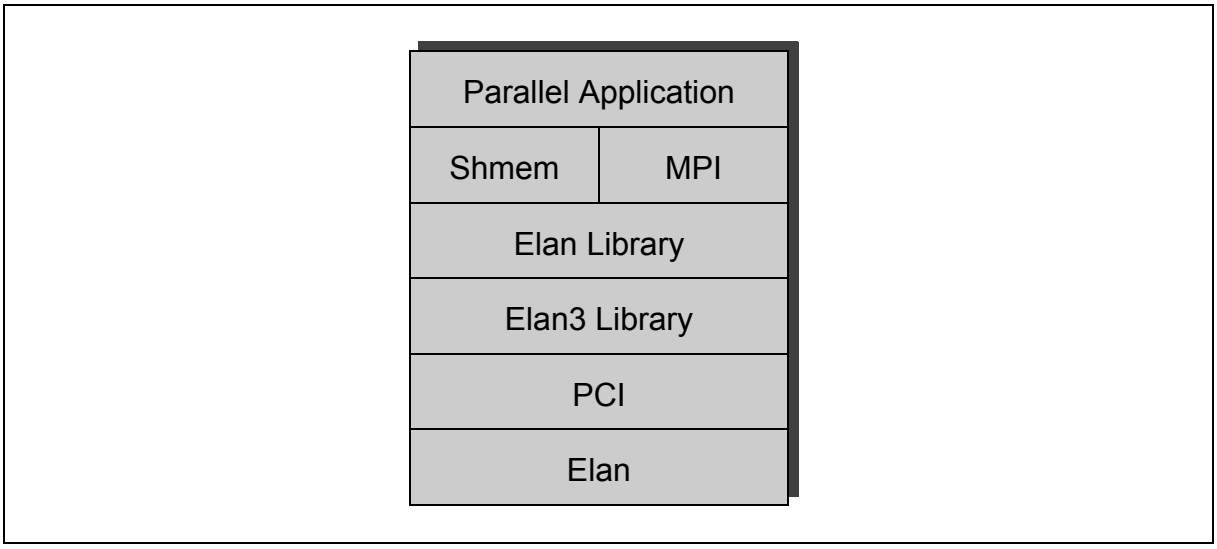


Figure 2-5 : Elan Software Stack

The principle of the QsNet is to build a kind of “network computer” that is inspired by the architecture of a single node computer. If an application is started on a single node this application will be represented by a process that is executed on the CPU. When a process is running on a CPU the whole address space of the process is mapped into the CPU. Data is copied inside the virtual address space of a process, by copying the data from the source buffer into the CPU and then from the CPU to the destination buffer. This principle of a single CPU system is shown in Figure 2-6.

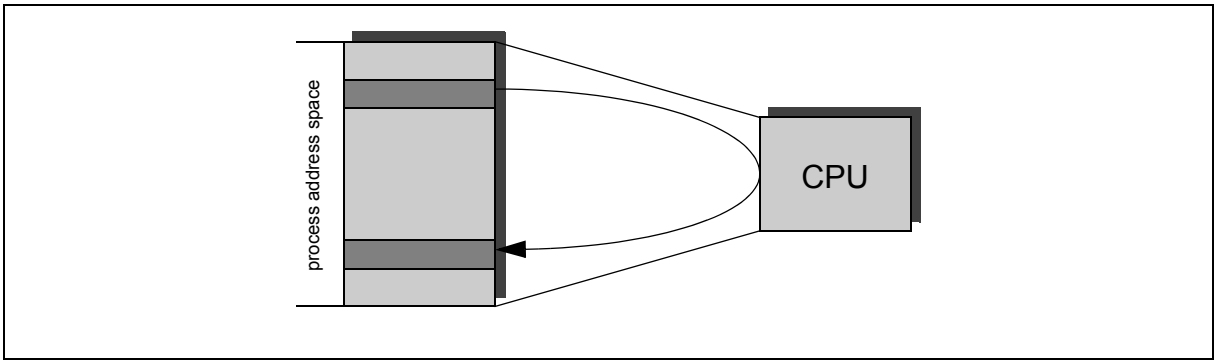


Figure 2-6 : Single Node Principle.

The QsNet approach is to map this concept to a parallel system that consists of several nodes. On a parallel system parallel applications are executed. A parallel program consists of several processes that can be run on any node in the parallel system. The amount of processes of a parallel application is determined at the start and is fixed during the whole execution time. All processes of a parallel application are numbered from 0 .. (N-1) with the so called virtual process identifier (VPID[1]). This VPID is used to identify a process inside a parallel application. A global address space of the parallel application is formed by arranging the address spaces of the processes in a sequential order of their VPIDs. The addresses of the global address space are build by prefixing the virtual address of each virtual address space with the corresponding VPID (#VPID#VADDR). The global address space is mapped into the quadrics network processor.

1. Note that the Quadrics VPID != Extoll VPID.

With the Quadrics network processor it is now possible to copy data inside this global address space from one location to another one, especially from the address space of one process to another one. An example of a parallel application with two processes is shown in Figure 2-7.

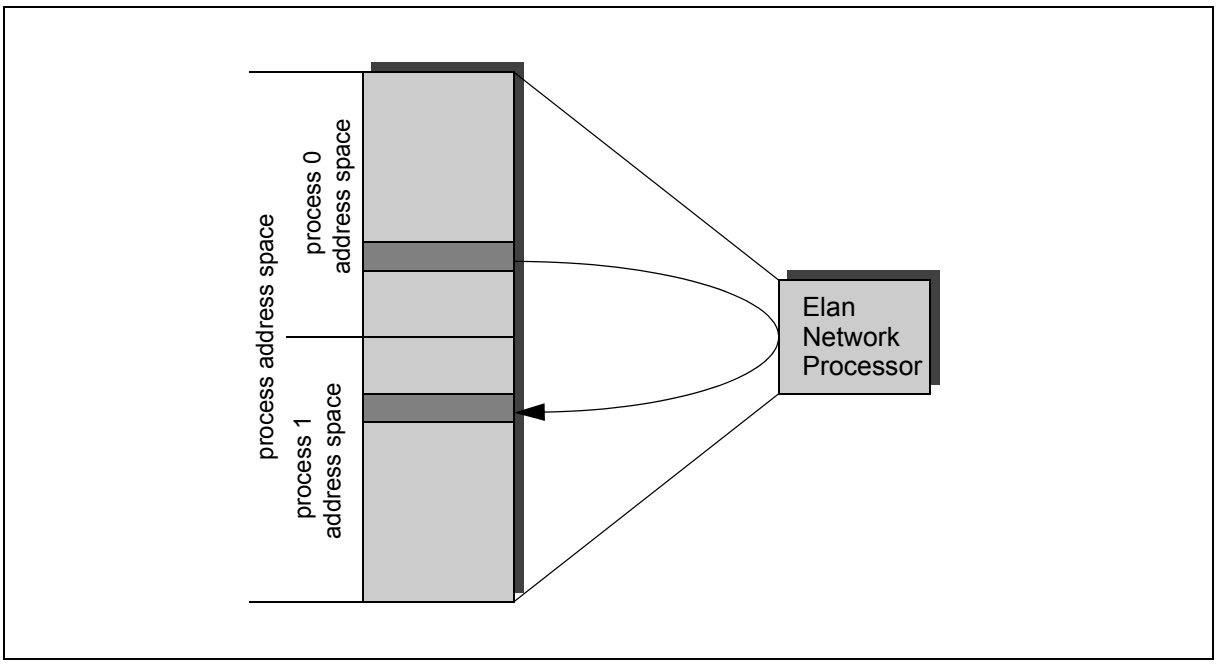


Figure 2-7 : Quadrics Network Processor

This is the basic concept of the QsNet. The Elan3 library offers an abstraction layer to this functionality that hides the differences of the different QsNet adapters. The library mainly offers functions to initiate the memory copy operation. The memory operations can be associated with two events that will be set when the operation is completed and two cookie values. One event/cookie is for the local side and the other one for the remote side. A process can either poll on an event or wait on an event which will cause process to be suspended until the event is set.

The Elan library works on top of the Elan3 library and offers more communication modes. The message passing interface of the Elan library allows a process to send a message to any other process of the parallel program. Each message can be tagged with a user specified value. The receiver can select by source VPID and/or specified tag which message will be received. Besides the point-to-point operations the Elan library supports some collective operations. The put/get communication mechanism consists of put and get functions that realise write and read operations from the local memory to the remote memory.

All communication operations are able to use the virtual address of the process. Therefore buffers do not need to be registered by the hardware before they can be used for communication.

2.5 InfiniBand - Verbs

The InfiniBand Architecture (IBA) is an industry-standard architecture for server I/O and inter-server communication. The IBA was developed by the InfiniBand Trade Association ([IBTA]) as a replacement for the current I/O bus architectures. The InfiniBand architecture specifies the whole system from the link up to the software interface to control an InfiniBand adapter

([PFISTER]). The software interface is called Verbs ([VERBS]) and consists of function prototypes and the corresponding semantic. The exact implementation of the Verbs is not specified and vendor specific. The InfiniBand architecture defines the following services:

- Reliable Connections (RC)
- Unreliable Datagram (UD)
- Unreliable Connection (UC) [optional]
- Reliable Datagram (RD) [optional]
- Raw IPv6 Datagram & Raw Ethertype Datagram [optional]

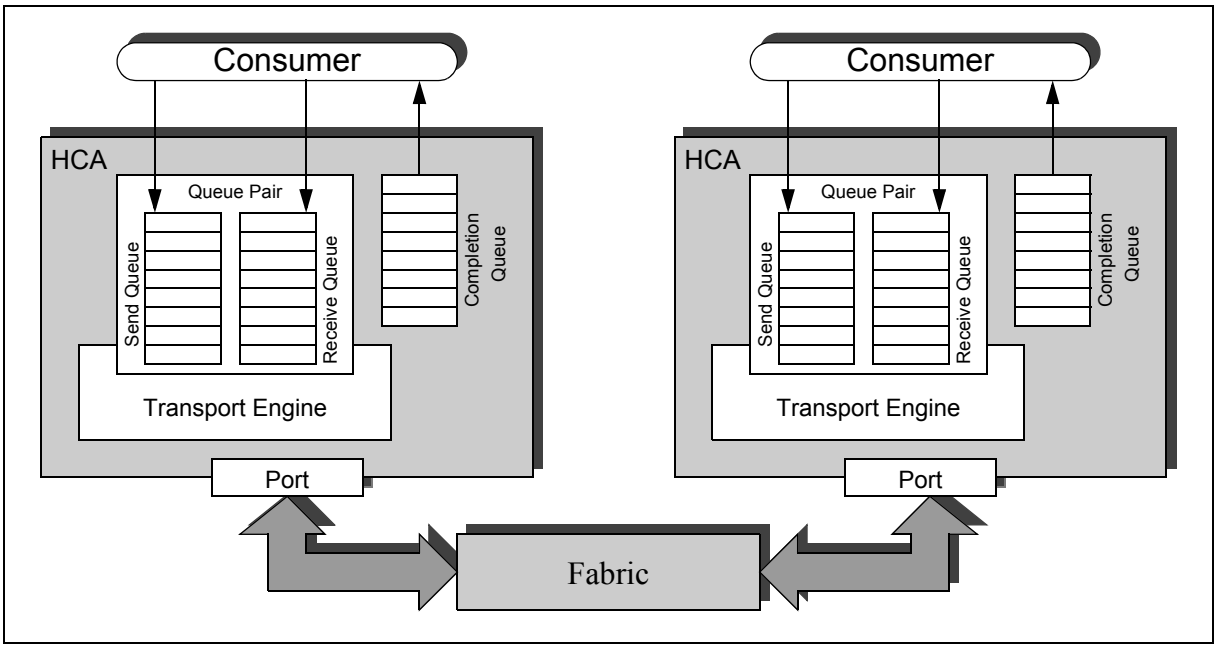


Figure 2-8 : InfiniBand HCA

The central part of each InfiniBand communication is the so called queue pair (QP). The queue consist of two queues, one send and one receive queue (see Figure 2-8). If the user wants to start an operation the user posts a descriptor that describes the operation in the corresponding send or receive queue. If the operation is an operation that actively transmits data the descriptor will be fetched by the transport engine and be processed. The data will be transported via the InfiniBand fabric to the destination host channel adapter (HCA). On the destination side the data will be processed. If the data needs to be received the next descriptor from the receive queue will be fetched to determine the destination of the data. On both sides a completion will be generated after the operation is completed. The created notification will be stored in a completion queue. If a connection service type is used then a connection is always exclusively established between two QPs. A QP can only be member of one connection which leads to the problem that for several connection several QPs per process needs to be allocated (see Figure 2-9 (a)). Only one QP is necessary to communicate with several other processes when the datagram service is used (see Figure 2-9 (b)).

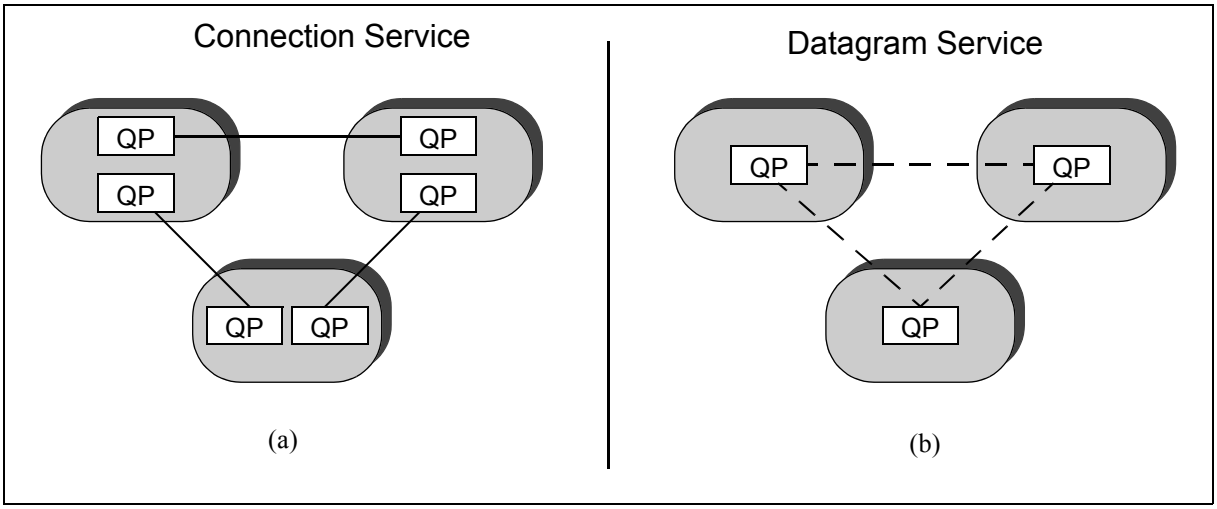


Figure 2-9 : QP connection and datagram service

With the Verbs specification InfiniBand defines a standard set of functions to implement the previous explained operations. The Verbs offer functions to allocate and manger the introduced queues. InfiniBand supports message passing communication, via posted sends and receive, and remote memory operations. All communication operations are asynchronous. After the communication operation has been initiated the user will receive a completion when the operation has completed. It is possible to either poll for new completions or use a non polling wait function that suspends the process until a new completion is available. Additionally, the user can register a callback function that is called when a new completion in the completion queue is created. Every memory that is used by a communication operation must be registered with the HCA. A registered buffer can be further sub-divided into memory windows that have different access rights (see Figure 2-10). While the registration of a buffer can be a slow operation that involves a kernel trap, the creation of a memory window is completely done in user space by posting a corresponding operation to the QPs.

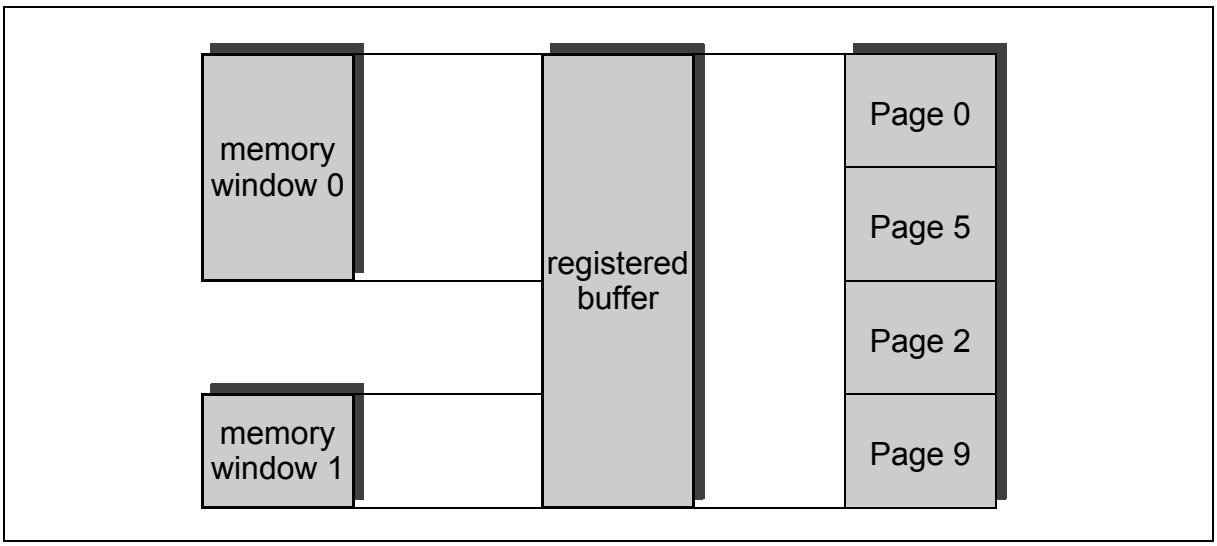


Figure 2-10 : Memory Window and Registered Buffers

2.6 DAPL

The Direct Access Application Library (DAPL) is an API that is suitable for all RDMA-capable interconnect networks. DAPL is a generic API interface specification that abstracts from the different interconnect networks and operating systems. The DAPL specification consists of a user-level (uDapl) and a kernel-level (kDapl) API specification. The whole specification of the DAPL interface is maintained and coordinated by the DAT-Consortium ([DAT]). The concept of DAPL is that all applications that have been written for DAPL are able to communicate via different interconnected networks without re-building. This is realized by selecting the used provider at the start of each DAPL program. A provider is a module that is dynamically loaded and offers the functionality for a certain communication system to the DAPL interface (see Figure 2-11).

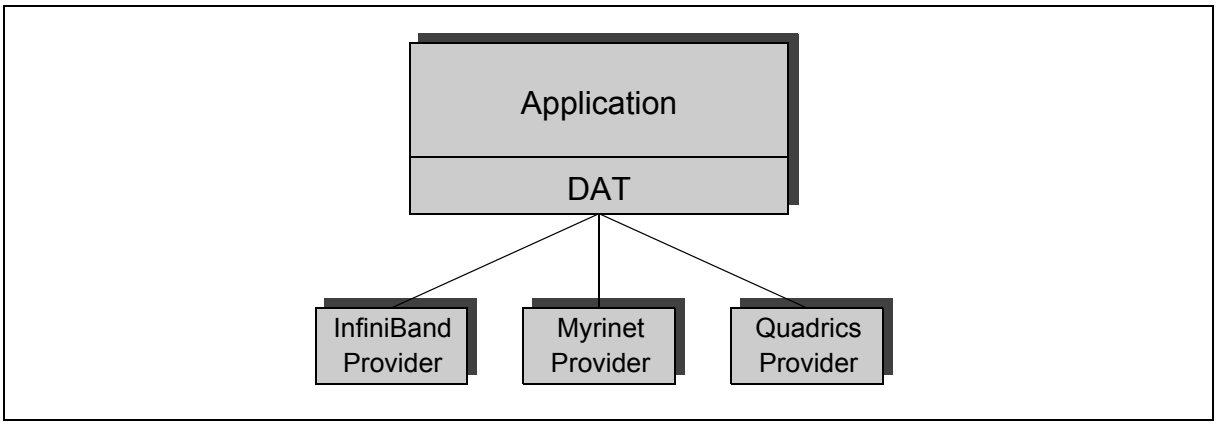


Figure 2-11 : DAT System

The connection setup in DAPL is similar to the BSD socket server/client approach. The server side creates a service point. A service point is like an open port that allows clients to create a connection. The DAPL endpoints are similar to BSD sockets and represent a connection to one remote node. The connection setup works similar to the BSD socket approach. The active side creates an EP and creates a connection to the remote service point. After the server side accepted the connection a corresponding EP for the connection will be returned. DAPL offers all common types of data transfer operations (DTO). This DTOs are posted send, posted receive, RDMA read and write. The DAPL communication is asynchronous. Like in InfiniBand the user must post a receive operation to receive data transferred by a send operation. The user receives all kinds of information (DTO completion, connection request, disconnection, ...) via events. The central objects of the event system are the event dispatchers (ED). An event dispatcher looks like an event queue that collects events from different event streams (see Figure 2-12).

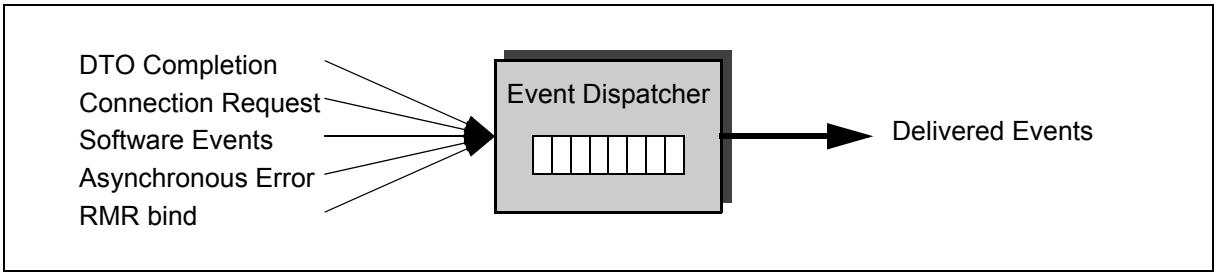


Figure 2-12 : Event Dispatcher

DAPL requires that all memory buffers have been registered before they are used with a DTO operation. Like InfiniBand DAPL supports the creation of memory windows on a registered buffer (see Figure 2-10).

2.7 MPI

At the beginning of parallel systems each system had his own API for using the parallel resources. This lead to the problem that a program that had been developed on a certain parallel system was not likely to run on any other system. Therefore independent software providers had to write a version of their product for each system they wanted to support. To solve this problem several companies and research organisations formed the MPI-Forum ([MPIFORUM]). The goal of this forum was to define an unique API for writing parallel programs. This API was called MPI which stands for "Message Passing Interface". The idea of this approach was to have a single API and a program that has been written with this interface would be able to run on different parallel systems without modification. The only thing that needs to be adapted to the different systems is the MPI implementation. Today MPI is the de facto standard for writing parallel programs with existing implementations for all major systems. For different reasons the first version of the MPI standards covered only message passing. The second and most current version the MPI standard extended the first version by adding support for parallel I/O, dynamic process management and remote memory access (see Figure 2-13).

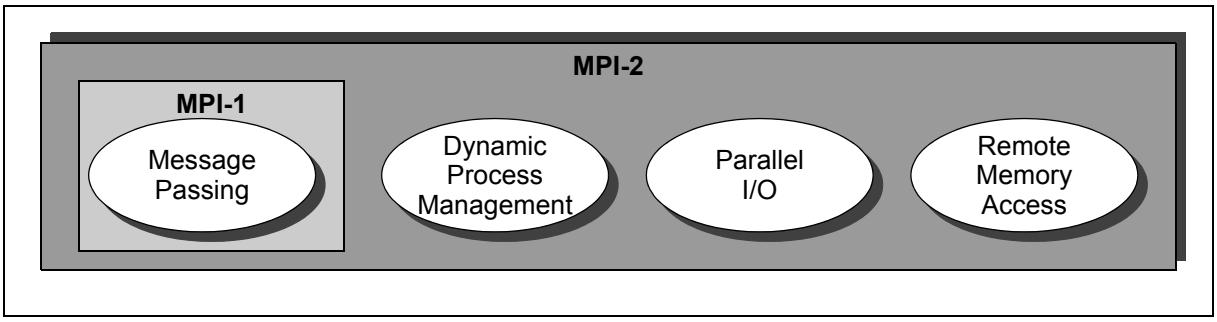


Figure 2-13 : MPI Overview

The basic object of MPI is a so called communicator. A communicator is an abstract object that allows the user to exchange messages between the processes that are associated with this communicator. Every communicator contains between one and two progress groups. Communicators with only one process group are so called Intra-Communicators. An Intra-Communicator can only be used to exchange data between processes of a single process group. A communicator with 2 process groups is called Inter-Communicator and can be used to exchange messages between 2 different process groups. This 2 different kind of communicators are visualised in Figure 2-14. Each process in a process group is identified by a unique number, the so called "rank". The first process of a group always gets the rank 0, the second rank 1, and so on. The order of the processes is not defined by MPI and depends on the implementation. This means that the same processes that is member of 2 different process groups can have a different rank in each group and that the same rank on different groups can belong to 2 different processes.

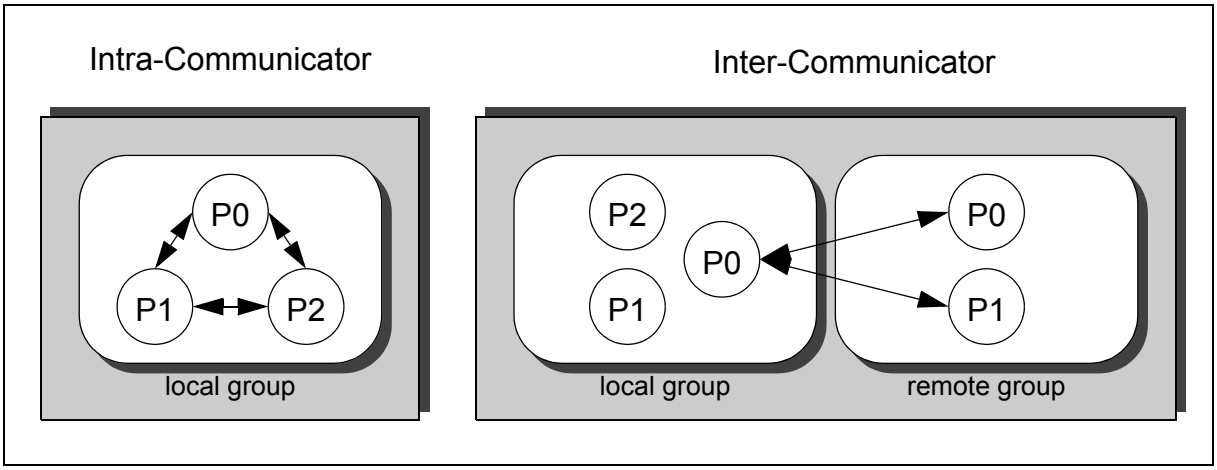


Figure 2-14 : MPI Communicators

The point-to-point communication between 2 processes is realised via a send on the sender side and a matching receive on the receiver side. The matching of a send and a receive operation is based on the used communicator, the source process rank and the specified tag. The MPI standard requires that send operations with the same matching information must be delivered in-order. The MPI standard specifies blocking and non-blocking send/receive operations. The semantic of MPI defines that when a blocking send returns the user is able to use the data buffer again without any danger. This does not mean that the corresponding send already finished or that the remote side already received the data. If non-blocking operations are used, the user gets a request for the corresponding operations that allows the user to keep track of the status of the operations and/or wait until the operations are finished. Additionally to the point-to-point operations the MPI standard specifies several collective operations. The collective operations can be sub-divided in scatter-, gather- and reduce-operations. A special collective function is the barrier because this function transfers no data and is only used for synchronisation.

When an MPI program is started all processes are grouped in one big process group that is associated with the global communicator *MPI_COMM_WORLD*. Because this static approach is limiting the Dynamic Process Management (DPM) has been integrated. The DPM consists mainly of one function to start a certain number of processes in a new process-group. This function returns an Inter-Communicator that allows communication between the process groups.

To avoid a bottleneck in the I/O performance and to increase portability the MPI standard defines a set of file operations. The MPI file operations were inspired by the POSIX file operations. While the opening and closing of a MPI file is a collective operation among the processes of to the used communicator, the file may be accessed by each process separately. Several file pointers per MPI file are managed, one global (shared) file pointer that is valid for all processes and one local file pointer per process.

The last part of the MPI standard covers Remote Memory Access (RMA). RMA allows a process to directly access and modify the memory of another process. MPI defines 2 different modes of RMA operations, the passive target and the active target mode. In the active target mode the target process is actively participating in the execution/completion of an RMA operation. In contrast to the active target mode the passive target mode no involvement of the target process is required. The first thing that needs to be done to share data via RMA is to create a MPI memory window. A MPI memory window is a description of an area of memory that is accessible by other processes. The creation of a MPI memory window is a collective operation that must be done amongst all processes of the used communicator. When the MPI window has been suc-

cessfully created each processes of the communicator is able to access the corresponding windows of all other processes in the communicator. For the real modification of the remote memory the MPI standard defines a get, put and accumulate function. The get function is used to read data from a remote memory into a local buffer while the put operation writes data from a local buffer to a remote memory window. The accumulate function applies local data with a function to a remote memory window. The functions for accumulation are all logical and arithmetical base functions (e.g. AND, ADD, OR).

2.8 Conclusions

As seen in the previous sections all communication systems (except the BSD sockets and the PALMS) support the following features to achieve the maximum possible performance:

- **Asynchronous Communication**. Asynchronous communication means that the user triggers the communication operations and will get notified when the triggered operation has ben completed. This approach allows to overlap the communication and computation and therefore leads to the maximum exploration of parallelism.
- **RDMA Capabilities**. RDMA capabilities allow the user to directly transfer to/from remote memory of another process. With this feature it is possible to realize true zero copy operations.
- **Event Driven**. As seen it is important to have an active notification system for the user which allows a non polling waiting operation. With this event driven approach it is possible to save CPU time and therefore increase the performance.

Extoll Hostport Interface

CHAPTER 3

In this chapter a design space analysis of the hostport interface is presented. First a short overview of the Atoll design is given. After the historical overview of Atoll a short description of the Extoll basic concepts is presented. Then each aspect of the Extoll hostport interface will be analysed.

3.1 Atoll Basics

3.1.1 Atoll Architecture

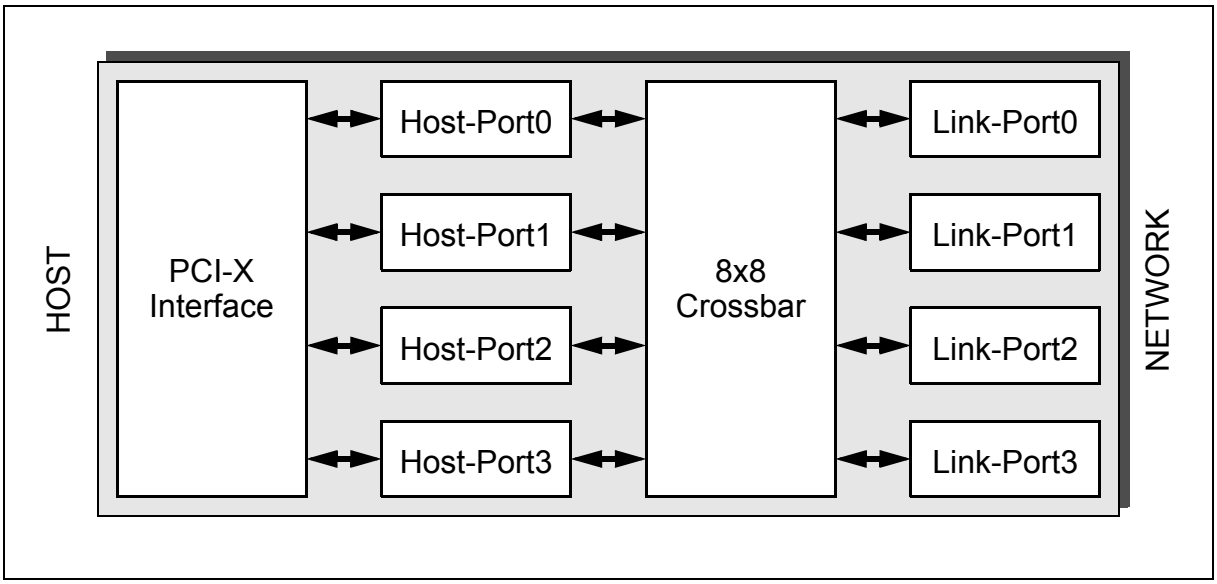


Figure 3-1 : Atoll architecture

After the PCI-X interface that transforms the PCI(X) interface into an internal interface, the Atoll has 4 hostports. Each hostport is represented by a hardware structure. Each hostport can be seen as a communication unit. In the Atoll network messages can be send from one hostport to another one (including to itself). To get the maximum performance, especially low latency, the Atoll uses user-level communication. In the case of user-level communication the user process is able to trigger operations directly from the user space without the need of trapping into the operating system. Because multiple access from several different processes at the same time would lead to problems a hostport is exclusively associated with one process. The Atoll offers 4 hostports which limits the maximum number of processes per node to 4*(number of Atoll NICs).

During the design phase of Atoll the first dual processor systems were available and 4 processor systems were expected in the near future. This was one of the reasons for the limitation to 4 hostports. Since the start of the Atoll project the technology made huge progress. Today it is possible to build nodes with 8 CPUs and more. After the introduction of hyper-threading (HT) real multi-core CPUs were introduced. If a user puts 8 dual-core CPUs into one node he gets a 16 CPU node. The user could run 16 processes in parallel, but is limited to run a maximum of 4 processes because of the limitations of the Atoll NIC, this is unacceptable.

Another problem of the Atoll NIC, and many other SANs, is that the data needs to be copied into/from special send/receive buffers. These extra copies limit the performance dramatically. For this reason many SAN controllers were extended to support Remote Direct Memory Access operations (RDMA). RDMA operations enable a device to read/write data directly from the process memory address space without the need to make a copy in the send/receive buffers. With this feature it is possible to realize a zero copy protocol.

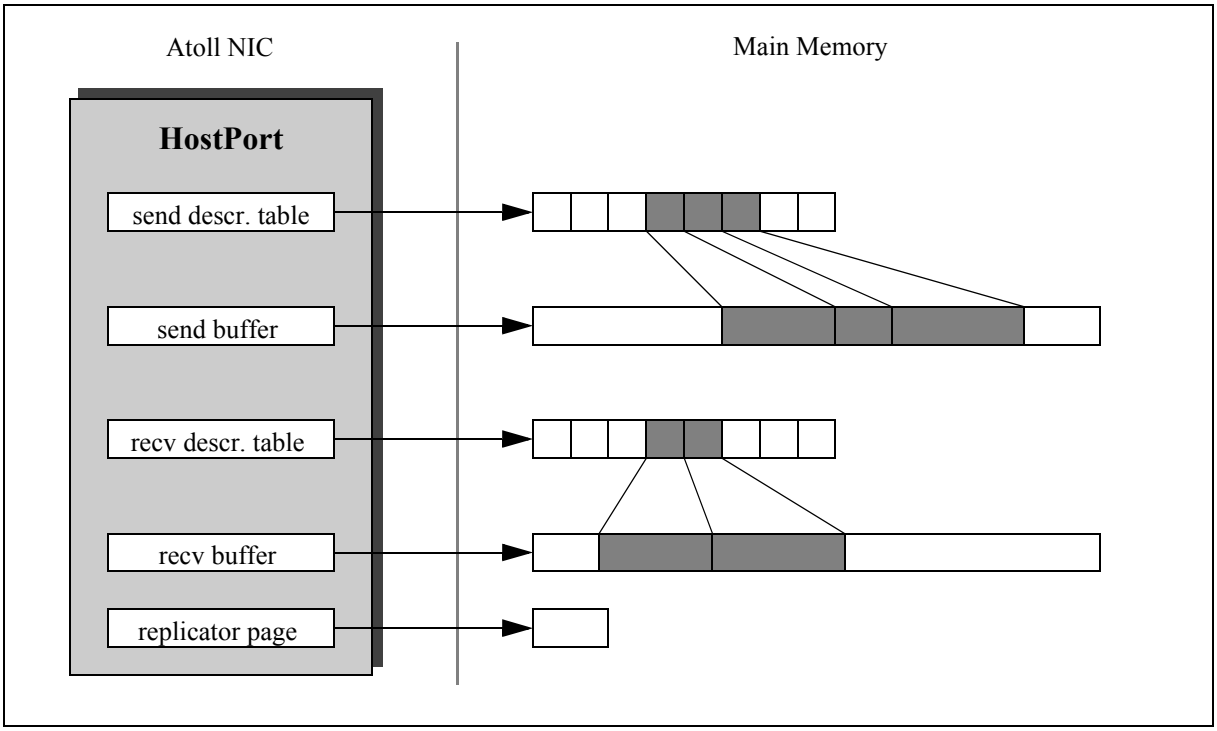


Figure 3-2 : Atoll Hostport

As shown in Figure 3-1 the Atoll NIC supports up to 4 hostports. All hostport have their own fixed hardware resources. Each hostport has a send/receive data buffer and a corresponding descriptor table. The send/receive data buffers contain the data of the messages that should be send or have been received. The send/receive descriptor tables contain the descriptors, one for each message in the corresponding buffer. All data buffers and descriptor tables are managed as ring-buffers with wrap-around semantic. To avoid the expensive hardware accesses each hostport mirrors its status information into a page in the main memory, the so called ‘replicator page’. In Figure 3-2 the relationships between the data buffers and descriptor tables are visualised. The ring-buffers are managed via read- and write-pointers. The state of an empty and a full ring-buffer is implicitly represented by the read-/write-pointer. This avoids an extra variable for an empty and full state. That approach not only reduces the number of updates it also avoids race

conditions. Because Atoll does not support a notification mechanism for newly arrived messages each process has to poll on the replicator page to check and wait for new messages. This behaviour leads to very small latency but also to a very high CPU utilisation.

3.1.2 Atoll Send

A message is sent by executing the following steps:

- Copy the whole data into send data buffer.
- Update the write pointer of the send data buffer.
- Create a new descriptor in the send descriptor table.
- Update the write pointer of the send descriptor tables. This update operation triggers the Atoll NIC to fetch the next descriptor from the main memory.
- Atoll fetches all the data from the memory and sends it to the destination.
- Atoll updates the read pointer of the send data buffer.
- Atoll updates the read pointer of the send descriptor table.

3.1.3 Atoll Receive

A message is received by the following steps:

- The Atoll NIC copies the message data into the receive data buffer.
- The Atoll NIC creates a message descriptor for the received message in the receive descriptor table.
- The Atoll NIC updates the receive data write pointer.
- The Atoll NIC updates the receive descriptor write pointer.
- The process detects that there is a new receive descriptor in the receive descriptor table. The process use the receive descriptor to consume the message from the receive data buffer.
- The process update the read pointer of the receive data buffer.
- The process update the read pointer of the receive descriptor table.

3.1.4 Summary

The Atoll SAN has been proven to work and to be efficient. During the whole development process the following main disadvantages have been figured out:

- The Atoll only supports 4 hostports.
- The Atoll only supports polling.
- The Atoll only supports message based data transfers (extra copies).

For more information about the Atoll design refer to the dissertation of Larz Rzymianowicz ([RZY97]).

3.2 Extoll Overview

This section gives a short overview of the Extoll basics. For further details refer to the diploma thesis of Dirk Franger ([FRANGER04]).

.

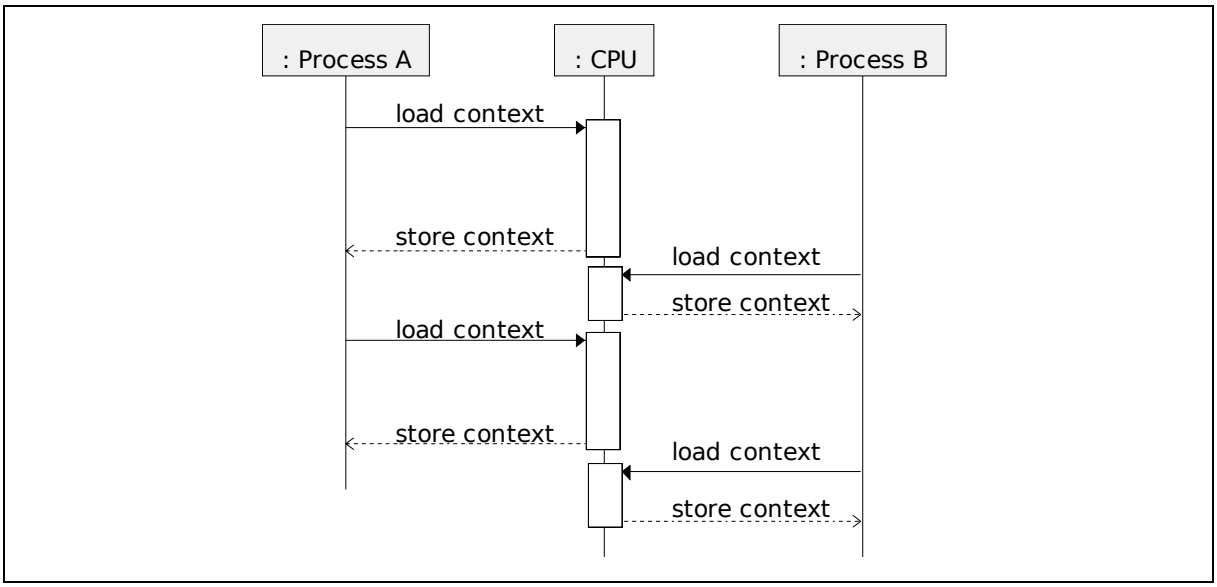


Figure 3-3 : Process Scheduling

A first approach to overcome the limited number of hostports could be to increase the number of hostports that are supported by the hardware. This approach has several disadvantages. The first disadvantage is that this approach does not solve the problem that the hardware limits the available number of hostports, it only increases the threshold when the problem occurs. Another disadvantage is that if the number of directly in hardware supported hostports is increased, this also increases the area of the die, the power consumption and so on. Because of these disadvantages it is necessary to support a large number of hostports without a direct representation in hardware.

This kind of problem also occurs on a computer system where an unlimited number of processes runs simultaneously on a fixed amount of CPUs. On a computer system this problem has been solved by virtualisation of the resource CPU. Instead of associating a process with a certain CPU a process is running on a virtual CPU. A virtual CPU consists of a set of registers that represent the current context of a process. Several processes can now be executed by time multiplexing the CPU(s) between the processes. The switching between two processes is done by storing the register set (context) of the actual running process into the main memory. Then the register set (context) of the next process is loaded into the CPU and executed (see Figure 3-3).

This approach is adopted by the Extoll to form a communication processor. In this communication processor a hostport will not be associated with a specific hardware hostport. A hostport is now virtualized and is therefore called virtual hostport (VP). The current state of a VP is described by a VP descriptor (context). This VP context is loaded into the Extoll when the hostport needs to perform an action. On a computer system the operating system is responsible for the decision which process is running on the CPU. This behaviour is highly discouraged in the Extoll design because the synchronisation via the operating system would increase the latency too much. Because the Extoll is supposed to communicate via user-level communication each VP that needs to perform an action must be able to cause a switch to his context from user space. A

general problem is that a switch to the current process can only be done when there is not already another VP in execution. In the case that there is another VP running the request of the current VP must be rejected and the VP must perform the request again. To perform a switch the Extoll requires the VP identifier (VPID) of the VP that wants to perform an operation. A solution could be to map I/O space from the Extoll NIC into the address space of every process that opened a VP. Each process that wants to trigger an operation via a VP write the corresponding VPID at the beginning of the I/O space (see Figure 3-4 (a)). This approach has the disadvantage that the VP that triggers the Extoll via this I/O space does notdoes not have the possibility to detect if the Extoll accepted the request, because write operations have no feed back.

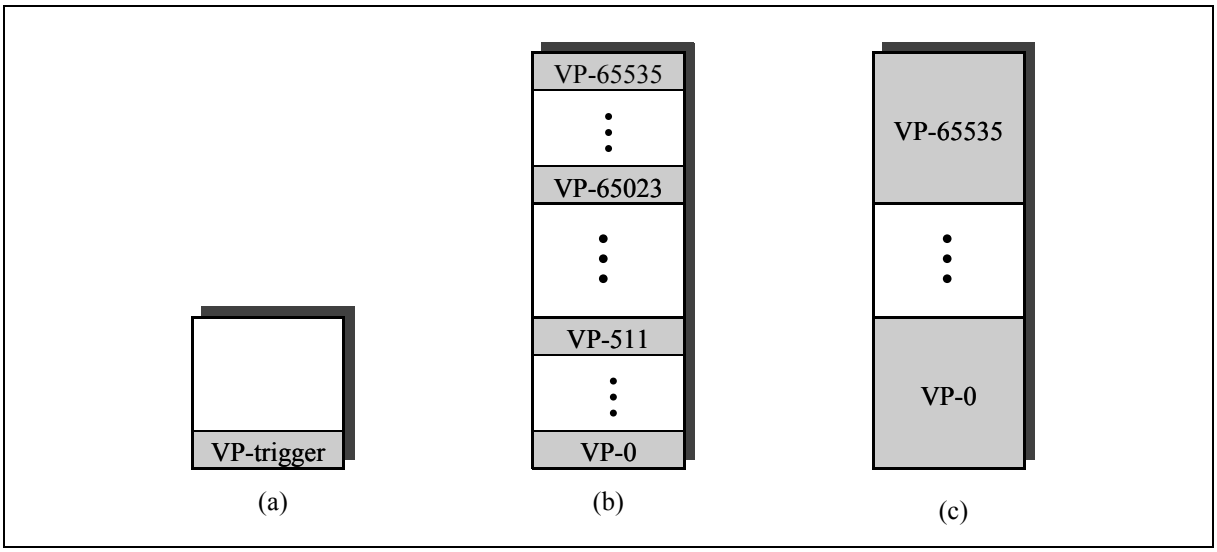


Figure 3-4 : Trigger Page

The problem with the missing feedback can be solved by replacing the write operation with a read operation. The feedback information can be returned in the value that is read from the device. If all VPs would read from the same location in the I/O space the Extoll could not differ between the different VPs. Therefore each VP reads from a different place in the I/O space. The offset of the read operation determines the VPID (see Figure 3-4 (b)). This approach works but has the disadvantage that there is a security issue. Because all processes have access to the same I/O space it is possible that one process reads from the offset of another process. This issue is solved by increasing the trigger area of each VP to the size of one memory page the so called trigger page. Every process maps only the trigger pages of the VPs that have been opened (see Figure 3-4 (c) and Figure 3-5).

The trigger page solution solves all of these problems. As mentioned before every process of a VP that has to perform an operation triggers Extoll via the trigger page. This should cause the Extoll to switch to the current VP. But if Extoll is already executing another VP this switch cannot be performed. In this case the process has to poll on the trigger page until the Extoll NIC has accepted the request to switch to the current VP. On a busy system with much communication this case will be the standard. To overcome this situation the Extoll has a buffer for incoming request of the trigger pages. If the request for a VP has been successfully inserted into the buffer the Extoll guaranties that the corresponding VP will be executed in the future. This buffer is called conditional store buffer ([CSB])(see Figure 3-5).

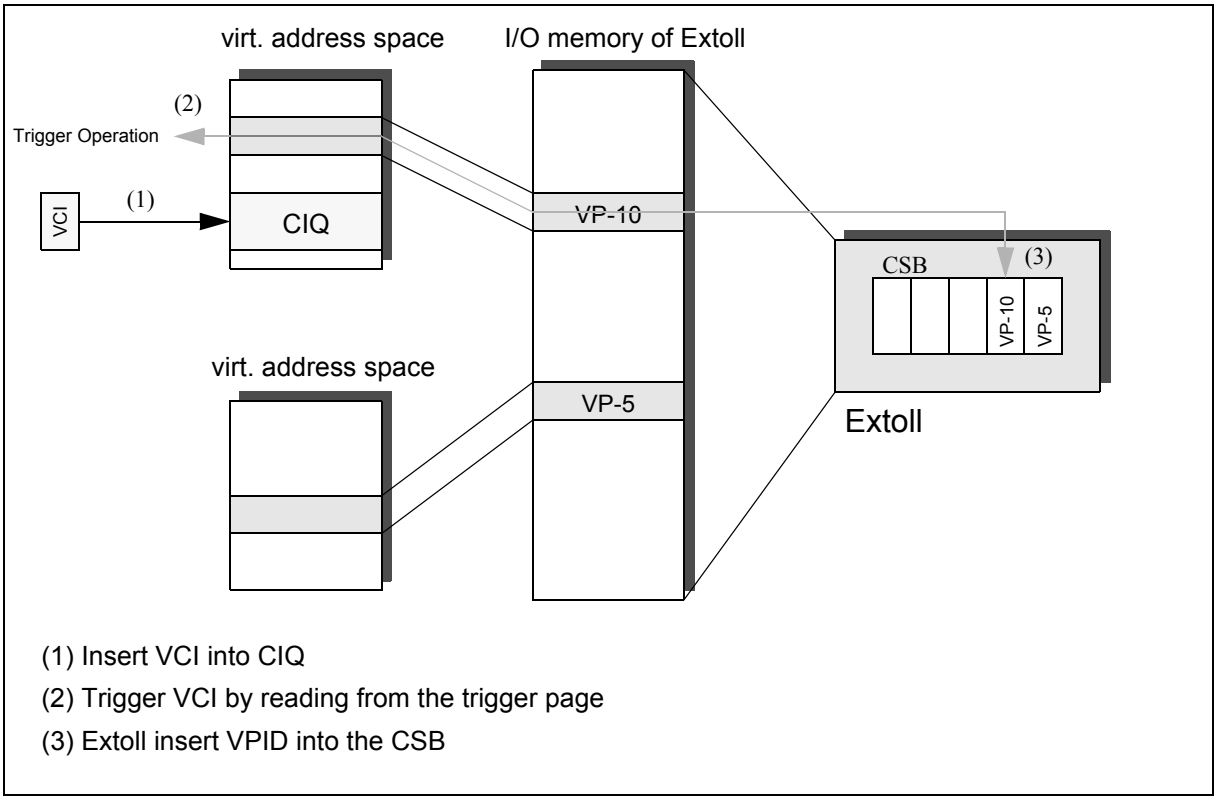


Figure 3-5 : Conditional Store Buffer

A missing feature of the Atoll design was the lack of zero-copy functionality. Therefore Extoll supports RDMA operations. Extoll will support Get and Put operations for reading / writing of the memory of a remote process. The Extoll NIC does not require to pre-register a buffer on the hardware but requires that the buffers that are used for RDMA operations are located inside a memory window. Like the QsNet adapters from Quadrics the Extoll NIC will be able to directly handle virtual addresses. Unlike QsNet, Extoll has no table-walk engine to translate the virtual addresses to physical addresses which are needed by the DMA engine. Therefore Extoll will only have a translation-lookaside buffer and the real address translation is done in software.

Unlike the Atoll NIC, the Extoll NIC will support notifications for the users. Notifications are supported for the completion of previously triggered operations and for the arrival of new messages. The operations that can be executed are called virtual communication instructions (VCI). The VCIs are stored in a physically continuous buffer in memory that is managed as a ring buffer. This buffer is called communication instruction queue (CIQ).

To execute a new operation a VP must create a VCI that describes the operation. This VCI is inserted into the virtual communication instruction queue (CIQ). The VP triggers the Extol NIC via the mapped trigger page. After the VPID has been successfully inserted into the conditional store buffer Extoll will load the corresponding context. After the context has been loaded the next VCI of this VP will be fetched and executed. After the execution a corresponding notification will be stored in the notification queue (NQ).

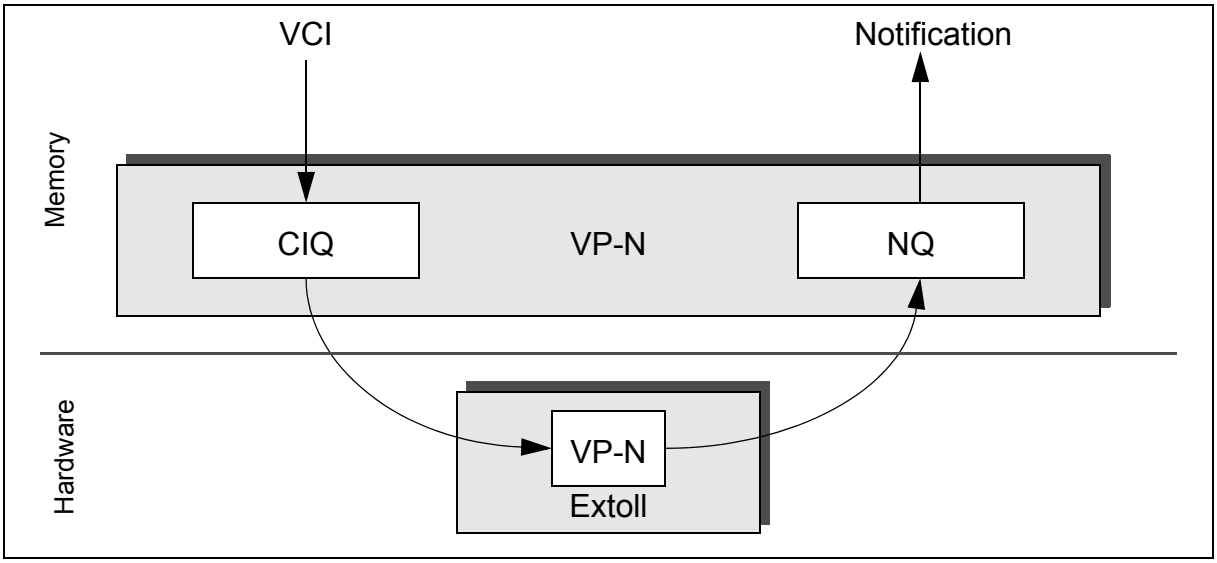


Figure 3-6 : CIQ Flow

To achieve the highest possible performance the Extoll SAN supports a special communication scheme called ultra low latency transaction (ULTRA). The ULTRA mechanism is designed for small messages (<= 64 Bytes) and use a PIO approach. More details of the ULTRA unit can be found in the diploma thesis of Heiner Litz ([LITZ05]). For fast synchronisation between processes the Extoll SAN also supports a hardware barrier.

3.3 Global Information for VPs

Table 3-1 lists the global information for each VP.

Name	*Description*
VPD table base address	The start address of the VPD table in the memory. The address must be a **physical address**.
VPD upper base	Specifies the end address of the VPD table. Together with the base address the length of the VPD table is specified. The address must be a **physical address**.
CIQ length[a]	Specify the length of the CIQ in bytes. The length value must be a multiple of the VCI size.
WDT length[a]	Specify the length of the WTD in bytes. The length value must be a multiple of the window descriptor size
SDR length[a]	Specifies the length of the SDR in bytes.
RDR length[a]	Specifies the length of the RDR in bytes.
NQ length[a]	Specify the length of the NQ in bytes. The length value must be a multiple of the notification size

Table 3-1: Global information for VPs

a. This value is global and specifies the length of each VPD.

3.3.1 Global Information for Event Queue

Table 3-2 lists the global information that configure the event queue.

Name	*Description*
EventQ base address	This is the physical base address of the event queue.
EventQ length	The length of the event queue in amount of events that can be stored in the event queue.
EventQ ReadPtr	The read pointer of the event queue. This pointer is managed by software.
EventQ WritePtr	The write pointer of the event queue. This pointer is managed by hardware.

Table 3-2: Global information for the Event Queue

3.4 Extoll Descriptors

3.4.1 Virtual Port Descriptor

.

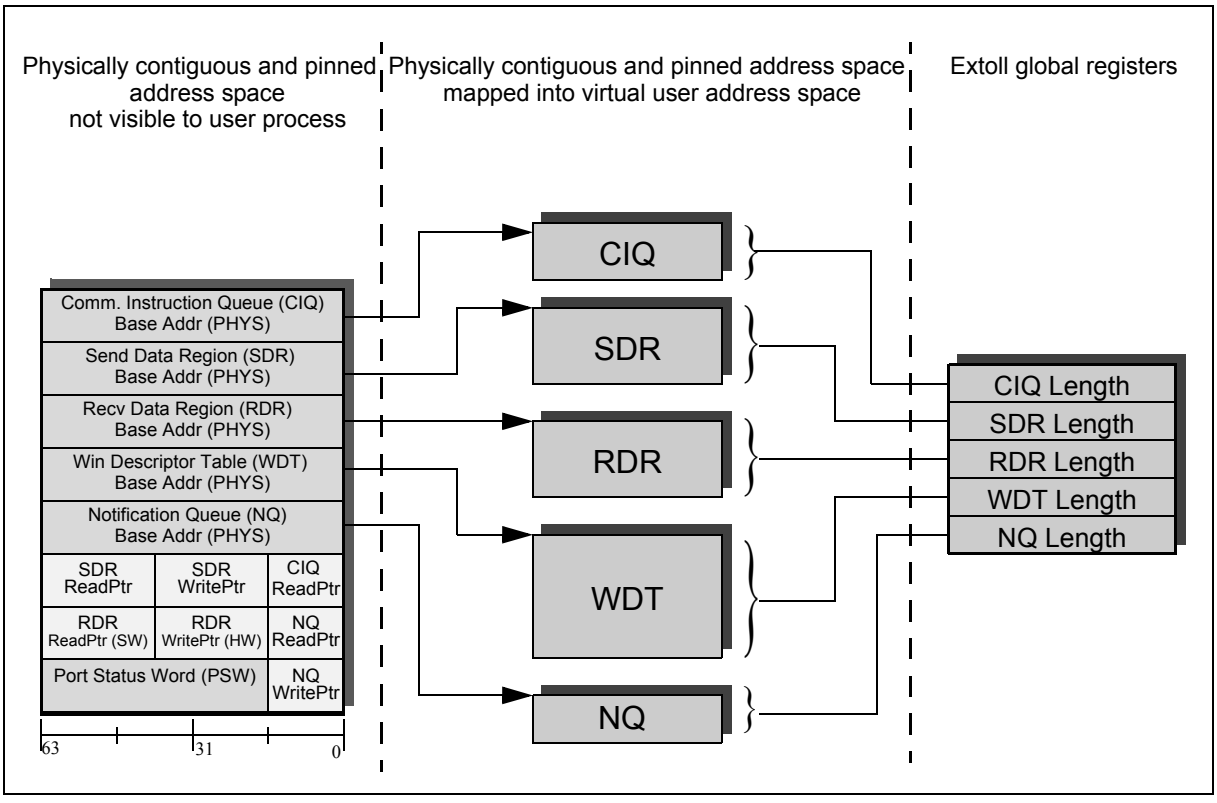


Figure 3-7 : Virtual Port Descriptor

The virtual port descriptor (VPD) describes the current state of a virtual port. Table 3-3 lists the members of the virtual port descriptor. The current layout of the virtual port descriptor is shown in Figure 3-7.

Name	*Description*
CIQ base address	The **physical address** where the CIQ starts in main memory. The length of this queue is equal for all virtual ports. See Table 3-1 for more information.
SDR base address	The **physical address** where the send data region starts in main memory. The length for the buffer is the same for all virtual ports. See Table 3-1 for more information.
WDT base address	The **physical address** where the window descriptor table starts in main memory. The length of this descriptor table is the same for all virtual ports. See Table 3-1 for more information.
RDR base address	The **physical address** where the receive data region starts in the main memory. The length for the buffer is the same for all virtual ports. See Table 3-1 for more information.
NQ base address	The **physical address** where of the notification queue starts in the main memory. The length for the buffer is the same for all virtual ports. See Table 3-1 for more information.
CIQ ReadPtr	The read pointer of the communication instruction queue. This value is managed by hardware. This pointer counts in VCI and not in bytes. See Table 3-4 for more information.
RDR ReadPtr	The read pointer of the receive data region. The meaning of this value depends on which receive system is used. See Table 3-4 for more information.
RDR WritePtr	The write pointer of the receive data region. The meaning of this value depends on which receive system is used. See Table 3-4 for more information.
SDR ReadPtr	The read pointer of the send data region. This value is managed by hardware. See Table 3-4 for more information.
SDR WritePtr	The write pointer of the send data region. This value is managed by the user. See Table 3-4 for more information.
NQ ReadPtr	The read pointer of the notification queue. The value is managed by the user. See Table 3-4 for more information.
NQ WritePtr	The write pointer of the notification queue. This value is managed by hardware. See Table 3-4 for more information.
PSW	This is the configuration status word of the VP. See Table 3-4 for further information.

Table 3-3: Virtual Port Descriptor Members

For performance reasons the virtual port descriptor has a limited, fixed size of 64 Bytes. Because of the limited space it is necessary to avoid the storage of useless data. In general the pointers are incremented by the size of the corresponding queue descriptor (e.g. window de-

scriptor). Therefore the least significant bits of the pointers are always 0 and can be omitted in the virtual port descriptor. Table 3-4 gives an overview of the different pointers and their unit sizes.

Pointer	*Size*[a]	*Unit Size*[b]	*Description*
CIQ ReadPtr	16	512[c]	The current read pointer of the virtual port for fetching new VCIs. This pointer is managed by hardware and count in VCI units.
SDR ReadPtr	24	64	The SDR read pointer of the virtual port. This pointer is managed by hardware. This pointer always counts in 8 bytes words.
SDR WritePtr	24	64	The SDR write pointer of the virtual port. This pointer is managed by the user. This pointer always counts in 8 bytes words.
RDR ReadPtr	24	64	• In the case of **DMA receive** this pointer is the **read pointer** of the DMA RDR. In this case the pointer counts in 8 bytes words. • In the case of **Posted Receives** this is the **write pointer** of the PRQ. In this case the pointer counts in posted receive descriptor units. In both cases the value is managed by the user.
RDR WritePtr	24	64	• In the case of **DMA receive** this pointer is the **write pointer** of the DMA RDR. In this case the pointer counts in 8 bytes words. • In the case of **Posted Receives** this is the **read pointer** of the PRQ. In this case the pointer counts in posted receive descriptor units. In both cases the value is managed by hardware.
NQ ReadPtr	16	512[b]	The NQ read pointer of the virtual port descriptors. The pointer counts in NQE units and is managed by the user.
NQ WritePtr	16	512[b]	The NQ write pointer of the virtual port descriptors. The pointer counts in notification units and is managed by hardware.

Table 3-4: Pointer metrics

a. Size of the descriptor member in bits.
b. The size of the corresponding queue entries.
c. The size of a VCI respectively of NQE.

3.4.1.1 Virtual Port Status Word

The virtual port status (PSW) is a 48 bit vector that contains several management and configuration bits. Table 3-5 lists the defined bits.

Name	*Description*
enabled	This bit indicates that the corresponding VP has been opened by an user. If this bit is not set the Extoll **must not** allow the execution of any operations.
pr_enabled	This bit indicates that the process uses the posted receive mechanism instead of DMA receive.
phys_addr_enabled	This bit indicates that in the case of activated posted receives the posted receives may contain physical addresses instead of virtual addresses. For security reasons only kernel space users are allowed to use physical addresses.
rdma_enabled	This bit indicates if the VP has a valid pointer to a window descriptor table or not. This option is used to reduce resources if some VP do not need RDMA functions.
events_enabled	If this bit is set an event for the device driver will be generated for all operations that are not represented by a VCI else not.

Table 3-5: PSW Members

3.4.1.2 Virtual Port Map

Table 3-6 shows all VPIDs that are reserved for a special purposes. All special VPIDs are located at the beginning of the VPID range. If the special VPIDs were allocated at the end of the VPID range then it would be impossible to reduce VPID range and therefore the required resources.

Number	*Purpose*	*Description*
0x0000	Reserved	This VPID is used to specify an invalid VPID. This VPID **must not** be used by an user.
0x0001	Daemon	This VPID is reserved for the daemon process.

Table 3-6: Virtual Port map

3.4.2 Window Descriptor

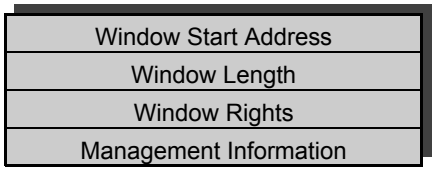


Figure 3-8 : Window Descriptor

The window descriptor (WD) describes a memory region that is allowed to be accessed by RDMA operations. The layout of the WD is shown in Figure 3-8. The members of the window descriptor are described in Table 3-7.

Name	*Size*	*Description*
Win Start Address	64	The **virtual address** where the window starts.
Win Length	64	The length of the window in bytes.
Right Flags	64	The rights of the window.
Management	64	Reserved for the software to manage the windows.

Table 3-7: WD Members

3.4.3 Notification Descriptor

The notification descriptors are described in “Notification system” on page 31.

3.4.4 Virtual Communication Instruction

The Virtual Communication Instruction descriptor consists of two parts. The first 3*64-bit data words belong to the header which is equal for all VCIs. The last 5*64-bit data words are different for each VCI type (e.g. SEND, FAST_SEND, ...). For detailed description of the variable part refer to the diploma thesis of Dirk Franger ([FRANGER04]).

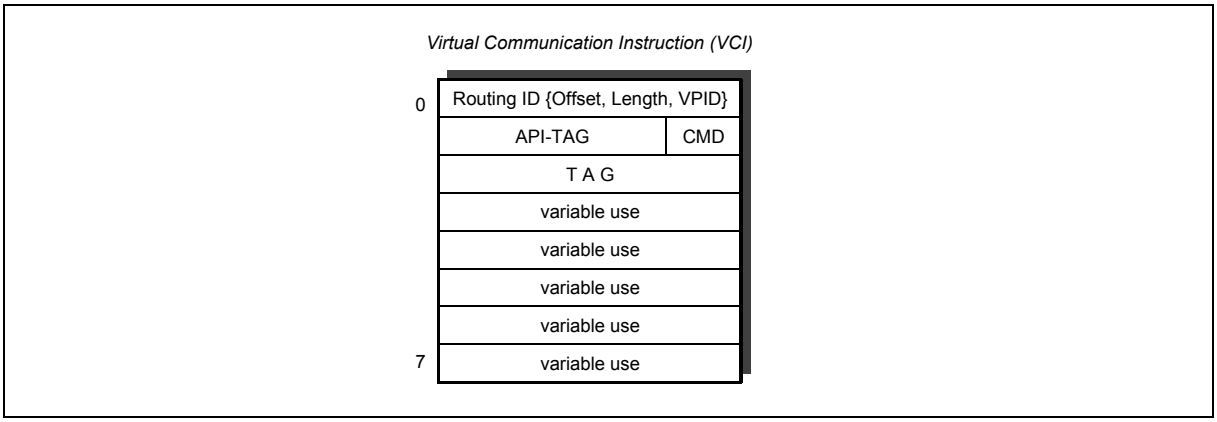


Figure 3-9 : VCI Descriptor

3.4.5 Consistency

In the case of the VP descriptor and the window descriptor there is a possible race condition. Both these descriptors are asynchronously fetched from the hardware every time it needs such a descriptor. This leads to the possible problem that, while the software is writing a descriptor the hardware is simultaneously fetching the descriptor. This means that the hardware will receive a descriptor that is a mixture of old and new data. Therefore the hardware must be prepared to handle invalid descriptors. An alternative would be to modify the descriptors in an order that avoids problematic descriptors. It is important to note that the Extoll NIC should not need to fetch a descriptor before this descriptor has been modified. In general this behaviour is caused by error in the application or system.

3.5 Notification system

Definition 3-1 : A **notification** is a fixed size information unit that is generated by the hardware. The notification is used to transfer information (e.g. completion of operations) from the hardware to the user of a specific VP.

In the design of Extoll each virtual port has a notification queue (NQ). The task of the NQ is to inform the user about completed operations. The Extoll NIC will insert a notification descriptor (see Table 3-10) into the NQ of the corresponding VP after the operation completed. The notification queue is implemented as a ring-buffer with wrap-around semantic. The following operations cause generation of a notification entry:

- All posted VCI operations generate a notification.
- Posted receives generate a notification.
- All DMA receives generates a notification.
- A completed ULTRA send message can generate a notification.
- A completed RDMA operation generates a notification on the destination host.

Table 3-8 compares the information that needs to be stored in a notification descriptor depending on the kind of the operations. As shown in Table 3-8 the posted-receive and the RMA operation limit the minimum size of the notification descriptor.

Entry	*VCI completion*	*RMA target*	*(DMA) Receive*	*Posted-Receive*
CMD	+	+	+	+
TAG	-	+	+	+
API-TAG	+	+	+	+
Counterpart-ID	-	+	+	+
ERR-CODE	+	+	+	+
OFFSET	-	+	+	+
SIZE	-	+	-	+
WIN-ID	-	+	-	+

Table 3-8: Information for Notification Descriptors

Table 3-9 offers a description of the notification descriptor members. The layouts of the different notification descriptors are presented in Figure 3-10.

Name	*Size*	*Description*
CMD/ERROR	16	Identifies which kind of operation caused the creation of the notification. The error part contains the status of the completion.

Table 3-9: Notification Members

Name	*Size*	*Description*
TAG	64	The tag of the operations that has been posted to CIQ is stored in this field. In the case of a receive or a RDMA target operation the TAG of the remotely posted VCI operation will be stored in this field.
API-TAG	48	The API-TAG is the API-TAG of the posted (either locally or remotely) VCI.
COUNTERPART-ID	48	The ID that identifies the Extoll NIC that executed the VCI. The COUNTERPART-ID consists of the ORIGIN-EXTOLL-ID and the ORIGIN-VPID.
OFFSET	64	In the case of an RDMA operation this value specifies the offset in the target memory window. In the case of a DMA receive this value specifies the offset in the RDR. In the case of a send this offset is the offset in the SDR. For the case of the FAST_SEND the OFFSET must be set to 0xffffffff, because the data is stored inside the VCI. In the case of a RDMA operation this value is always the local offset (e.g. PUT the origin process that started the operation gets the src-offset while the remote process get the target-offset).
SIZE	64/16	The size of the transferred data in 64 bit words.
WIN-ID	16	The window is set to the target WINDOW-ID of the operation. If the operation does not have a target WINDOW-ID then the WIN-ID must be set to 0xffff.
RDR ReadPtr	32	The current read-pointer of the RDR.
RDR WritePtr	32	The current write-pointer of the RDR.
CIQ ReadPtr	16	The current read-pointer of the CIQ
VCI index	16	The index of the corresponding VCI inside the CIQ. If the source for a notification has not a corresponding VCI the value is 0xfffff.
PRQ index	24	The index of the posted

Table 3-9: Notification Members

The notification descriptors does not contain a write pointer for the NQ. The software detects new entries by a special signature directly in the NQ instead of checking for a changed write pointer.

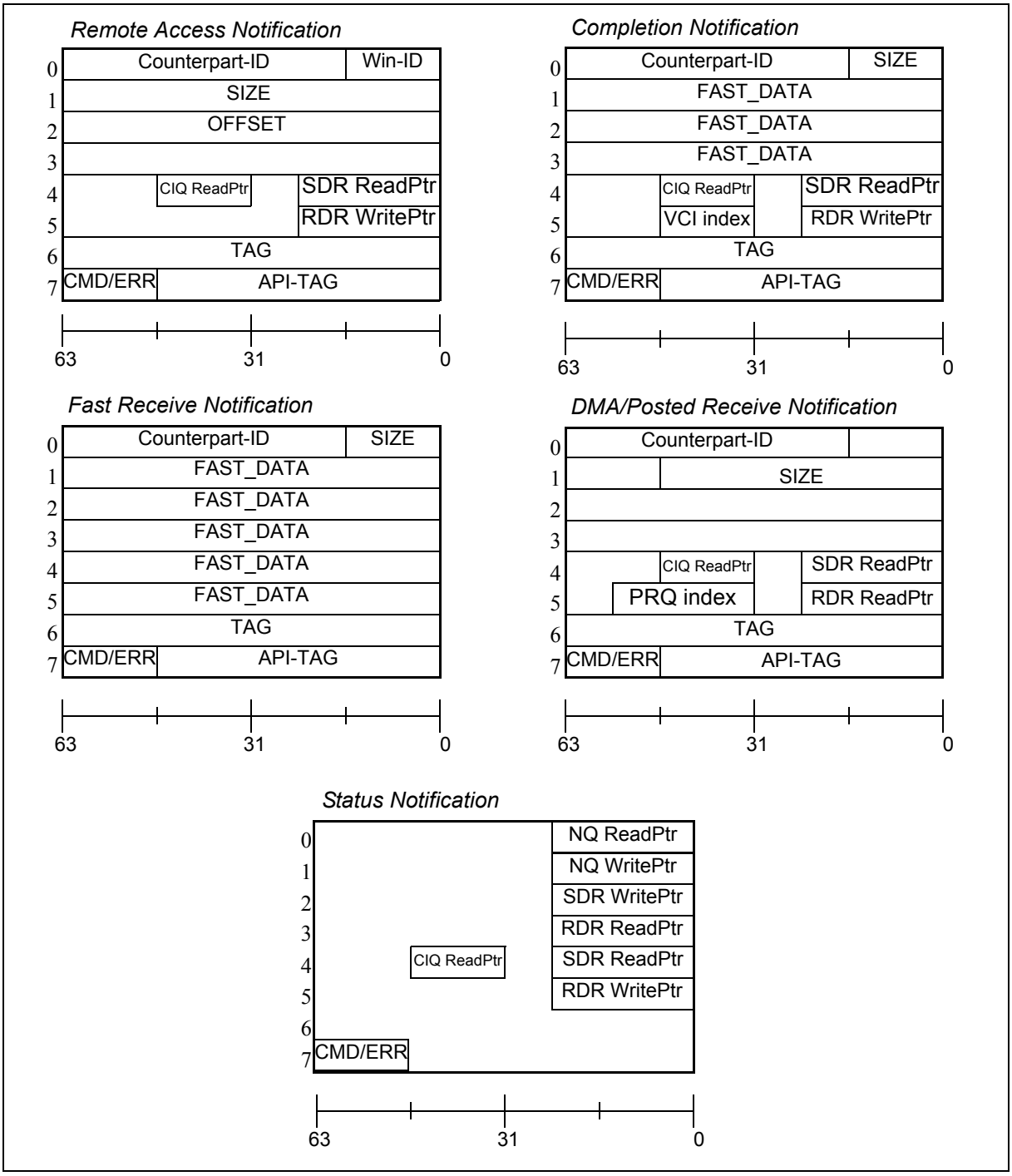


Figure 3-10 : Notification Formats

To increase the performance a snapshot of the current pointer set is stored in each notification. With this approach the user gets information about the current pointer state of the VP without extra costs for accessing the Extoll. The notifications are stored in the notification queue in the main memory. The whole queue is initialized with invalid notifications. An invalid notification is identified by a special error code. The hardware will overwrite an invalid notification with a valid notification. The software will detect the new valid notification and consume it from the NQ. If the CMD would be stored at the beginning of the notification descriptor there could be a race condition. The software could detect a notification descriptor with a valid error code that

is only partially written and consume the notification descriptor. To avoid this race condition the cmd/error is stored at the end of the notification descriptor. In this case the software will detect a valid notification descriptor only after the whole descriptor has already been written.

3.5.1 Notification Error Codes

Table 3-10 gives an overview of all error codes that have been identified.

Name	*Description*
ERR_NOERR	Obviously no error occurred
ERR_CMD_INV	Invalid command
ERR_OVPID_INV	Invalid origin VPID
ERR_ROUTE_INV	Route exceeding upper bound / routing length=0
ERR_OWINID_INV	Invalid origin WINID (disabled or exceeding upper bound)
ERR_OWINID	Segfault on origin WINID
ERR_OOFFSET	Misaligned origin offset
ERR_OLENGTH	Misaligned origin length
ERR_TVPID_INV	Invalid target VPID
ERR_TWINID_INV	Invalid target WINID (disabled or exceeding upper bound)
ERR_TWINID_CAPA	Invalid target WINID capability
ERR_TWINID	Segfault on target WINID
ERR_TOFFSET	Misaligned target offset
ERR_TLENGTH	Misaligned target length
ERR_ROUTE_BROKEN	Destination tag does not match, link down, ...

Table 3-10: Notification Error Codes

3.6 Event System

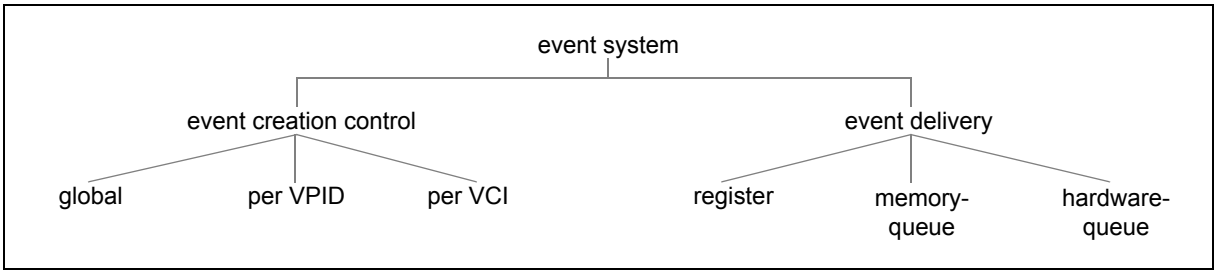


Figure 3-11 : Decision tree for the Event System

The Extoll event system is an abstract communication system between the Extoll NIC and the driver. The event system is a uni-directional channel where information is transferred from the Extoll NIC to the Extoll device driver. The design of the event system is based on the following conditions:

- The event system should be as abstract as possible. This means that the general design of the event system should not depend on any hardware specific feature.

- The event system should deliver the events fast and efficiently to the device driver.
- The device driver should be able to handle the event in an efficient way.

Definition 3-2 : An **event** is a fixed size information unit that is transferred from the Extoll NIC to the device driver. The generation of events does not include the signalling.

Definition 3-3 : The **event signalling** is a mechanism to inform the device driver about the existence of new events. The event signalling mechanism is hardware specific. In general it is not necessary to signal each generated event to the device driver as long as there does not appear a race-condition a or dead-lock. The most used approach for event signalling is the hardware based interrupt.

Definition 3-4 : The **event signal handler** is the software mechanism that is activated when an event signal is emitted. This mechanism is responsible for processing the events.

3.6.1 Event creation control

As mentioned above it is not necessary to signal the generation of every event, but in the worst case every event needs to be signalled. In this worst case the flood of event signals could have a negative impact to the whole system. Therefore it is necessary to reduce the amount of generated event signals to the minimum possible amount.

It is not possible to reduce the amount of events that are necessary for correct execution of the hardware and driver (e.g. address translations, management information, errors, ...). But it is possible to save events that are related to user communication. These kinds of events are necessary for the device driver to wake up a sleeping process. As shown in Figure 3-11 there exists 3 different ways to control the event generation for the user communication events. These approaches will be discussed in the following sub-sections.

3.6.1.1 Global Event Creation Controlling

In this case the generation of communication events is controlled globally for all VPs. This means that events for communication are generated either for all VPs or for none.

Advantages

- This approach needs only one global bit in an Extoll control register.

Disadvantages

- This approach is not per process. This means that if there is at least one process that needs the generation of communication events then communication events must be created for all processes. In this case unnecessary events would be generated.

3.6.1.2 Event Creation Control per Virtual Port

In this case the generation of communication events is controlled per VP. Each user can select for his VP if communication events should be generated or not.

Advantages

- In this mode each user has the full control over his own VPs. This allows a finer granularity than the global event control approach.
- The amount of generated events can be reduced for the case that a process does not need any kind of events.

Disadvantages

- This approach is still globally for the whole VP. This leads to the fact that still unnecessary events will be created.
- The change of the event control behaviour during runtime is an expensive operation. Because the VPDs are only accessible for the device driver the process would need to jump into the device driver. After the modification of the VPD in the main memory the cached version in the Extoll NIC needs to be synchronised with the main memory.

3.6.1.3 Event creation control per VCI

In this approach the user can select for each VCI if the corresponding completion should generate an event or not.

Advantages

- Very fine granularity of event generation.
- The VPD does not need to be modified.

Disadvantages

- This approach has the disadvantage that not all operations that could cause the creation of an event also have a corresponding VCI (e.g. receives)
- This system needs space for one additional bit in each VCI.

3.6.1.4 Conclusion : Event creation control

The best event creation control mechanism is offered by the event creation control per VCI approach. As mentioned has this approach the drawback that not all kind of operations (e.g. receives) have a corresponding VCI. The other approaches have the disadvantage that their control capabilities for the event creation are to coarse.

The solution is the combination of the event creation control per VP approach and the event creation control per VCI approach. For all operations that have a representation via a VCI the event creation is controlled via a bit in the VCI. For all other classes of operations there exists a corresponding bit in the VPD. For performance reasons and to avoid race conditions the decision of the global bit in the VPD is fixed for the whole time a VP is opened.

3.6.2 Event delivery

After the hardware created a new event the device driver needs to be informed about this new event. After the device driver has been activated the device driver needs to get access to the event. As mentioned the event signalling between the hardware and the software layer is an expensive operation. Therefore the usage of this mechanism should be minimised as much as pos-

sible. The realisation of the device driver access to the generated events must be as fast and efficient as possible. Figure 3-11 shows possible delivery mechanisms which are discussed in the following sub-sections.

3.6.2.1 Lazy Event Signalling

To avoid the expensive event signalling the lazy event signalling principle could be introduced. Like the normal event signalling system the lazy event signalling only emits an event signal when the event system is enabled. The special feature of the lazy event signalling system is that everytime when an event signal has been emitted the event signalling system will be automatically disabled. Therefore only the first event that is created will be signalled. All events that will be created later will not emit an event signal. The first emitted event signal allows the device driver to be activated. After the device driver has been activated the first event will be processed. After the first event has been processed the device driver tries to process more events until no further events are available. When there are no further events to process the device driver needs to enable the event signalling system again to be informed for new events. After the device driver has enabled the event signalling system again, a further check for new events needs to be done. This additional check is necessary to avoid a race condition.

Advantages

- Reduce the emitted event signals to the minimum necessary amount by processing several events with only one emitted event signal.

Disadvantages

- Need an extra access to enable the event signalling system again.
- In the worst case this approach causes more overhead than the approach where every event is signalled.

3.6.2.2 Global Event Register

A simple approach is to create a global event register. The Extoll NIC stores the events in this event-register and emits an event signal. The event signalling system will then be disabled. The event signal handler will read the event from the event register and process it. This reading of the last data word will release the event register for the next event. Then the event signal handler tries to get more events until an invalid event is returned. When an invalid event has been received, the device driver enables the event signalling system again. To avoid race conditions the event signal handler has to check for a new event after enabling the event signalling system.

Advantages

- Simple realisations.

Disadvantages

- The single event register is a bottleneck in the system. Only one event at a time can be exchanged.
- The event-register solution requires 3 Extoll accesses in the worst case to transfer a single event: read the first event, read the invalid event and enable the event signalling system.
- Read operations are more expensive that write operations.

3.6.2.3 Hardware Event Queue

As shown the approach with the event register in hardware is very limited. In general it is possible that in a very short period of time several events need to be created. In this scenario the single register approach is not sufficient. The next step is to use several event registers instead of using only one event register. Because an event signal has in general no ability to transfer additional information, the event registers need to be arranged and managed as a queue. The corresponding management information is held in registers inside the hardware (see Figure 3-12).

Newly generated event are inserted into the event queue (see Figure 3-12 (1)). This insertion causes the triggering of the events signal system. If the event signalling system is active a corresponding event signal is send to the event signal handler and the event signalling system is automatically disabled (see Figure 3-12 (1)). The event signal activates the event signal handler in the device driver. Because the event signal handler manages a private copy of the event queue management information, the event can be directly fetched (see Figure 3-12 (3)). The reading of an event from the event queue automatically releases the corresponding entry in the queue by updating the read pointer. While the event handler is receiving the event signal and processing the event, more events could be inserted into the event queue (see Figure 3-12 (2)). The insertion of these events will also trigger the event signalling system, but no event signal will be emitted because the event signalling system has been disabled before (see Figure 3-12 (2)). After the first event is processed the event signal handler will try to read the next event from the queue. The event signal handler will continue to read new events until an invalid event is returned. In this case the event signal handler will enable the event signalling system again and do an extra check for events to avoid a race condition.

In the worst case where always only one event is available the mechanism would need 2 read operations, the first to read the event and a second to read the invalid event. At the end the event signal handler needs to perform a write operation to enable the event signalling system again. This means that in the worst case it is necessary to access the I/O bus 3 time. But the mechanism of the queue approach can be further optimized.

The first optimization could be to return to each event additional information of how many further events are stored in the event queue in the moment of the read operation. This would allow the event signal handler to read outstanding events with a burst. In the worst case whenever only one event is available, this would lead to one read operation for the event. Because the additional information of the event tells the event signal handler that there are no further events and the event signalling system could be enabled directly. This mechanism improves the worst case scenario to 1 read operation for the event and one write operation to enable the event signalling system again. The drawback of this mechanism is that there is a race condition. In the time between the reading of the event and the activation of the event signalling system there could appear new events in the queue. To overcome this drawback the event signalling system could be modified to emit event signals also on an activation when there is a valid event in the event queue. The preferred approach would be to use a read operation instead of a write operation to enable the event signalling system. In the return value of the read operation an error could returned to inform the event signal handler about additional existing events in the event queue.

Advantages

- More capacity than a single event register.
- The number of extoll access in the worst case is lower than for the single event register.

Disadvantages

- The capacity of the hardware queue is fixed and limited.

- Requires more hardware resources.

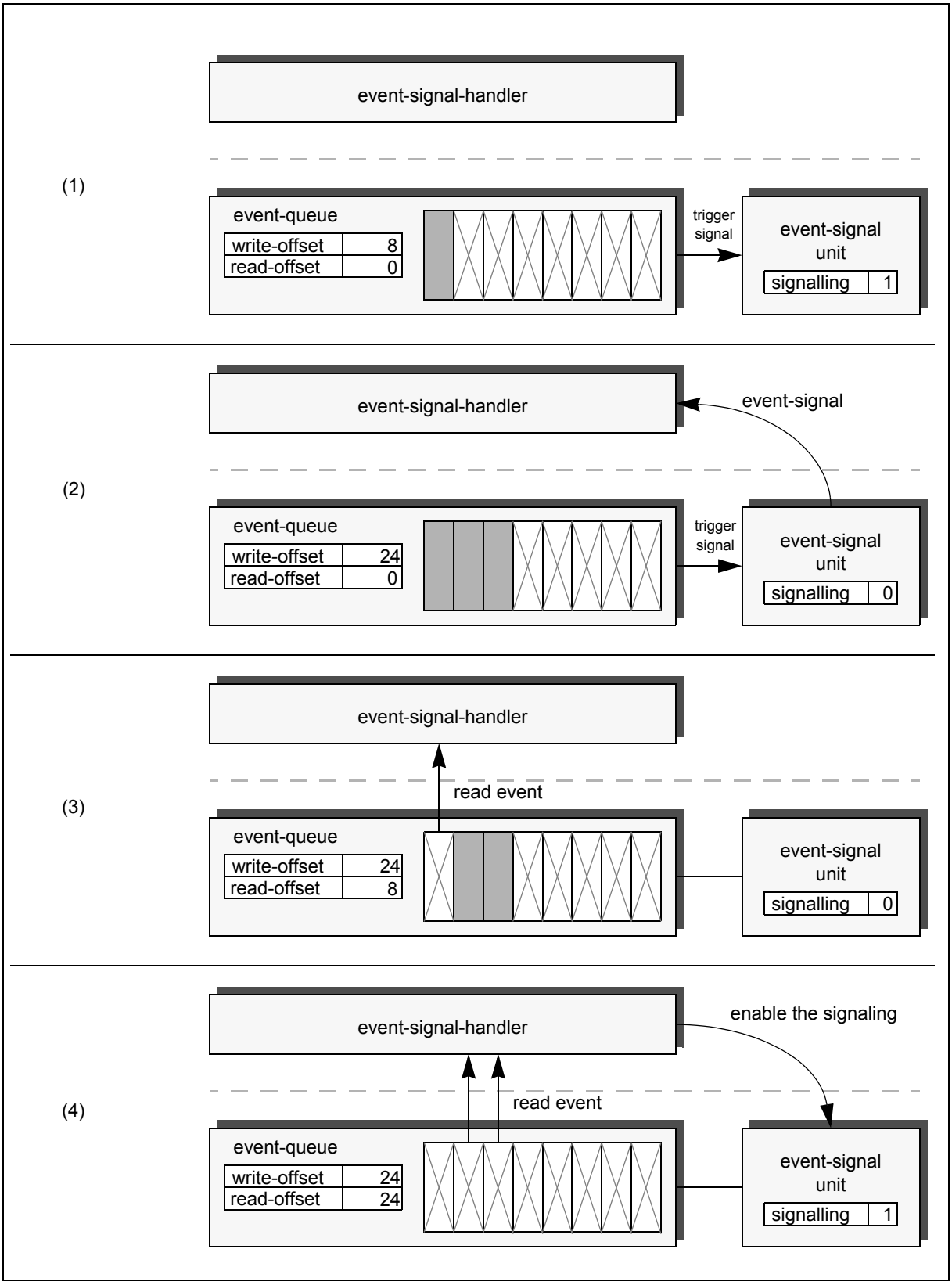


Figure 3-12 : Hardware Event Queue

3.6.2.4 Memory Event Queue

The main drawback of the hardware based event queue is the fixed and limited size of the queue. This is a general problem of all hardware based storage mechanisms. A solution for this problem is to move the event queue from the hardware to the main memory of the host system. In the memory, the size can be bigger than in hardware and can be adapted to different systems. The memory of the event queue must be physically continuous and the queue must be managed as a ring buffer with wrap-around semantic. Therefore it is possible to describe the whole event queue in hardware by the base address, the read and write offset (see Figure 3-13). The whole event queue must be initialized with the signature of invalid events.

The principle of the memory based event queue is similar to the hardware based event queue. New events are stored at the current write offset position in the queue. In the case of the memory based event queue, the events are directly written into the main memory (see Figure 3-13 (1)). After every event that has been written to the event queue the event signalling system is triggered (see Figure 3-13 (2)). If the event signalling system is enabled an event signal is emitted to the event signal handler and the event signal system will be disabled otherwise no signal will be emitted (see Figure 3-13 (2)). The event signal handler will be activated by the event signal and process the next event in the event queue. To avoid unnecessary accesses of the Extoll NIC the event signal handler uses a local copy of the management information. After an event has been consumed the event signal handler overwrites the corresponding entry in the event queue with an invalid event signature. After the event has been processed, the event signal handler updates the local management information about the event queue. Like in case of the hardware event queue the event signal handler checks if there are more valid events in the event queue. When the event signal handler detects an event with an invalid signature then there does not exit any further events. Because the event queue is located in the main memory the cost to check for further events are cheaper compared with the cost of a hardware access. When there are no further events in the memory queue the event signal handler has to update the hardware read offset in the Extoll to free the processed events from the hardware view of the event queue (see Figure 3-13 (6)). Then the event signal handler also needs to enable the event signalling system again. To avoid an extra access the activation of the event signalling system is coupled with the read offset update.

In the worst case where always only one event is available this system requires two write operations. The first write from the Extoll NIC to the main memory writes the event into the main memory. The second write from the event signal handler updates the event queue read offset in the Extoll NIC.

Advantages

- This approach requires very few hardware resources.
- Allows to realise a variable sized event queue.
- Buffered events in the hardware can be written with a burst.

Disadvantages

- Because the write back of the read offset is coupled with the activation of the event signalling system it is not possible to use a lazy pointer approach.

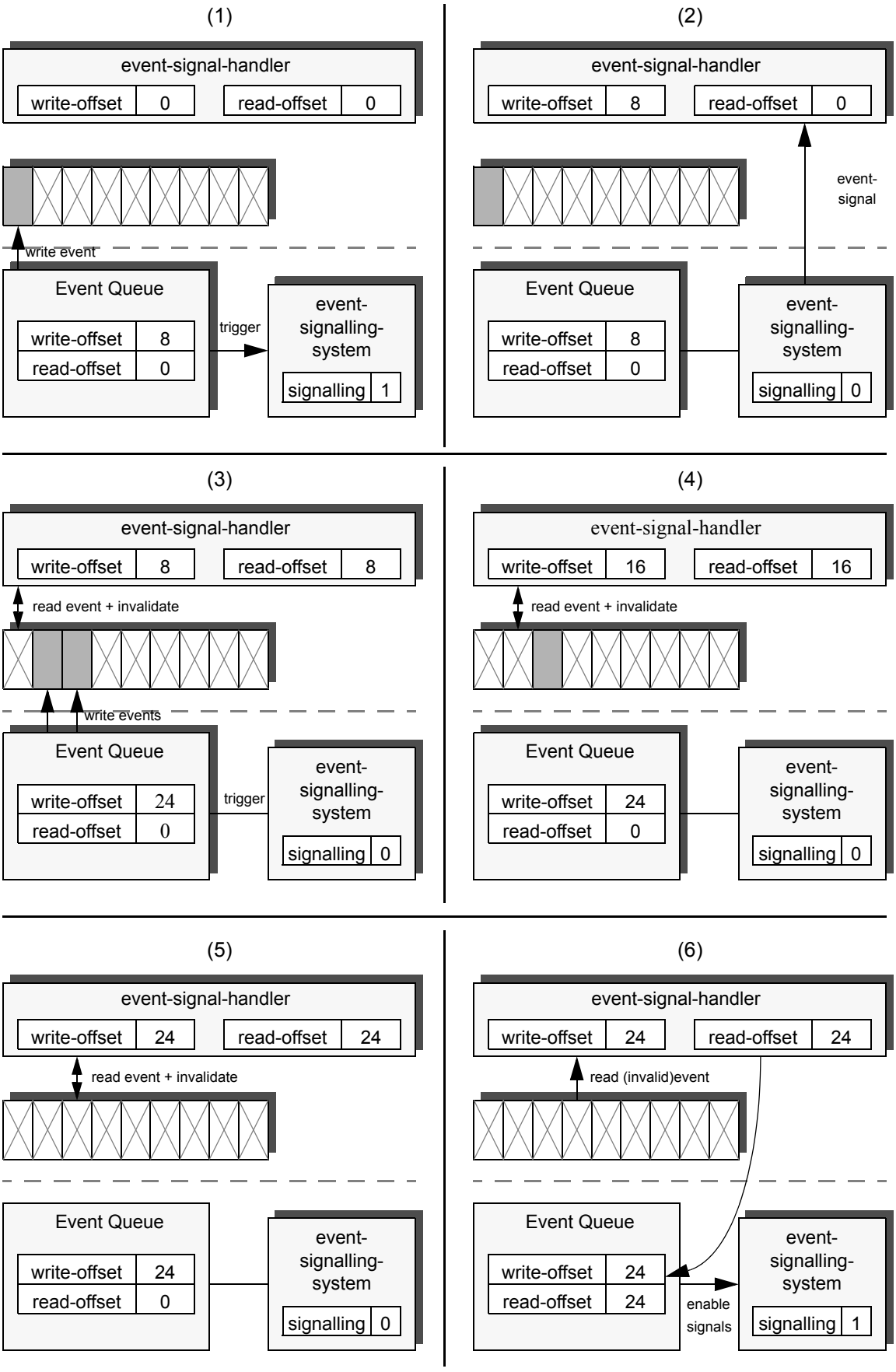


Figure 3-13 : Memory Event Queue

3.6.2.5 Conclusion: Event delivery

The event register approach is very limited and would be a bottleneck in the final design. Therefore the event queue based approach is preferred. As shown before the hardware and the memory event queue have almost similar performance shapes. Because the memory event approach is more flexible and has no hard limits like the hardware event queue, this solution is the preferred approach for the final design of Extoll.

3.6.3 Event Format

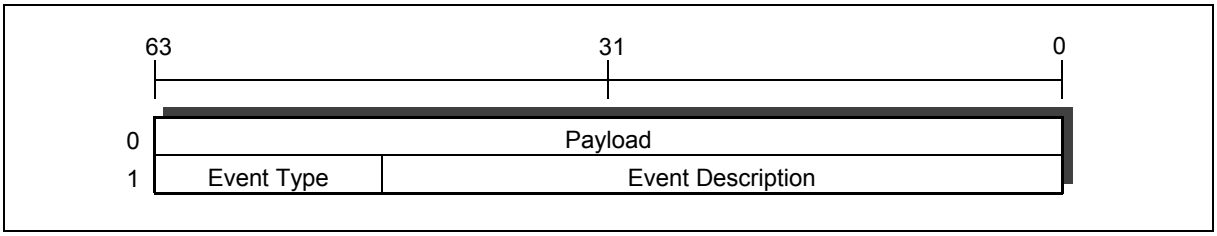


Figure 3-14 : Event layout

The event layout is shown in Figure 3-14. The event type field in the event is used to identify the kind of event. The event description field is event specific and is used to provide more details about an event. The payload is used for data that needs to be transferred by an event. The event type of 0x0000 specifies an invalid event. To avoid a race condition, as in the case of the notifications, it is important that the event type field is located on the highest possible address of the event descriptor.

3.7 Order of communication

3.7.1 General observations

The following list gives an overview of general observations in respect to the order of communication:

- The operations that belong to different virtual connections are independent and therefore can be executed in parallel.
- All the send operations that belong to the same virtual connection should be received in-order. If the sends are not received in-order sequence-numbers have to be introduced to restore the correct order of the messages.
- Because the “GET” operation does not modify data it is possible to exchange the order between “GET” operations, even for operations on the same virtual connection.
- If the order of operations is important a specific mechanism needs to be introduced to guaranty in-order of the operations.
- Accumulate operations need to be atomic on the destination for at least the minimum data word.

3.7.2 Order of Communication in MPI

In MPI messages do not overtake each other. If a process sends two messages to the same destination that match the same receive operation, then the receive operation cannot receive the second message if the first message is still pending. If a user posts 2 receive operations with the same matching information and a matching message arrives than the second receive operation cannot be satisfied while the first is still pending.

MPI does not define a specific order between RDMA operations or specific order between RDMA operations and send operations. This allows the implementation to exploit the maximum possible performance by executing these operations in parallel. In general there are cases where the order of the operations is important. For these cases MPI defines several synchronisation mechanisms.

The simplest synchronisation mechanism is the *MPI_Win_fence* ([MPI2], [UMPI2]) function. The *MPI_Win_fence* function is a collective operation across all processes that share the same window. The collective calls of the *MPI_Win_fence* function divides the time in several time slices. These time slices are called *access epochs*. In general there exist two different kinds of access epochs. In the first kind of access epochs only RDMA operations are allowed to access the memory window while in the second kind of access epochs only local accesses to a memory window are allowed. This is necessary to guarantee the correct semantic behaviour ([MPI2]). If a RDMA capable device would access the main memory at the same time as the local CPU, there could be some coherence problems of the data. Between all RDMA operations that are executed in a certain access epoch no specific order is defined. For all access epochs the following rules apply:

- All RDMA operations that have been issued in certain access epoch must apply on the target window in the same epoch. They must not apply to the target window in any other access epoch.
- The MPI_Win_fence call that finishes a certain access epoch is responsible that all locally issued RDMA operations have been completed.

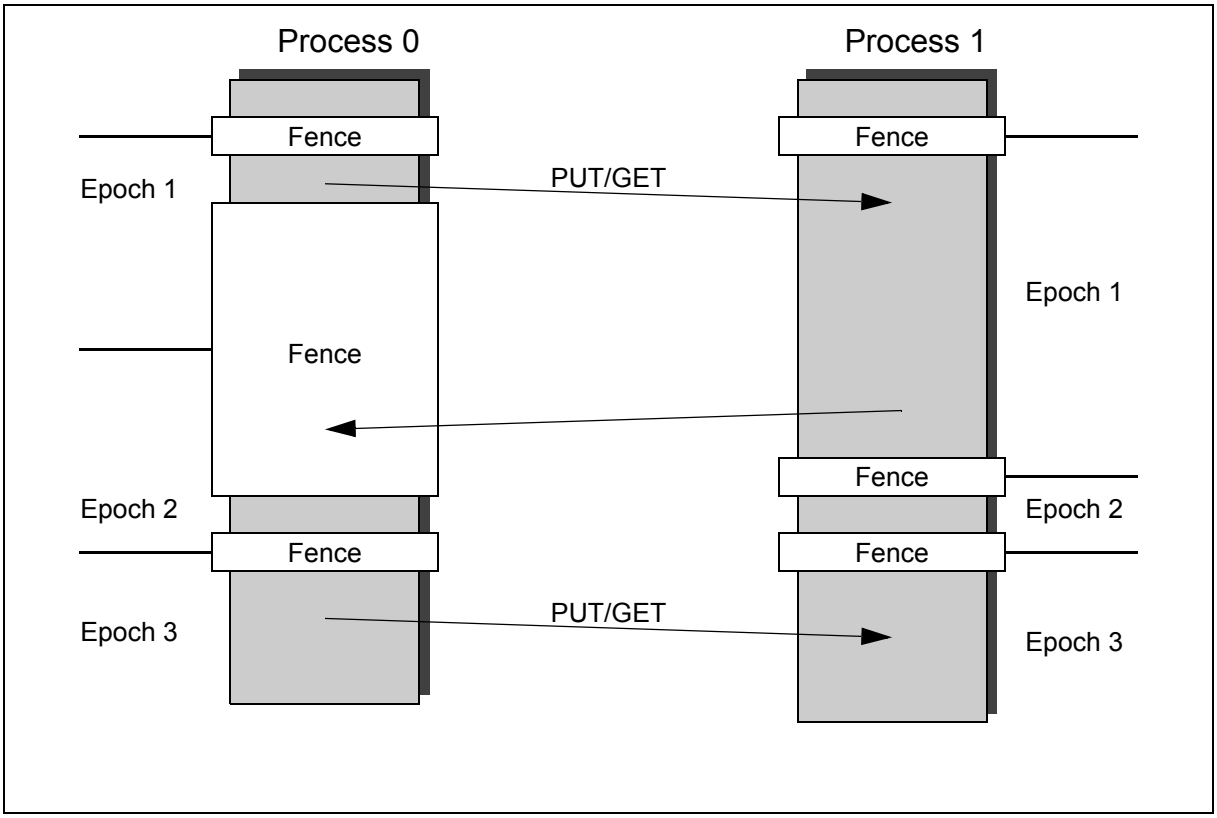


Figure 3-15 : MPI_Win_fence Operation

As mentioned the *MPI_Win_fence* call is the simplest mechanism to synchronise the access to memory windows. The *MPI_Win_fence* function is collective across all processes that share the same memory window. In many cases it is enough to synchronise with a few processes (e.g. nearest neighbours). For these cases MPI provides the *MPI_Win_post*, *MPI_Win_start*, *MPI_Win_complete* and *MPI_Win_wait* functions. These functions only synchronise between a special group of processes that share the same window.

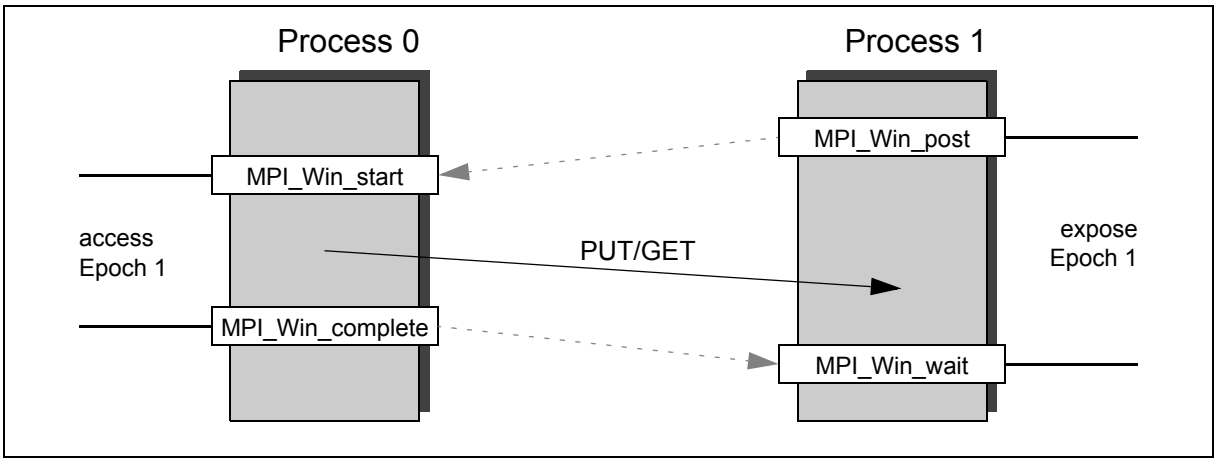


Figure 3-16 : Synchronisation between a small group of processes

The *MPI_Win_start* and *MPI_Win_complete* are used to specify the start and the end of an access epoch. The *MPI_Win_post* and *MPI_Win_wait* are used to specify the start and the end of an expose epoch. RDMA operations are always issued by the processes that opened an access

epoch. The issued operations are only allowed to access the window of processes that are in the corresponding expose epoch. For all operations that are issued in the access epoch no specific order has been defined. The *MPI_Win_complete* functions must guarantee that all operations that have been issued in the access epoch have been competed. The MPI_Win_wait function must guarantee that all RDMA operations that have been issued by all corresponding access epochs have been locally completed (see Figure 3-16). The *MPI_Win_fence* function can be seen as a special case of this synchronisation scheme (see Figure 3-17).

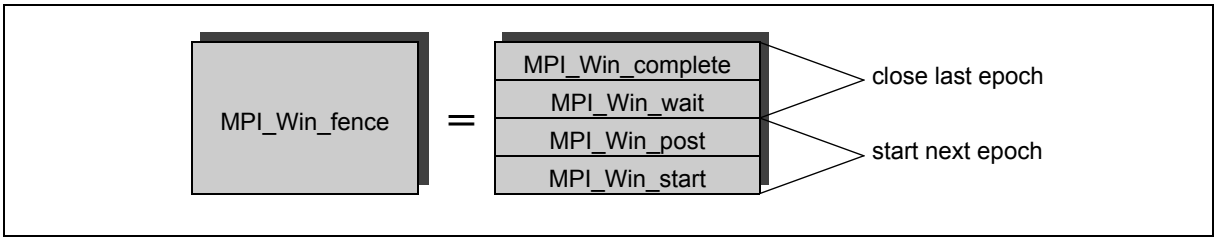


Figure 3-17 : *MPI_Win_fence* emulation

The two previously discussed synchronisation mechanisms have the drawback that the target process always needs to participate actively in the synchronisation. There are cases where a process does not need the involvement of the target process. For these cases MPI specifies the *MPI_Win_lock* and *MPI_Win_unlock* functions. With the *MPI_Win_lock* function a process can lock the memory window of a process. The grant for the lock does not require any active involvement of the target process. A memory window can be locked either exclusive or shared. After a process has been successfully locked a window the process is allowed to issue RDMA operations on the locked window. The return of the *MPI_Win_unlock* function must guarantee that the issued operations have been completed on the origin and the target side (see Figure 3-18).

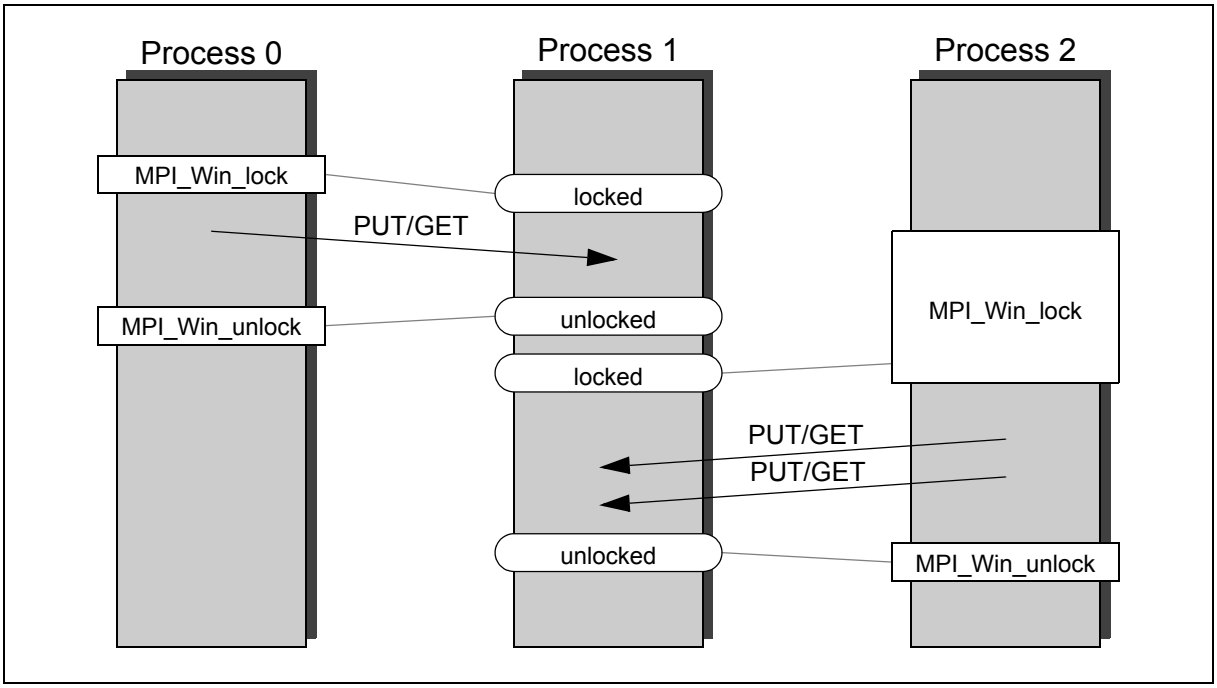


Figure 3-18 : MPI Window Locking

3.7.3 Order of Communication in DAPL

The DAPL standard ([DAT]) defines the following ordering rules:

- The data payload for the send operation matching a receive operation must be delivered into the receiver-indicated memory buffer without errors prior to the receive completion.
- Receive operations on a connection must be completed in the order of posting of their corresponding sends.
- Each RDMA write operation posted on a connection prior to a send operation must have its data payload delivered to the target memory region prior to the completion of the receive operation matching that send.

3.7.4 Order Classifications

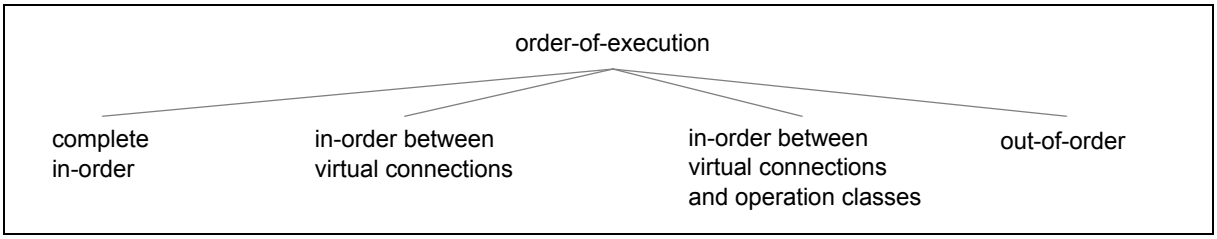


Figure 3-19 : Orders of communications

In Figure 3-19 an overview of different kind of communication orders are presented. The level of parallism increase from the left to right side.

3.7.4.1 Complete in-order

Complete in-order means that all operations are executed sequentially in the same order as they have been started by the user. In the case that the layer below the MPI respectively DAPL would support a complete in-order, no extra efforts would be necessary to guarantee the required ordering.

Advantages

- No further software corrections are necessary. This may lead to a simpler software stack.

Disadvantages

- No parallism is exploited which could lead to a performance penalty.

3.7.4.2 Complete in-order on a Virtual Connection

Complete in-order on a virtual connection means that all operations that will transfer data via a virtual connection are executed in-order. This implicitly means that operations that perform communication on two different virtual connections may be executed out-of-order.

In the case that the layer below MPI respectively DAPL would support a complete in-order on a virtual connection no extra efforts would be necessary to guarantee the required ordering.

Advantages

- Better performance can be achieved because of the exploitation of some parallism.

- No further software correction for the correct semantic is necessary.

Disadvantages

- Requires additional hardware to control/manage the execution of the VCI.
- For the communication between two processes there is no exploitation of parallism.

3.7.4.3 In-order on Virtual Connections and the same Communication Class

In this mode, operations that operate on the same virtual connection and belong to the same class of operations are executed in order. The different operation classes are send/receive and RDMA operations.

In the case that the layer below MPI would support in-order on virtual connections and the same communication class no further efforts would be necessary to guarantee the correct semantic that is defined by MPI.

In the case of DAPL there would be a problem. The DAPL standard requires that the receive that matches to a send operation completes after all previously posted RDMA write operations have been completed. In the case of an in-order on virtual connections and the same communication class there is no relation between the RDMA write operation and the send/receive operations. Therefore the DAPL standard requirements cannot be satisfied without any extra efforts.

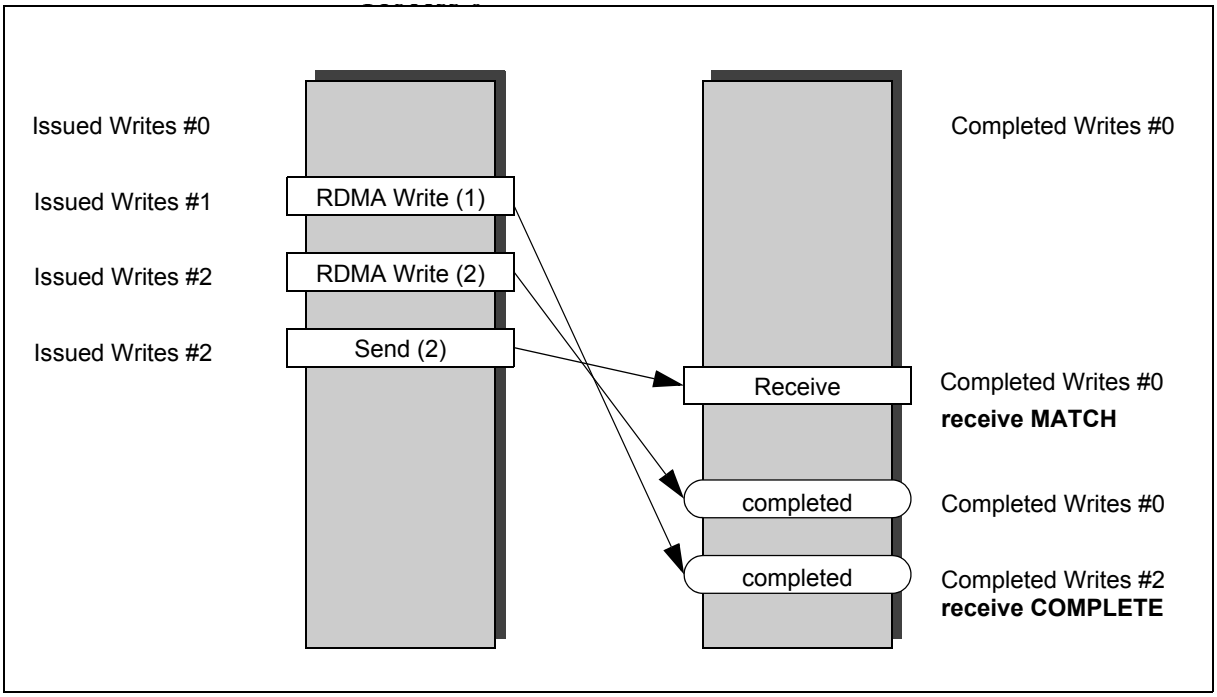


Figure 3-20 : Fix for RDMA writes in DAPL

To synchronise between the send/receive and the RDMA write operations each virtual connection manages sequence numbers for RDMA write operations. Each endpoint has 2 sequence numbers for RDMA write operation, one for the locally issued and one for locally completed RDMA operations. Both sequence numbers a sequentially incremented when either a RDMA write operation is issued or completed. All RDMA write operations and send operations are tagged with the current RDMA write sequence number. If the send operation should match to a

receive before the previously issued RDMA write operations has been completed, the completion of the receive can be delayed until all RDMA write operations have been completed (see Figure 3-20).

Advantages

- Higher exploitation of parallism.
- Exploitation of parallism on a virtual connection.

Disadvantages

- Increased hardware effort to guarantee the in-order of operations of the same operation class.

3.7.4.4 Out-of-order

Out-of-order means that there is no guarantee for any order between any operations. In the case that the layer below MPI or DAPL would not support any kind of ordering extra efforts are necessary to guarantee correct semantic behaviour for both cases.

In both cases it is necessary that the sends on a virtual connection match in order to the posted receives. This can be guaranteed by introducing sequence numbers for the send operations. With this sequence numbers the receiver would be able to reorder the arrived sends if necessary. In the case of DAPL also the synchronisation between the RDMA write and send operations needs to be added. This topic has already been discussed in “In-order on Virtual Connections and the same Communication Class” on page 47.

Advantages

- Simple hardware.
- Maximum use of parallism.

Disadvantages

- Software must restore the order of the operations after they have been disturbed or use only one operation per virtual connection at once.

3.7.5 Conclusion : Order of Communication

As shown in the previous sections is it only necessary to have software corrections for the cases of “In-order on Virtual Connections and the same Communication Class” and “Out-of-order”. In both cases the software overhead consists in sequence numbers that must be exchanged, plus in worst case it can be necessary to store some information in a temporary buffer to avoid incorrect behaviour. Because the used protocols tend to exchange only small messages via send/receive and the big chunks of data via RDMA this worst case is very unlikely. Therefore it could be an alternative to use an out-of-order system with software correction if the resulting system would be faster and/or much simpler to realise.

3.8 Send and Receive System

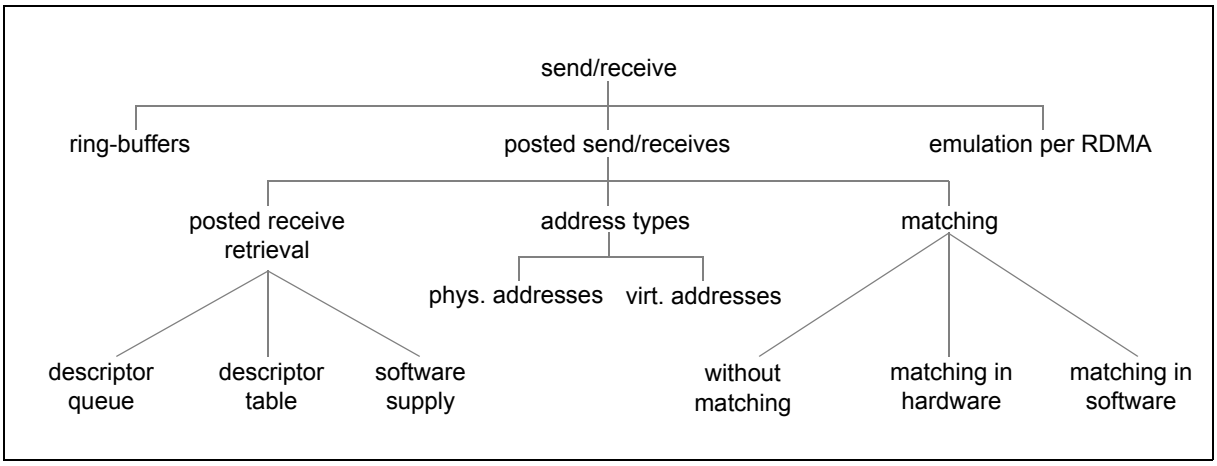


Figure 3-21 : Send/Receive Decision Tree

For the send/receive system several different approaches exist (see Figure 3-21). All approaches will be discussed and analysed in the following sub-sections.

3.8.1 Send/Receive with Ring Buffers

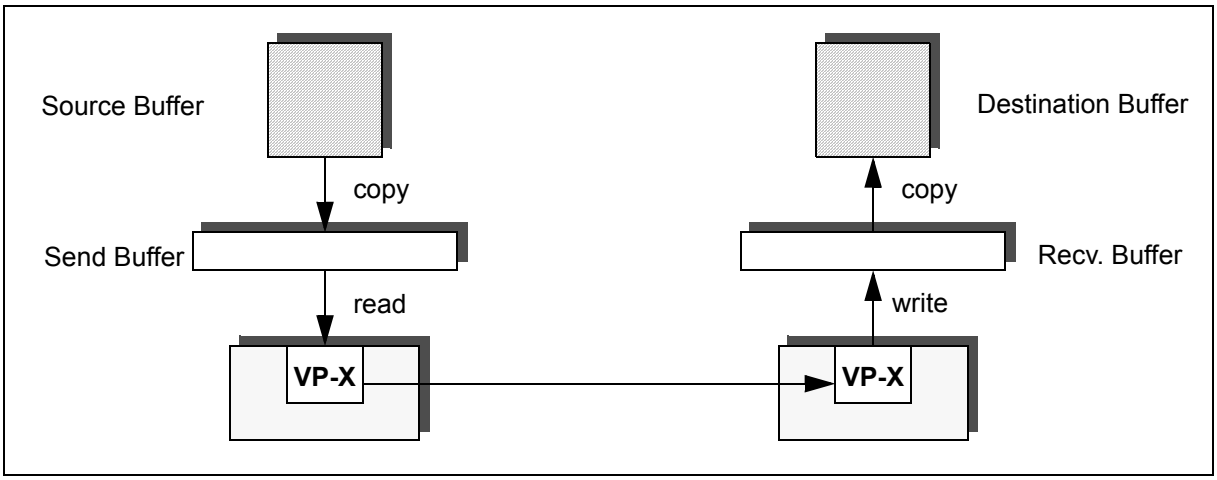


Figure 3-22 : Send/Receive with Ring Buffers

The send/receive system with ring-buffers is the same approach that has been used in the Atoll design. In this approach every VP has a send- and a receive-buffer. Both buffers consist of physically continuous memory and are managed as a ring-buffer with wrap-around semantic. To send a message the data needs to be copied into the send buffer. Then a corresponding VCI, that describes the location of the data and the destination, must be inserted into the CIQ and triggered. The data will be fetched from the send buffer when the corresponding VCI will be processed inside the Extoll. After the message has been transferred to the destination the whole message will be stored in the corresponding receive buffer of the destination VP (see Figure 3-22). After the whole message has been stored in the receiver buffer a receive notification is inserted into the NQ of the destination VP.

Advantages

- Because the memory is physically continuous there will be no trashing of the Extoll TLB.
- It is possible to manage the send buffer completely in software.
- It is possible to manage the send buffer in any way and not only in a strict ring-buffer semantic.
- Perfect resource scalabillity, because an VP always needs only a buffer pair to communicate with any number of other VPs.

Disadvantages

- Requires 2 extra copies of each data (1. copy data from source into send buffer, 2. copy data from receive buffer to destination).
- Because there exists only one send/receive buffer for all communication partners, these buffers could become a bottleneck.

3.8.2 Posted Send/Receive Operations

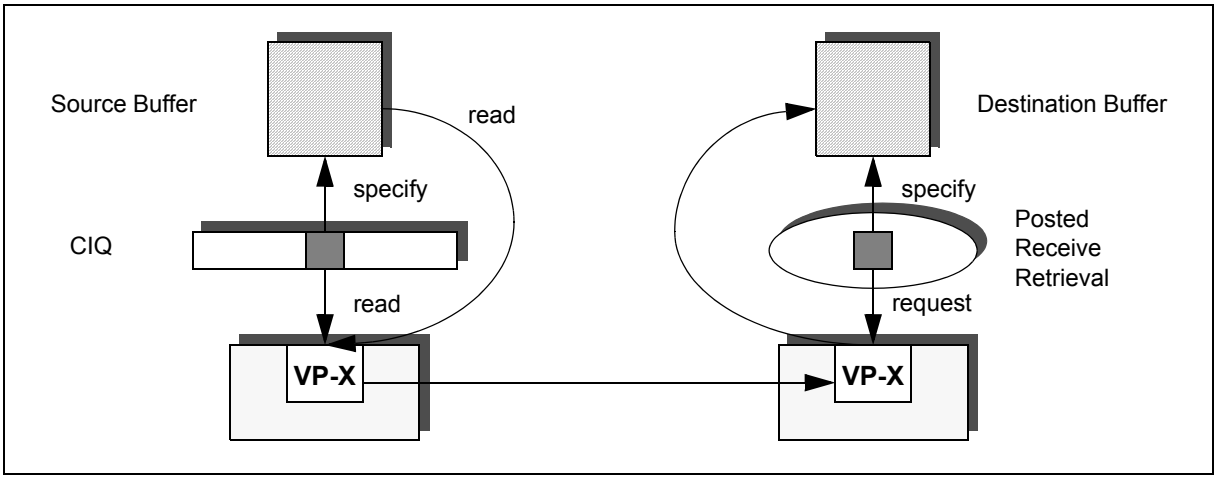


Figure 3-23 : Posted Send/Receive

In the case of posted sends, the send VCI directly specifies the data buffer in the process space (see Figure 3-23) and not an offset in the send buffer. When the send operation will be processed the Extoll NIC will read the data directly from the source buffer and sends them to the destination. There is no further need for a send buffer because no data needs to be copied into the send buffer to send a message.

On the receiver side the received message will cause the Extoll to request the corresponding receive descriptor. This descriptor specifies the destination buffer where the message should be received. After the message has been transferred into the destination buffer a notification in the corresponding NQ will be created.

Advantages

- Allows zero-copy data transfers.
- Works with virtual addresses.

Disadvantages

- Contributes to TLB cache pollution.

- Because of TLB misses the overall throughput and the latency suffers.

3.8.2.1 Posted Receive Descriptors stored in Queue

The posted receives can be stored in a physically continuous buffer in the main memory. The buffer will be managed like a ring buffer with wrap around semantic. The posted receive descriptors will be appended at the end of the queue by the user while Extoll will fetch received descriptors from the beginning. If there shouldn't be a receive descriptor available Extoll could wait until a new one is posted or create an event for the device driver.

Advantages

- Simple implementation in hardware (reuse of the RDR as posted receive queue).
- If there are receive descriptors available (should be the default case) the receive operation can be performed fast.

Disadvantages

- Do not support matching of messages an posted receives.

3.8.2.2 Posted Receives Descriptors stored in a Table

As in the event descriptor queue approach, the receive descriptors are stored in a physical memory buffer in the main memory. But instead of managing the buffer like a queue the buffer is managed as a table. Extoll can find a receive descriptor either by searching through the whole table or by a hash mechanism.

Advantages

- Supports matching.

Disadvantages

- Fixed matching mechanism in hardware.
- Searching of the device in main memory is expensive.

3.8.2.3 Posted Receives provided by Software

Instead of performing the determination of the right posted receive descriptors in hardware the determination will be done in software. Therefore if the Extoll requires a receive descriptor a corresponding event for the device driver will be created. The device driver will perform all necessary actions to determine the corresponding receive descriptor and provide it to Extoll.

Advantages

- The software solution is not fixed therefore more flexible.
- A software search will be much faster than the equivalent search done by the Extoll NIC.

Disadvantages

- High latency for every receive operation.

3.8.2.4 Posted Send/Receive without Matching

In the case that explicit matching is performed the messages that arrive will be matched in any order that depends on the receive descriptor delivery mechanism. In the case of a receive descriptor queue the receive descriptors will be matched in the order as they have been posted.

Advantages

- Simple realisation.
- No overhead for matching.

Disadvantages

- Because there is no matching it is not possible to receive the messages directly into the corresponding receive buffers.

3.8.2.5 Posted Send/Receive with Matching in Hardware

If the Extoll would support matching of arrived messages with posted receives in hardware, the Extoll NIC would be responsible for determination of the right receive descriptor. If the corresponding receive descriptor cannot be determined the Extoll NIC would generate an event to inform the device driver. The device drive would receive the message in a temporary buffer.

Advantages

- The support of matching would allow a real zero-copy implementation.

Disadvantages

- Searching in hardware is expensive.
- The management/search of the receive descriptors in hardware is fixed.

3.8.2.6 Posted Send/Receive with Matching in Software

The search of receive descriptors that are located in the main memory can be done cheaper in software. Therefore when a message arrives, the Extoll NIC would generate an event to query the device driver for the corresponding receive operation.

Advantages

- The matching in software is very flexible and can be exchanged.
- The matching in software is faster than in hardware.

Disadvantages

- For every incoming message an event (and possibly an event signal) must be created.

3.8.2.7 Posted Send/Receive with Virtual Addresses

The use of virtual addresses in the send/receive descriptors has the following advantages/disadvantages:

Advantages

- Available in user- and kernel-space.

Disadvantages

- The virtual address needs to be translated into a physical address. This would lead to a bigger cache pollution.
- Requires that send- and receive-buffers are inside a memory window because of the TLB design([FRANGER04],[REMBOR06]).

3.8.2.8 Posted Send/Receive with physical addresses

The use of physical addresses in the send/receive descriptors has the following advantages/disadvantages:

Advantages

- Fast because no address translation is required.

Disadvantages

- Physical addresses are available in kernel space.
- If an ordinary user-level process would be able to specify any physical address this would endanger the security of the system.

3.8.3 Send/Receive Emulation per RDMA

Instead of realising the send/receive mechanism (Figure 3-22) explicit in hardware it is possible to emulate this system with RDMA operations. In the case of the send/receive emulation via RDMA operations a send-/receive-buffer needs to be allocated and a corresponding memory window needs to be created for each connection.

Advantages

- Does not require explicit hardware support for a send/receive mechanism.

Disadvantages

- The RDMA operations contribute to the TLB cache pollution.
- This approach does not scale very well because the send-/receive-buffers are allocated per connection.

3.8.4 Conclusion: Send/Receive Systems

As shown all presented solutions have their advantages and drawbacks. Because the TLB cache is a very critical resource in the Extoll design the send/receive mechanism via ring buffers will be supported because this mechanism will not affect the TLB cache.

To avoid the unnecessary copies of the data the Extoll will also support the mechanism of posted send/receive. For performance reasons the Extoll will use the posted receive descriptor queue approach. Because of the posted receive descriptor queue approach and for performance reasons the Extoll will support only the simple matching approach where the incoming messages are matched to the next available posted receive descriptor in the receive descriptor queue. In the case of Extoll there is no special queue for posted sends. The posted sends are represented as a special VCI. The posted receive descriptors are stored in the posted receive descriptor queue. To avoid additional entries in the VPD the ring buffer of the receive queue will be used as posted receive queue. Therefore it is not possible to use both communication mechanisms simultaneously with the same VP. Because of the design of the TLB cache all the send/receive buffers that are posted need to be covered by a memory window. To avoid the creation of a memory window for every piece of data a global memory window that covers the whole accessible address space of the process could be created. Because all data of the process will be inside this global window the user will not need to create an extra window for all posted buffers. The emulation of the send/receive system is automatically supported by the Extoll.

3.8.4.1 Posted Receive Descriptor Format

Figure 3-24 : Posted Receive Descriptor Format

The posted receive descriptor is shown in Figure 3-24. In Table 3-11 all members of the receive descriptor are described.

Name	*Size*[a]	*Description*
Address	64	This field contains the virtual/physical base address of the buffer.
Length	23	This field describes the length of the buffer in bytes.
Window ID	16	The window that enclose the send/receive buffer. In the case that the posted receives work with physical addresses the window ID must be set to 0xffff.
VA/PA	1	This bits indicate if the address should be interpreted as physical address (bit set) or as virtual address (bit not set). The use of physical addresses is only permitted when the corresponding bit in the VPD is set (see See “Virtual Port Descriptor” on page 26).

Table 3-11: Posted receive descriptor members

a. Size in bits.

3.9 Extoll Caches

For performance reasons the Extoll uses several different kind of caches. The Extoll has the following 4 caches which will be described in this section:

- Routing Cache
- Window Descriptor Cache
- Context Cache
- Translation Lookaside Buffer

For more details about the Extoll cache system refer to the diploma thesis of Felix Rembor ([REMBOR06]).

3.9.1 Cache Management

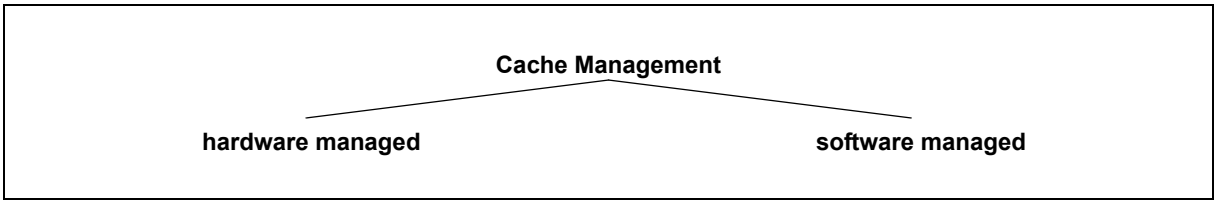


Figure 3-25 : Cache Management

The cache management decides which entries in a cache will be replaced when a new entry needs to be inserted into the cache. As shown in Figure 3-25 two possibilities to manage a cache exists: either by software or by hardware (some cache types like a direct mapped cache are implicit managed by the hardware and a software management does not make any sense). Each approach has its advantages and disadvantages.

The advantage of a hardware management is the performance. Because of the very fast access to the cache, the hardware management can use LRU information about the entries efficiently. The disadvantage is that the hardware management is fixed.

A software management has the advantage that the implementation is variable and can be exchanged. In general the software has much more information about the processes and the host-system and therefore it is possible to make more sophisticated optimizations. The biggest disadvantage of a software management is the bigger latency for accessing the cache.

3.9.2 RC - Routing Cache

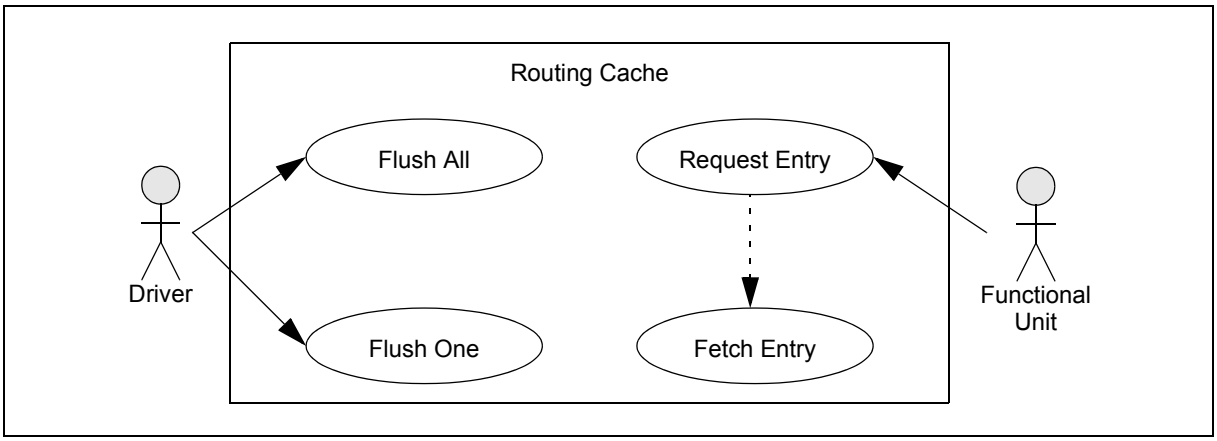


Figure 3-26 : Routing Cache Use Case Diagram

The routing cache is responsible for caching the routing strings which are used to send data through the network. While all the other Extoll caches are caching objects with a fixed size the routing cache needs to cache routing strings with variable lengths. The whole routing information is stored in a physically continuous memory block in the main memory. Each VCI contains the length and the offset of the corresponding routing string inside the routing area. As shown in Figure 3-26 the software needs the following functionality:

- **Flush One.** The flush one operation removes one specific routing string from the cache. The entry will be specified by the tupel of length and offset. This function must be exclusively accessible for the device driver.
- **Flush All**. The flush all operation resets the whole routing cache. This function must be exclusively accessible for the device driver.

The routing strings are globally shared between all VPs. All operations must create a corresponding event when the requested operation have been performed.

3.9.3 WDC - Window Descriptor Cache

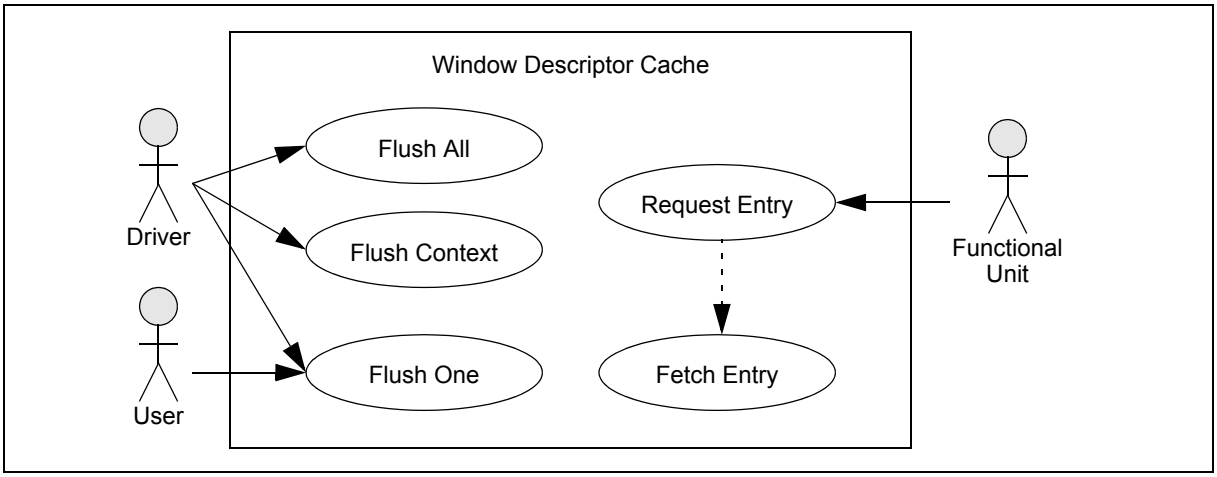


Figure 3-27 : Window Descriptor Cache Use Case Diagram

The window descriptor cache is responsible for the caching of the window descriptors. The window descriptor is a fixed size object. All memory objects are sequentially stored in a physically continuous buffer in the main memory. As shown in Figure 3-27 the software requires the following functionality:

- **Flush One**. The flush one operation invalidates one specific entry in the window descriptor cache. The entry is specified by the VPID and the window descriptor Id. This functionality must be accessible to the user and the device driver.
- **Flush VPID**. The flush VPID function invalidates all entries that belong to the specified VPID. The function requires the VPID of the entries that should be invalidated. This functionality must be exclusively accessible to the device driver.
- **Flush All**. The flush all function invalidates all entries in the window descriptor cache. This functionality must be exclusively accessible to the device driver.

3.9.4 CC - Context Cache

The context cache is responsible for the caching of VPDs. A VPD is a fixed size object. The context cache must allow the hardware to modify some parts of the VPD (e.g. read/write pointer). This leads to a synchronisation problem because some parts of the VP descriptor are managed in software while some others are managed by hardware. This problem arises if the software updates the VP descriptor in the memory while in the same time the Extoll modifies

the corresponding entry in the hardware. This problem can be solved by performing all modify operations in the hardware. If the software needs to update an entry in the VPD (e.g. pointer update) a corresponding VCI will be posted.

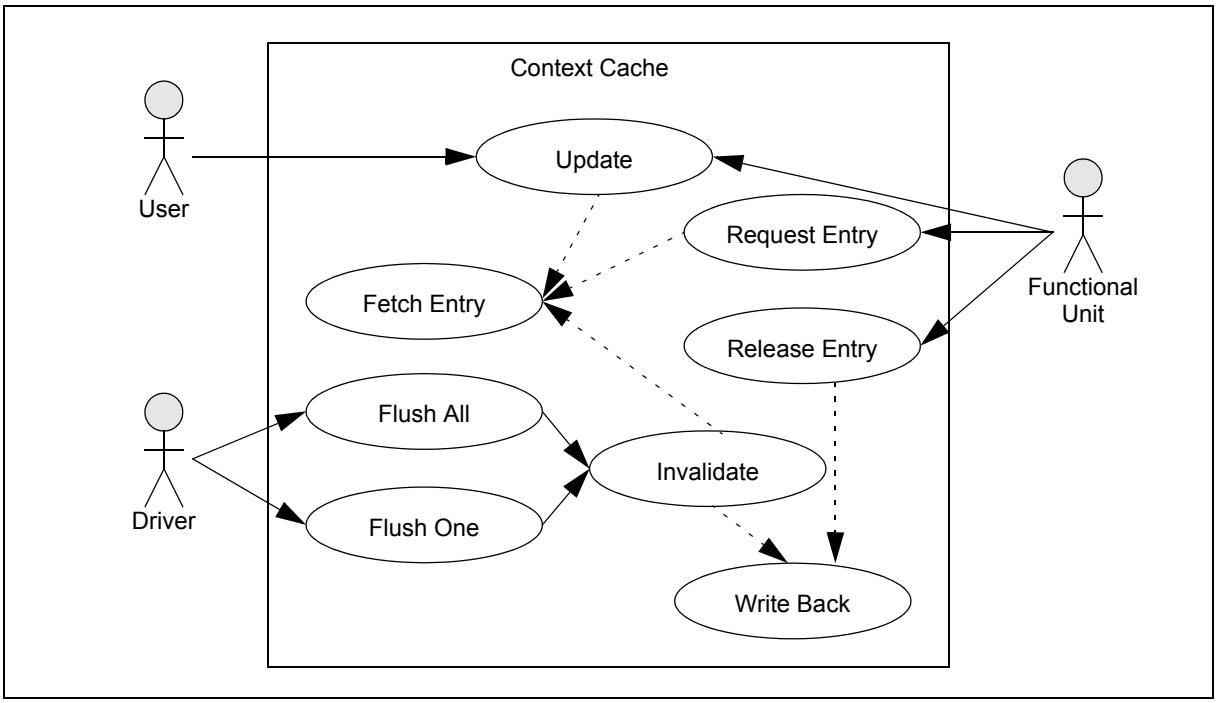


Figure 3-28 : Context Cache Use Case Diagram

As shown in Figure 3-28 the software layer requires the following functionality:

- **Flush One**. This flush-one operation invalidates a specific entry in the cache. The entry is specified by the VPID. The cache will generate an event when the corresponding entry in the cache has been invalidated or if there was not a matching entry. This functionality must be exclusively accessible to the device driver.
- **Update**. The update operation modifies a VPD. The update operation can be requested either by a functional unit or by a user request via a VCI. The entry is specified by a VPID. This functionality must be exclusively accessible to the device driver.
- **Flush All**. The flush all operation invalidates all entries in the context cache. This functionality must be exclusively accessible to the device driver.

3.9.5 TLB - Translation Lookaside Buffer

The translation lookaside buffer is responsible for caching address translations from virtual to physical addresses. The TLB stores address translations for memory regions that are described by the base address and a length and not single pages. The size of memory regions must be a multiple of 4k with a power of 2 and must be aligned to 2^x*4k. This is necessary because Extoll will run in different environments with different page sizes. Therefore it is not possible to build the TLB for a fixed page size. The next advantage is that if the address translation of a virtual area maps to physical continuous pages it is possible to represent this by only one entry in the TLB. This will lead to a significant reduction of the cache entries. Because the address translation is specific for every process the corresponding VPID needs to be stored for each entry.

Additionally, the correct permission flags of the memory pages need to be stored in the corresponding cache entries. If a virtual address range maps to physically continuous pages with different permission than it is not longer possible to use only one entry for the whole mapping. In this case the mapping needs to be splitted into smaller mappings that have the same permissions. These extra permissions are necessary because the permission flags of a memory windows are not enough to guarantee the security of the system. Actually this would lead to a security whole that would offer administration rights to every user that is able to communicate via the Extoll.

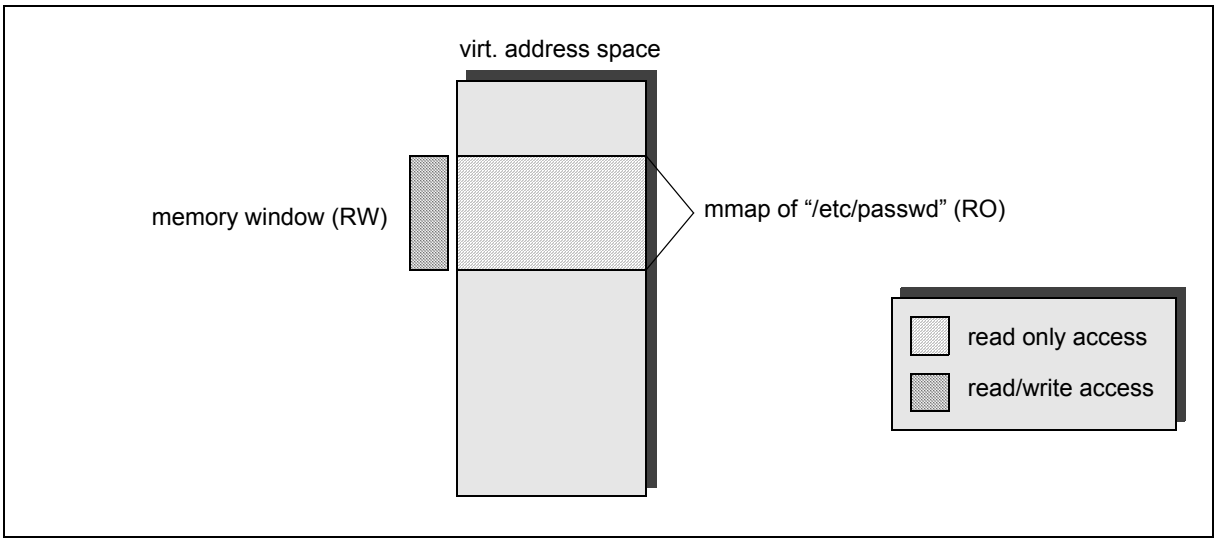


Figure 3-29 : Memory Window Exploit

Figure 3-29 shows one possible exploitation of a permission check that is only based on the permission of the memory window. The process could memory map a file for which he has only read permission (e.g. “/etc/passwd“). Than the process would create a memory window that covers the whole memory mapping of the file with read/write permissions. Now the process could modify the content of the memory mapped file by using put operations on the memory window. Without the check of the page permissions the Extoll would overwrite the content of the memory mapped file. Therefore it is necessary to check also for the current page permissions.

Everytime a function unit requires an address translation a corresponding request is sent to the TLB (Figure 3-30 (a)). If the TLB has a cache miss the software needs to be activated to provide the corresponding address translation (Figure 3-30 (b)). There are situations when the software layer cannot provide a valid address translation, e.g. if the address translation does not exits in the address space of the process or the address belongs to a prohibited area (e.g. a user process tries to access the kernel space). In this case the software layer will provide a special address translation to the TLB (Figure 3-30 (c)). This address translation will be map all invalid addresses to a unique magic value (paddr = -1, length = -1). This magic value means that there does not exist a valid address translation. After the software transferred this special translation to the TLB, the TLB interprets this special translation as an error and returns an error to the function unit that requested the address translation (Figure 3-30 (d)).

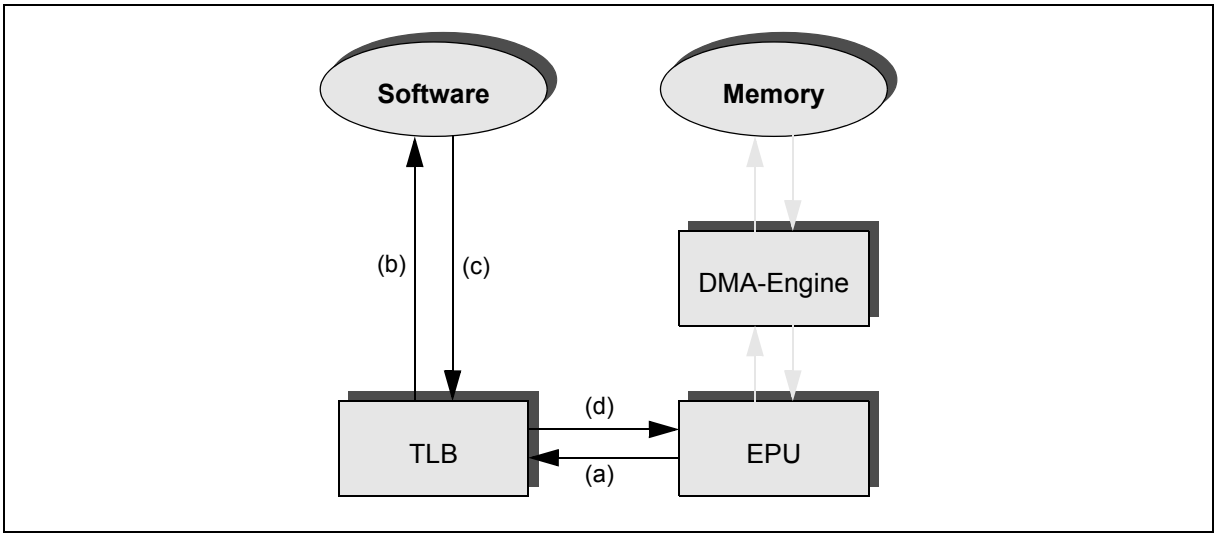


Figure 3-30 : Cache miss with direct error delivery

The functional unit will detect the error and will gracefully finish the operation with an error. Gracefully means that if the error happens at the first address translation of the operation the whole operation is completely discarded and a corresponding notification with an error is generated. If the operation has already been started the functional unit is responsible that the operation finishes gracefully and does not cause any problems on the local or the remote side.

A problem that is not directly related to the TLB is the problem of the posted receive that uses physical addresses. Because the user directly provides the physical address in the receive descriptor the TLB is bypassed. In this case the user can provide any physical address. If the specified address has no representation in the system this may lead to a problem. Therefore Extoll needs a mechanism to handle this kind of invalid addresses. This problem could be solved by introducing a lower- and an upper-memory-bound for physical addresses. Only operations that transfer data between the lower- and the upper-memory-bound are legal. If an operation violates one of the bounds, the functional unit is responsible to gracefully shutdown the operation. This limitation would also offer the possibility to support a basic form of virtualisation. The Extoll could be limited to operate only on the memory of a guest operating system (see Figure 3-31).

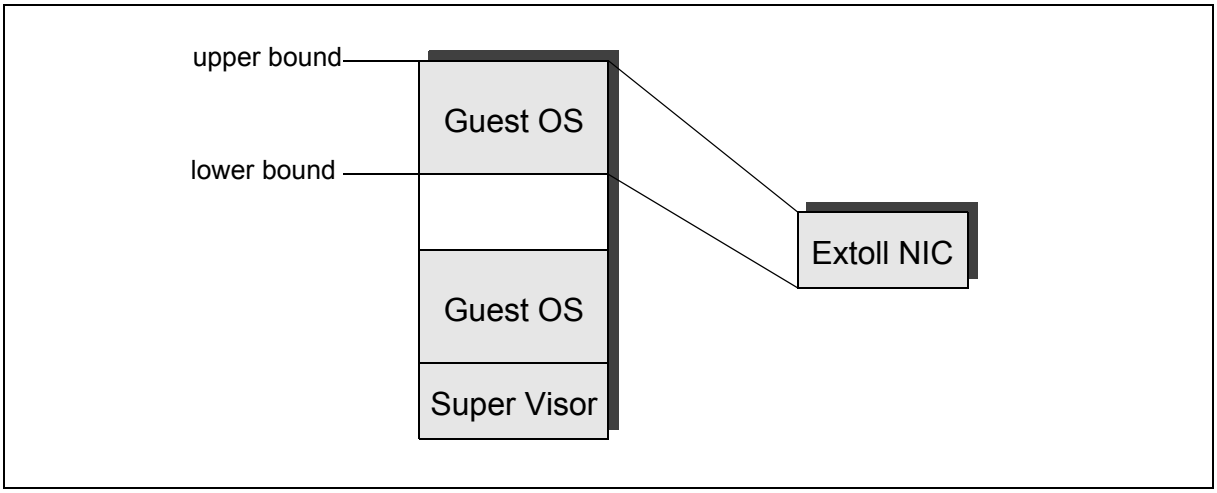


Figure 3-31 : Virtualisation Support

As shown in Figure 3-32 the software requires the following functionality:

- **Flush One**. The flush one function invalidates one specific cache entry. The entry is specified by the VPID, the window ID and the virtual address. After the entry has been invalidated by the cache a corresponding event will be generated. This functionality must be exclusively accessible to the device driver.
- **Flush Context**. The flush context function invalidates all entries that belong to a certain context. The software provides the context of the entries. This functionality must be exclusively accessible to the device driver.
- **Flush All**. The flush all function invalidates all entries of the cache. After the whole cache has been invalidated an event will be generated. This functionality must be exclusively accessible to the device driver.
- **Provide Entry**. The provide entry function transfers an address translation to the cache. The software provides the virtual address, physical address, the context and some management/status information. The functionality must be exclusively accessible to the device driver.

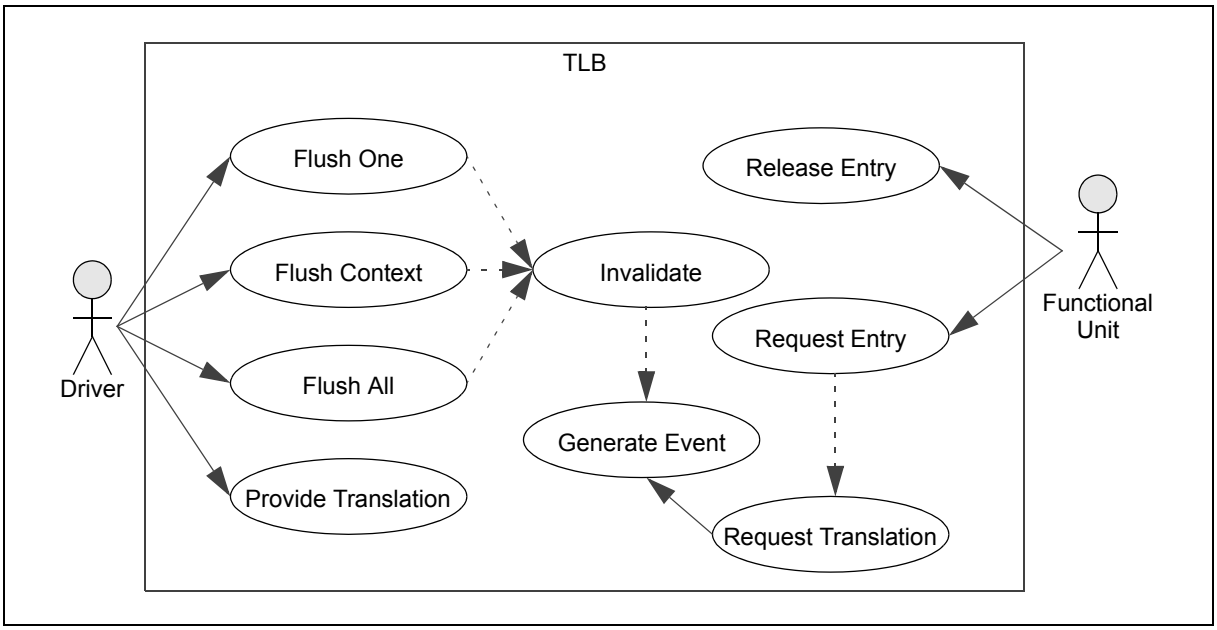


Figure 3-32 : TLB Use Case Diagram

3.10 The Barrier

Definition 3-5 : A **barrier** is a collective operation across a set of processes. The barrier call blocks all processes that participate on the barrier until all processes have entered the barrier. This means that there must be one point in time where all processes are inside the barrier.

The synchronisation of processes across a cluster is a very common task. In general the cluster interconnects does not support any special functionality to perform such a synchronisation. Therefore the barrier functionality is emulated on top of the message passing system.

The Extoll SAN will have support for a barrier synchronisation directly in hardware. The following description gives a short overview of the barrier mechanism if a single barrier. Extoll will support a support a small (about 8) amount of barriers. For a deeper description of the barrier functionality refer to the Diploma thesis of Richard Sohnis ([SOH05]).

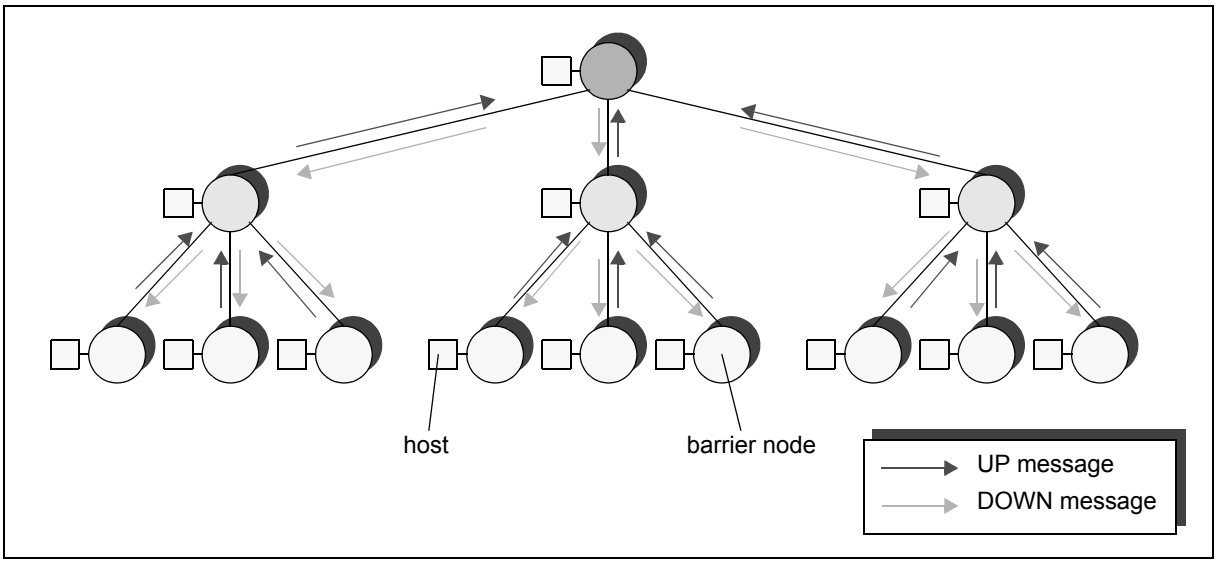


Figure 3-33 : Extoll Barrier Tree

The Extoll barrier uses a tree based approach (see Figure 3-33) where the synchronisation is performed in the root node of the barrier tree. A barrier node (see Figure 3-34) is a special purpose unit in each Extoll NIC. As shown in Figure 3-33 and Figure 3-34 each barrier node has one uplink to his parent and several down links to his children. A child of a barrier node can be either another direct connected Extoll NIC or host system of the current Extoll NIC. A barrier node always has a host system but does not need to have a parent (root node) or any other child than the host system (leaf node).

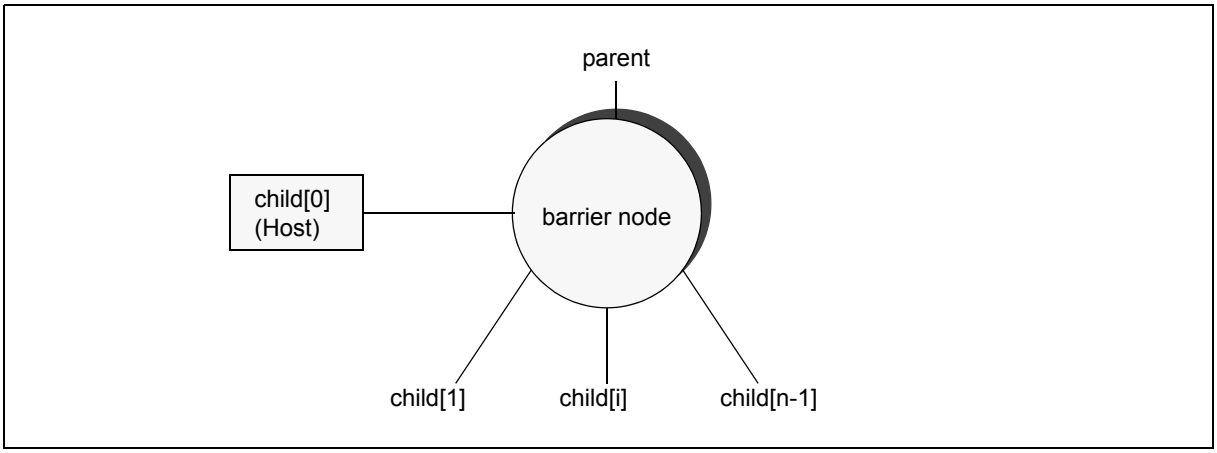


Figure 3-34 : Barrier node

Every host system that enters the barrier sends a so called UP message to the root barrier node. After a barrier node received the UP messages from all children one UP message will be sent to the corresponding parent barrier node. After the root barrier node received the UP messages

from all his children the root barrier node will send a so called DOWN message to each child to signal that the barrier is released. Every barrier node will forward this DOWN message to all of his children.

If every host system in the tree would send the UP message directly to the root barrier node or the root barrier node would send a DOWN message to every host system, the produced traffic on the root barrier node would become too high. Therefore the UP messages are collected in every barrier node and only after all children have signalled that they entered the barrier, one collective UP message is sent to the parent barrier node. In the case of the DOWN messages a broadcast mechanism is used instead of sending a DOWN message to every host system.

An important constraint is that the barrier tree is a global tree across all nodes in the cluster. This means that if some host systems does not participate in the barrier the corresponding host system must enable a dummy mechanism that always enters a barrier or the corresponding barrier node would wait forever for all UP messages. A barrier node has only one connection to the host system and therefore only one process can participate on the barrier. If several processes on the same system want to participate on the same barrier they have to first synchronise on the host with the process that has access to the barrier. After the local synchronisation with the process this process will send an UP message to the barrier node and wait for the corresponding DOWN message. After the down message arrived the other local processes will be informed.

3.10.1 Design of the Barrier Software Interface

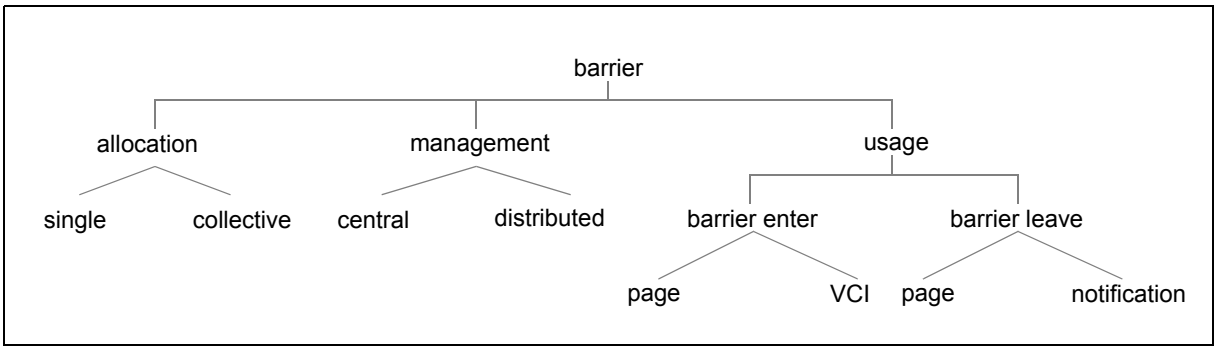


Figure 3-35 : Barrier decision tree

3.10.1.1 Single Barrier Allocation

A single barrier allocation means that if a group of processes needs to allocate a barrier, only one process requests a barrier by the local Extoll Daemon. The allocation request contains information about all processes that participate in the barrier. The Extoll daemon will perform all necessary actions to allocate and setup a barrier tree and return the corresponding barrier ID. The process that requested the barrier will distribute the returned barrier ID to the other processes in the process group. Then all processes will open the received barrier on the corresponding hosts. The release of a barrier works inversely to the allocation. First all processes close the barrier. Then the processes that requested the barrier must release the barrier by sending a release request to the daemon.

Advantages

- The user can optimize the exchange of the information.

Disadvantages

- First information about all processes needs to be collected and then the barrier ID needs to be distributed.
- Too many tasks need to be done by the user.

3.10.1.2 Collective Barrier Allocation

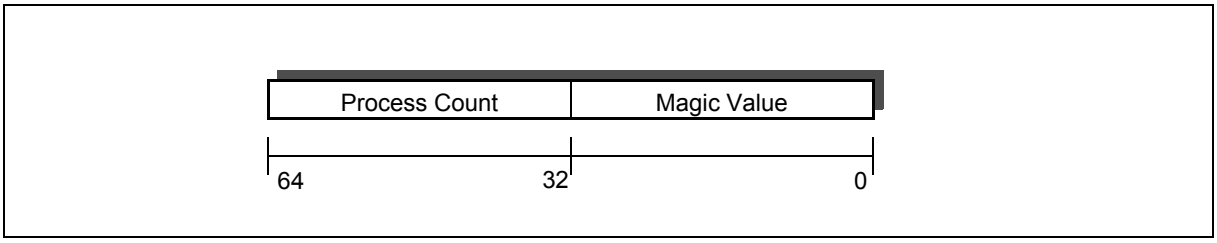


Figure 3-36 : Barrier Allocation Cookie

In the case of a collective barrier allocation all processes in a process group request simultaneously a barrier by the responsible Extoll daemon. To match the different request issued across the network all processes that participate in the same barrier need to pass the same cookie with the barrier request. The layout of the cookie is shown in Figure 3-36. The magic value is used to match the request that have been issued across the network. The process count is necessary to detect when all processes have sent the allocation request. The information about which hosts are participating in the barrier and which hosts are not participating can easily be determined by the received allocation request messages. Each node that has send an allocation request is participating in the barrier and the rest is not. Before the barrier allocation requests of the processes returns the allocated barrier could automatically be opened. The release of a barrier works in the same way. All processes of a barrier send a barrier release request to their corresponding Extoll daemon. This release request will close the barrier and release the barrier tree.

Advantages

- Simple interface for the user.
- The internal management of the barriers could be optimized.

Disadvantages

- The used magic number for a process group must be unique.

3.10.1.3 Central Barrier Management

In the case of a central managed barrier there exist one distinct resource that is responsible for the barrier assignment.

Advantages

- Simple to realize.

Disadvantages

- Single point of failure.
- Despite the central barrier arbitration the corresponding barrier units on each node needs to be configured.

3.10.1.4 Distributed Barrier Management

In the case of a distributed barrier management there does not exist a distinct management resource in the network. The arbitration of the barrier is made by all Extoll daemons across the network by using a distributed allocation algorithm.

Advantages

- The arbitration and the configuration of the barrier could be done in one step.

Disadvantages

- Very complex.

3.10.1.5 Barrier Enter via Memory mapped I/O Page

As mentioned above the process that is running on a host system must tell the local barrier node that the barrier has been entered. A simple way is to map a special piece of Extoll I/O memory to the process. When the process enters the barrier, the process needs to read from a certain offset of the mapped I/O space. This read operation will inform the barrier unit inside Extoll that the process entered the barrier. In the return value of the read the Extoll is able to transfer additional information (e.g. errors like barrier not ready, barrier already entered, ...).

Advantages

- The read operation is very fast and the user gets additional information.
- Faster than a VCI.

Disadvantages

- Does not fit in the VCI principle.

3.10.1.6 Barrier Enter via VCI

An alternative approach to signal that the local process has entered the barrier is by using a special VCI.

Advantages

- Fits in the general design of the Extoll.

Disadvantages

- The VCI flow would increase the latency of the barrier.

3.10.1.7 Barrier Leave via I/O mapped Memory

After the barrier has been release each barrier node needs to inform the process on the corresponding host system. One way to get informed about the release of a barrier would be to poll on I/O memory that has been mapped to the process.

Advantages

- Results in a very low latency.

Disadvantages

- Polling approach waste CPU cycles.

- Polling on the I/O device cause unnecessary traffic on the I/O bus. This would lead in a decreasement of the whole communication performance.
- Does not fit the rest of the Extoll design.

3.10.1.8 Barrier Leave via Notification

When the barrier is released the process could be informed via a notification.

Advantages

- Fit in the global Extoll design.

Disadvantages

- The creation of the notification and the generation/signalling of the corresponding event would increase the latency too much.

3.10.1.9 Conclusion : Extoll Barrier

The Extoll barrier will use a collective allocation operation. As shown the collective barrier allocation mechanism will simplify the usage for the user and allows the management system to implement some optimizations which are transparent to the user.

The management of the barrier will be done by a central unit because the amount of available barriers is very limited. Therefore the overhead of a distributed allocation mechanism is not justified and would possibly increase the complexity and decrease the performance.

Because the main goal of the barrier is to achieve a very fast synchronisation primitive the approaches to enter and leave the via a mapped I/O memory will be used. The other approaches with the notification and the VCI would fit better to the global design of the Extoll but are too inefficient.

3.10.2 Extoll Barrier Usage

This section will give some suggestions how Extoll barriers could be used in practice. Therefore the following sub-sections will provide different approaches how the *MPI_Barrier* functionality could be realised with the Extoll barrier. The *MPI_Barrier* function is used to synchronise the processes that belong to the passed communicator.

3.10.2.1 Barrier Mapping 1:1

The first and simplest approach would be a 1:1 mapping of a communicator and an Extoll barrier. In this case the processes of communicator allocate a Extoll barrier which would be used for synchronisation.

Advantages

- Simple realisation.

Disadvantages

- For some cases (e.g. a communicator with 2 processes on the same node) the overhead of the Extoll barrier would be bigger than a software based synchronisation.
- There exist only few Extoll barriers for the whole network (about 16).

3.10.2.2 Barrier Mapping M:N

As shown the 1:1 mapping of the communicator and an Extoll barrier is not very practical. Therefore the next approach could be to use a M:N mapping between communicators and Extoll barriers. This means that a certain amount of communicators will be mapped to a different amount of Extoll barriers. In general the number of communicators is much bigger than the available number of Extoll barriers. Therefore the MPI program will allocate a certain amount of Extoll barriers at the start of the program. These Extoll barriers will be configured to cover all processes in the MPI program and will be managed in a special Extoll barrier pool (see Figure 3-37). Then the implementation can decide which communicators will be mapped to a barrier and which not. If a communicator is mapped to an Extoll barrier and has less processes than the whole MPI program the processes that are not in the corresponding communicator must always enter the barrier automatically.

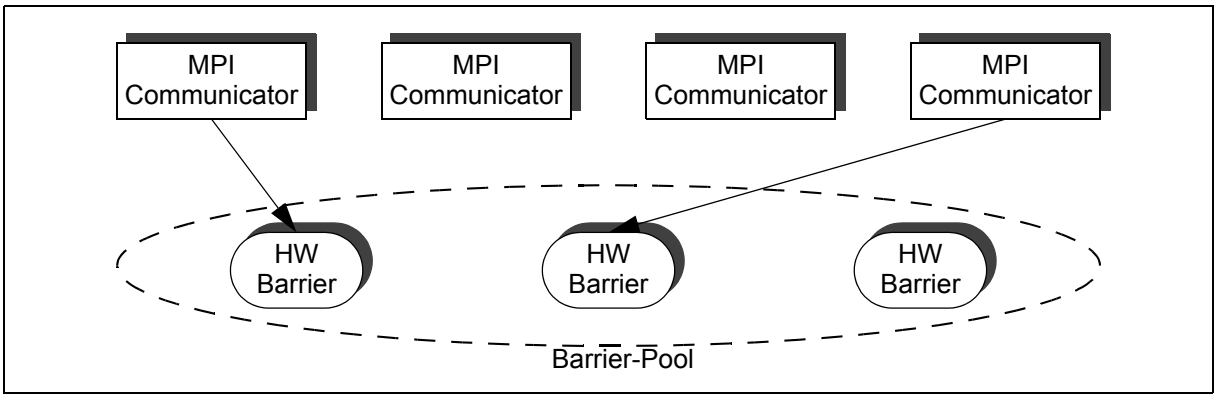


Figure 3-37 : Barrier M:N Mapping

Advantages

- Only valuable communicators are associated with an Extoll barrier.
- The resource Extoll barrier will be efficiently used.
- Only a small amount of Extoll barriers are needed per MPI program.

Disadvantages

- Does not work with dynamic process creation features of MPI-2.

3.11 The ULTRA System

ULTRA stands for “Ultra low latency transaction”. The ULTRA system is a special purpose communication system to archive the lowest possible latency. ULTRA consists of send and receive units which are called send/receive ports. The rest of this section gives an overview of the ULTRA system and the corresponding software interface. For more detailed information especially design decisions refer to the diploma thesis of Heiner Litz ([LITZ05]).

- ULTRA units are not virtualized. Therefore each hardware unit is exclusively assigned to one specific user.
- The ULTRA system only supports the transfer of messages and no RDMA operations.
- The ULTRA message size is limited to a maximum size of 64 bytes.

- The ULTRA system guaranties in-order message delivery on a virtual connection.
- Each ULTRA send port is able to send messages to each ULTRA receive port and each ULTRA receive port is able to receive messages from each ULTRA send port.
- There exists one I/O configuration space for all ULTRA units that must not be accessible to a user.
- The ULTRA system uses PIO for realisation of fast send operations.
- Ultra communication is uni-directional from a send port to a receive port (see Figure 3-38).

Figure 3-38 : ULTRA Communication System

3.11.1 ULTRA Management

The Extol NIC will support up to 8 send ports and 8 receive ports per Extoll NIC. All ULTRA ports are managed via a special I/O memory area in the Extoll configuration space. This memory is only accessible for the device driver.

3.11.2 ULTRA Send Port

One of the design goals of the ULTRA unit is to reach the lowest possible latency. With the FAST_SEND VCI the VP interface already has a low latency send operation. But the FAST_SEND operation still needs several accesses until the send operation can be performed. As shown in Table 3-12 the FAST_SEND requires 2-4 I/O bus accesses. The approach of the ULTRA system is to reduce the number of I/O bus accesses to the smallest possible amount. The big number of I/O bus accesses are caused by the virtualisation of the hostport. Because the ULTRA units are not virtualized it is possible to avoid these extra I/O bus accesses.

I/O Bus Accesses	*FAST_SEND*	*ULTRA*
best case	2[a]	1
worst case	4[b]	1

Table 3-12: Comparison of FAST_SEND and ULTRA

a. triggering of the VCI + fetching of the VCI
b. triggering of the VCI + fetching of the VPD + fetching of the VCI + fetching of the routing

The ULTRA approach requires only one I/O bus access (see Table 3-12) per message. Because there is only one I/O bus access the data needs to be transferred by this access via PIO. In general it would be possible to write FAST_SEND VCI into the ULTRA send unit. With this VCI the ULTRA send unit would have all necessary information and the data at once. The problem with the FAST_SEND VCI is that there could be a cache miss. To avoid this additional cache misses the ULTRA unit stores the routing to the destination directly in hardware.

3.11.3 ULTRA Receive Port

3.11.3.1 ULTRA Receive via PIO

A symmetric approach to the send would receive the messages also via PIO. This approach has the disadvantage that the process would need to poll on the Extoll NIC, which would lead in unnecessary I/O bus accesses.

Advantages

- Very fast way of receiving the data.

Disadvantages

- Polling in Extoll NIC lead in unnecessary I/O bus accesses.
- Data must be buffered in hardware until the user is able to consume them and buffers in hardware are always limited and relatively small compared to buffers in software.

3.11.3.2 ULTRA Receive via DMA Buffer

This approach would be to store the incoming ULTRA messages via DMA into a special message queue in the main memory. The buffer of the message must be physically continuous so that the ULTRA receive unit needs only the base address and the length of the buffer.

Advantages

- Only one I/O bus access for every received message.
- The user polls only on memory or the cache.
- Buffers in Software can have different sizes.

Disadvantages

- Slower than the PIO mode because the data needs first to be written to the main memory before the user is able to load them into the CPU.

3.11.4 Conclusion : ULTRA Receive

Because the solution with the message buffer is more flexible and only minimal slower than the PIO approach the ULTRA receive port will use the message buffer in the main memory approach.

3.12 Proposal for a new RDMA Operation

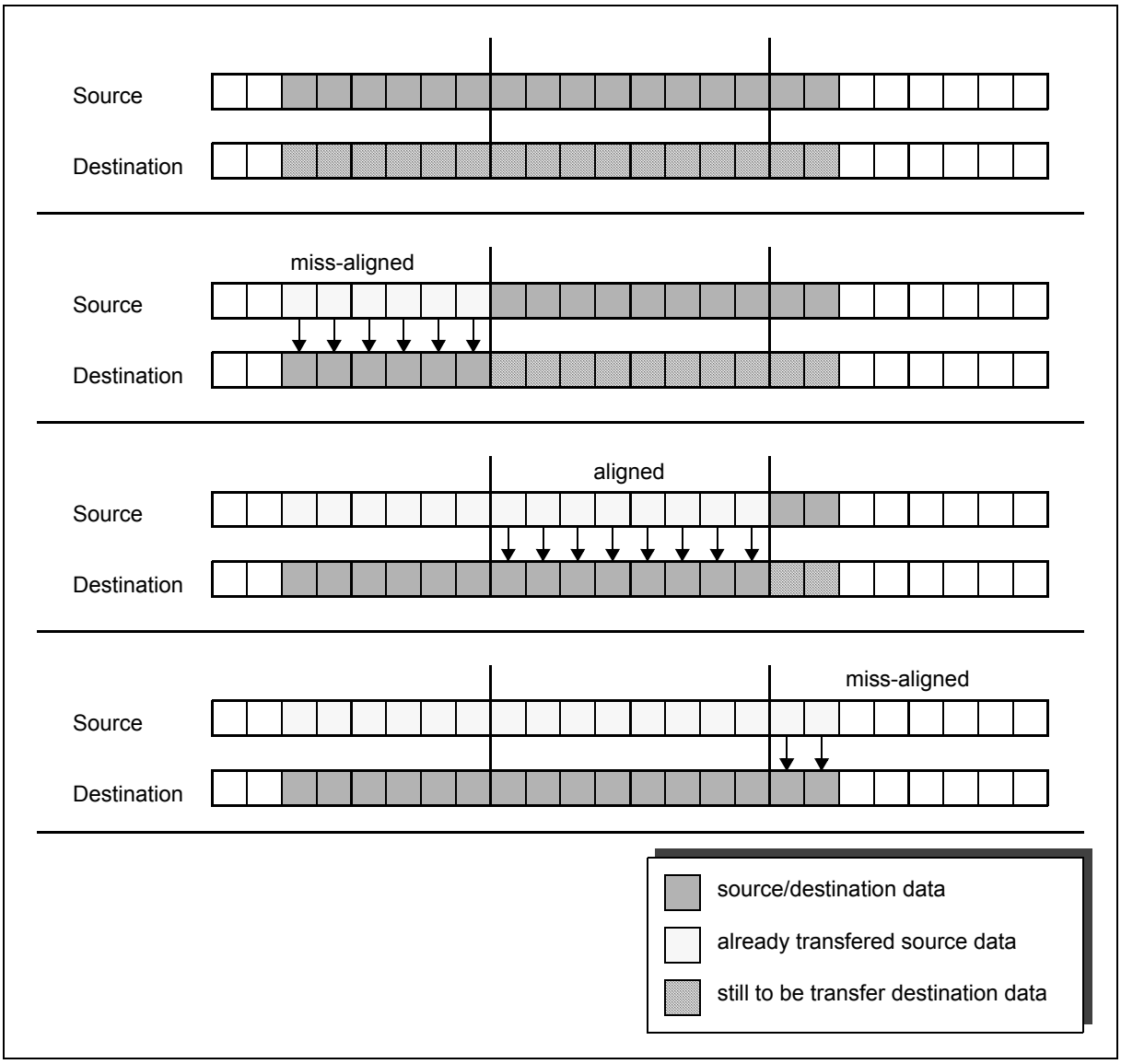


Figure 3-39 : Optimal RDMA Transfer

So far two different classes of RDMA operations have been specified ([FRANGER04]), the aligned and miss-aligned operations. The aligned RDMA operations are able to transfer any amount of data as long as the data is aligned to 8 bytes and the length is a multiple of 8 bytes. The miss-aligned RDMA operations are limited to a maximum amount of 8 bytes data which are allowed to be miss-aligned. If a bigger miss-aligned buffer needs to be transferred two different cases exists. In the first case the source and the destination buffer have the same miss-alignment offset (offset relative to the 8 byte alignment). In this case the transfer of the buffer can be splitted into a maximum of three RDMA operations, two miss-aligned and one aligned RDMA operation (see Figure 3-39). In the case that the source and the destination buffer have a different miss-alignment offset it is not longer possible to transfer the buffer with only three RDMA operations. After the first miss-aligned RDMA operation the source buffer is aligned but the destination buffer not, after the second miss-aligned operation the destination buffer is aligned but the source buffer not and so on (see Figure 3-40). This means that the buffer must

be transferred by many miss-aligned RDMA operations. The amount of required miss-aligned RDMA operations scales linearly with the buffer size. A solution for this problem would be the introduction of a new RDMA operation which supports double miss-aligned buffers.

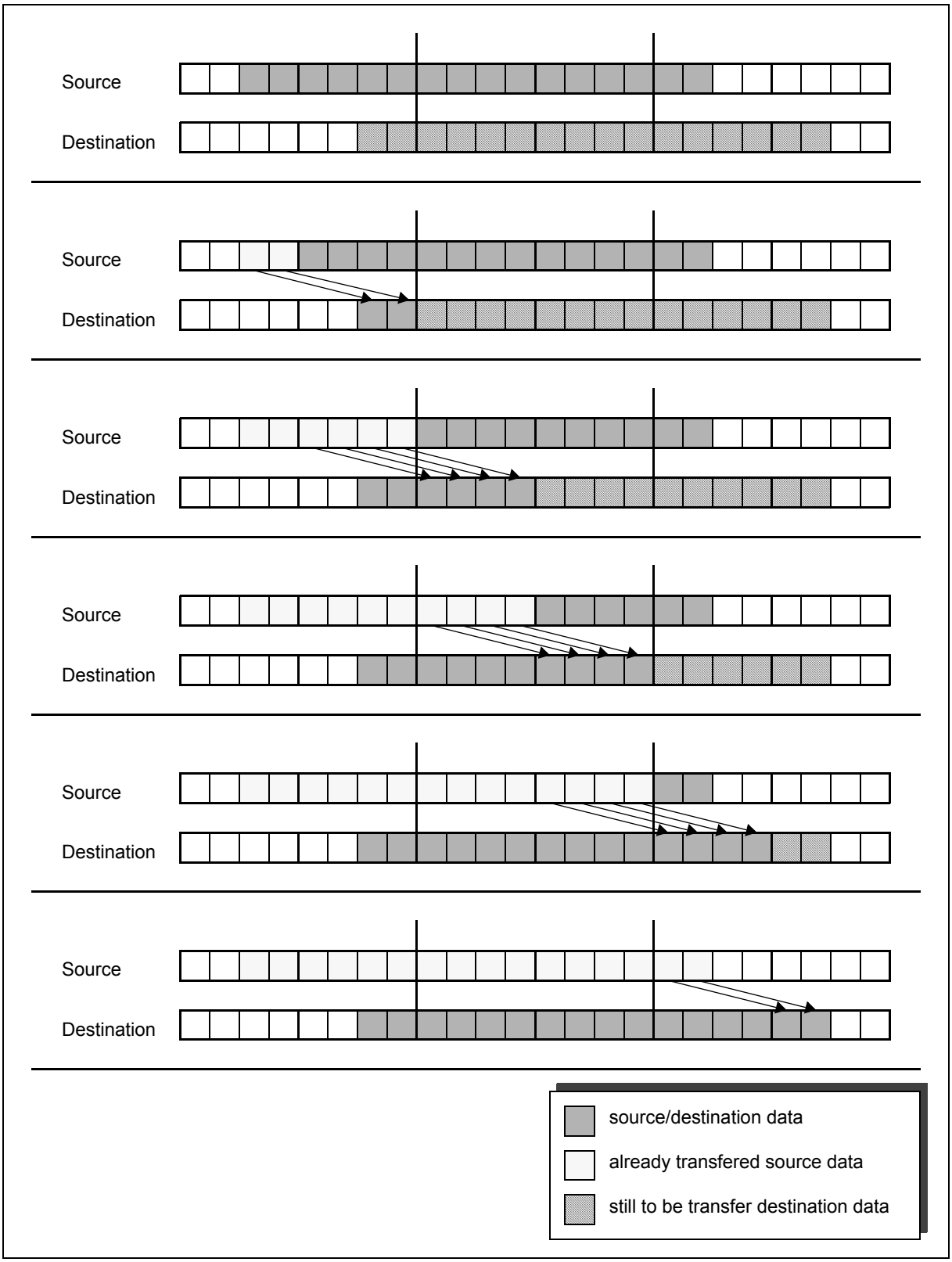


Figure 3-40 : Worst Case RDMA Transfer

Design of the ESS

CHAPTER 4

This chapter introduces the design of the Extoll Software Stack (ESS). First the global design of the ESS will be presents. The first step consists of the analysis of the requirements of the ESS. Based on the requirements the design with all design decisions will be shown.

After the global design of the ESS has been described each component will be presented in a more detailed view. As for the global design, the first step consists of a requirement analysis, followed by the presentation of the component design.

The chapter ends with discussion of features that are supposed to be handled by the Extoll Daemon.

4.1 Extoll Software Stack

4.1.1 Requirements of the ESS

This section gives an overview of the requirements of the ESS. Based on these requirements the final ESS design will be presented.

- The ESS is a software environment for a high performance and low latency RDMA NIC. Therefore ESS must add the lowest possible latency penalty to the whole communication. Especially the critical path should be optimized and free of all unnecessary operations which would increase the latency.
- The software should be efficient. Efficient means efficient in speed and resource footprint. The required resources should be as small as possible but as high as necessary to reach the maximum performance.
- The software must be robust. Robust means that the Extoll Software Stack and the host system must not crash even if some operations fail (e.g. a process dies). It also means that there will no be resources lost.
- The software must avoid any backdoor mechanism that would violate the security of the host system or the whole network.
- The software should be modular. It should be possible to replace each component of the software by a different implementation without any change in the behaviour of the whole software stack.
- The Extoll Software Stack API for communication should be usable in the user- and the kernel-space with the same interface.
- The Extoll Software Stack should have strong debugging and monitoring support.
- The Extoll Software design should follow the object oriented paradigm as much as possible.
- The software should be portable and run on different architectures. The initial target architectures are IA32 and AMD64.

- The Extoll Software Stack must be able to handle multiple Extoll NICs in the same host system simultaneously.
- The Extoll Software Stack should work closely with currently existing operating system features (e.g. *epoll*).
- This version of the ESS is supposed to work under the Linux operating system. Because of additional platforms that may be supported in the future the ESS should be implemented as portable as possible.
- The ESS should enable and simplify the development of an ethernet emulation device driver (IPoExtoll) that tunnels IP data over the Extoll SAN. The development of the IPoExtoll is not part of this thesis.

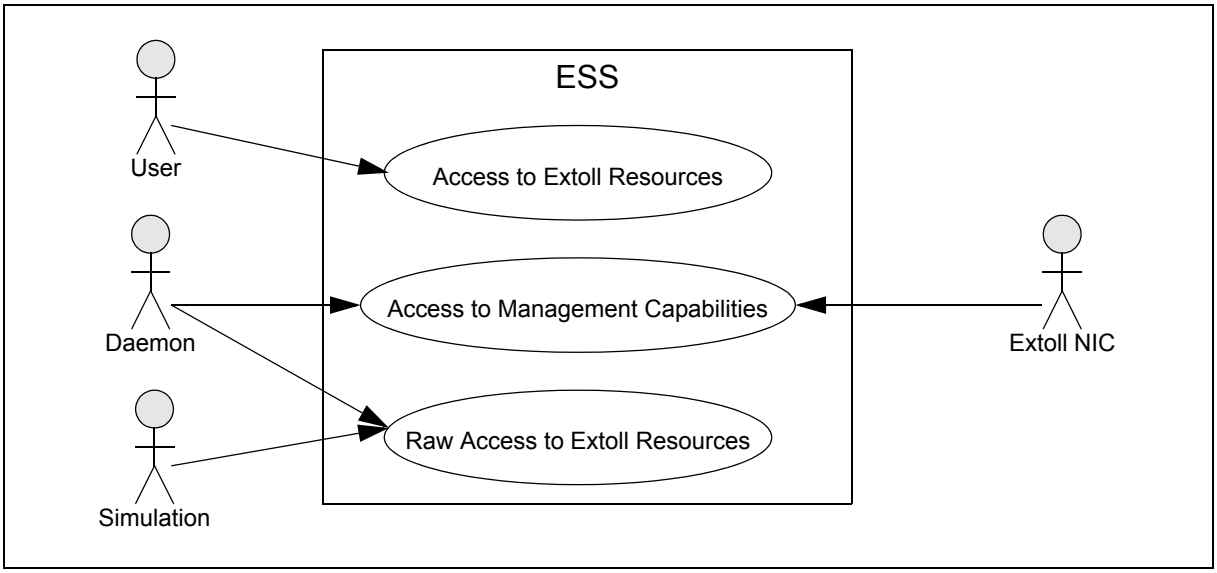


Figure 4-1 : ESS Use Case Diagram

As shown in Figure 4-1, 4 different customers of the ESS can be identified. These customers are the user, the daemon, the simulation and the Extoll NIC. The user is a process, either in user- or kernel-space, that needs access to Extoll resources for communication purposes. The daemon is a management process that runs on every host in the network. The daemon's main tasks consist of management and configuration of the Extoll NICs (See "Extoll Daemon" on page 88). To fulfil these tasks the Extoll Daemon needs raw access to the Extoll Resources and also access to the Extoll management functionality. The simulation ([SPONER05], [FELDNER04], [FRANGER04]) consumer is an ordinary user process that requires a raw access to the Extoll resources to simulate the behaviour of the Extoll NIC in software. The last consumer is the Extoll NIC itself. The Extoll NIC needs access to the management functionality of the ESS. Every time a problem is encountered by the Extoll NIC (for e.g. the Extoll NIC needs an address translation, a link broke, ...) the Extoll NIC will inform the ESS to solve the problem.

4.1.2 Design of the ESS

4.1.2.1 Routing Management

Definition 4-1 : The **routing management** covers the complete management of the routing buffer in the main memory. The main tasks of the

routing management are the insertion, deletion and the query of routing strings.

Definition 4-2 : The **routing determination** is the process of calculating the routing strings for a given topology.

The routing is stored in a physically continuous memory buffer. In the case of Atoll the routing was calculated and managed by the Atoll Daemon. If a user of the Atoll needed the routing information a corresponding request has been send to the Atoll Daemon and the daemon replied with the requested information or an error.

This approach does not work for the Extoll because there are user-level and kernel level consumers. If the management of the routing would be done by a user-level process there would be a problem when a kernel consumer needs to query routing information, because there does not exist a standard way for communication from kernel space to user space. Therefore the complete routing management must be done in kernel space by a dedicated kernel module. This routing managing module is called Extoll Routing Manager (ERM, see “Extoll Routing Manager” on page 79 for more information).

4.1.2.2 Memory Management

Almost all buffers that are associated with a VP must be physically continuous memory. The allocation of physically continuous memory during runtime can become a problem because of the fragmentation of the memory. Fragmentation is caused by usage of the paging mechanism ([TANENBAUM]). In general fragmentation increases with the uptime of a system.

The first approach of the Atoll design to overcome this problem was to truncate the physical memory that is managed and associated with the host system ([FELDNER04]). To void this truncation of memory the next approach was the introduction of the mempin module. The task of the mempin module is to be loaded after the system start when the memory fragmentation is still minimal and grab all the required physically continuous memory buffers that are necessary to satisfy the requirements of all hostports. This approach is not longer working for Extoll because the Extoll design supports up to 2^{16} VPs and not only 4 like the Atoll design. Therefore the required resources cannot be allocated at once. Because the static memory allocation approach does not work very well with Extoll a dynamic memory management system is required. The dynamic memory management system of the Extoll is called Extoll Memory Manager (EMM, see “Extoll Memory Manager” on page 77 for more informations).

4.1.2.3 Logging Support

A strong logging system is essential to keep track of the ongoing actions of the software stack. Therefore the ESS will have a strong logging mechanism. Instead of implementing the logging functionality for each ESS module separately an extra logging module is created. The logging module is called Extoll Logging System (ELS, see “Extoll Logging System” on page 76 for more information). The ELS must be working on kernel- and user space because the ESS is working on both systems.

4.1.2.4 Application Programming Interface

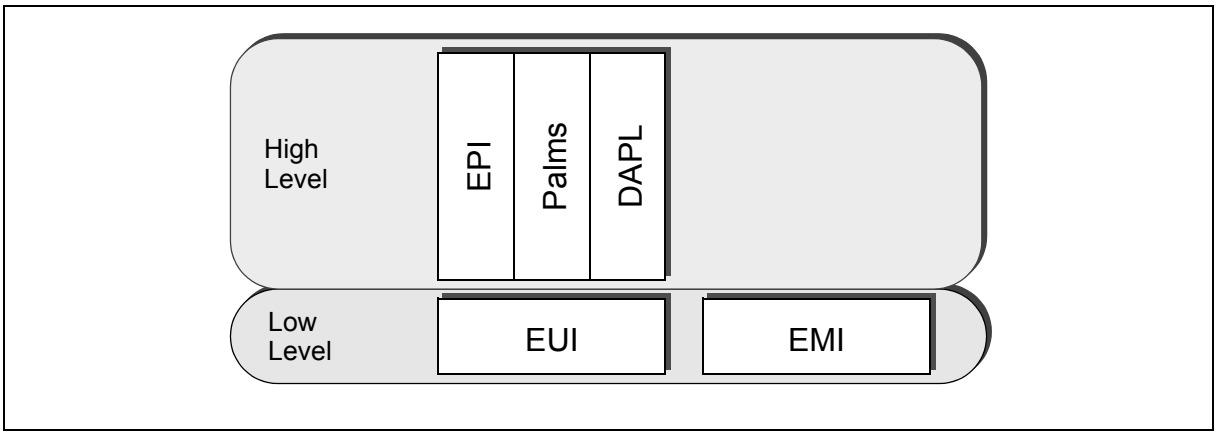


Figure 4-2 : Extoll API

The Extoll API is partitioned in 2 parts, one low level and one high level part. The low level part is a layer that abstracts from the hardware and the differences between kernel- and user-space. The low-level part of the Extoll API itself is partitioned again into two different parts, the Extoll User Interface (EUI) and the Extoll Management Interface (EMI). The EUI offers an interface for any user that wants to communicate via the Extoll (see “Extoll User Interface” on page 84 for more information). The EMI offers a special interface that allows the direct access to Extoll resources and the management functions. On top of the EUI the different API can be implemented. The standard API of the ESS is the Extoll Programming Interface (EPI). The EPI has the task to offer a high level interface to the user that simplifies the usage of the ESS (See “Extoll Programming Interface” on page 87 for more information). All components of the Extoll API must be running in kernel- and user-space.

4.1.2.5 Device Driver

The Extoll Device Driver (EDD) is the central component of the ESS. The EDD is responsible for all resource configuration and management tasks of the Extoll NICs (See “Extoll Device Driver” on page 80 for more information).

4.1.2.6 Conclusion : Design of the ESS

As shown in the previous sections the ESS consists of the following components:

- Extoll Logging System
- Extoll Memory Manager
- Extoll Routing Manager
- Extoll Device Driver
- Extoll User Interface
- Extol Management Interface
- Extoll Programming Interface

Every component has a certain task to serve. All components are independent of each other and offer a well defined interface to the other components. The arrangement of the components is shown in Figure 4-3.

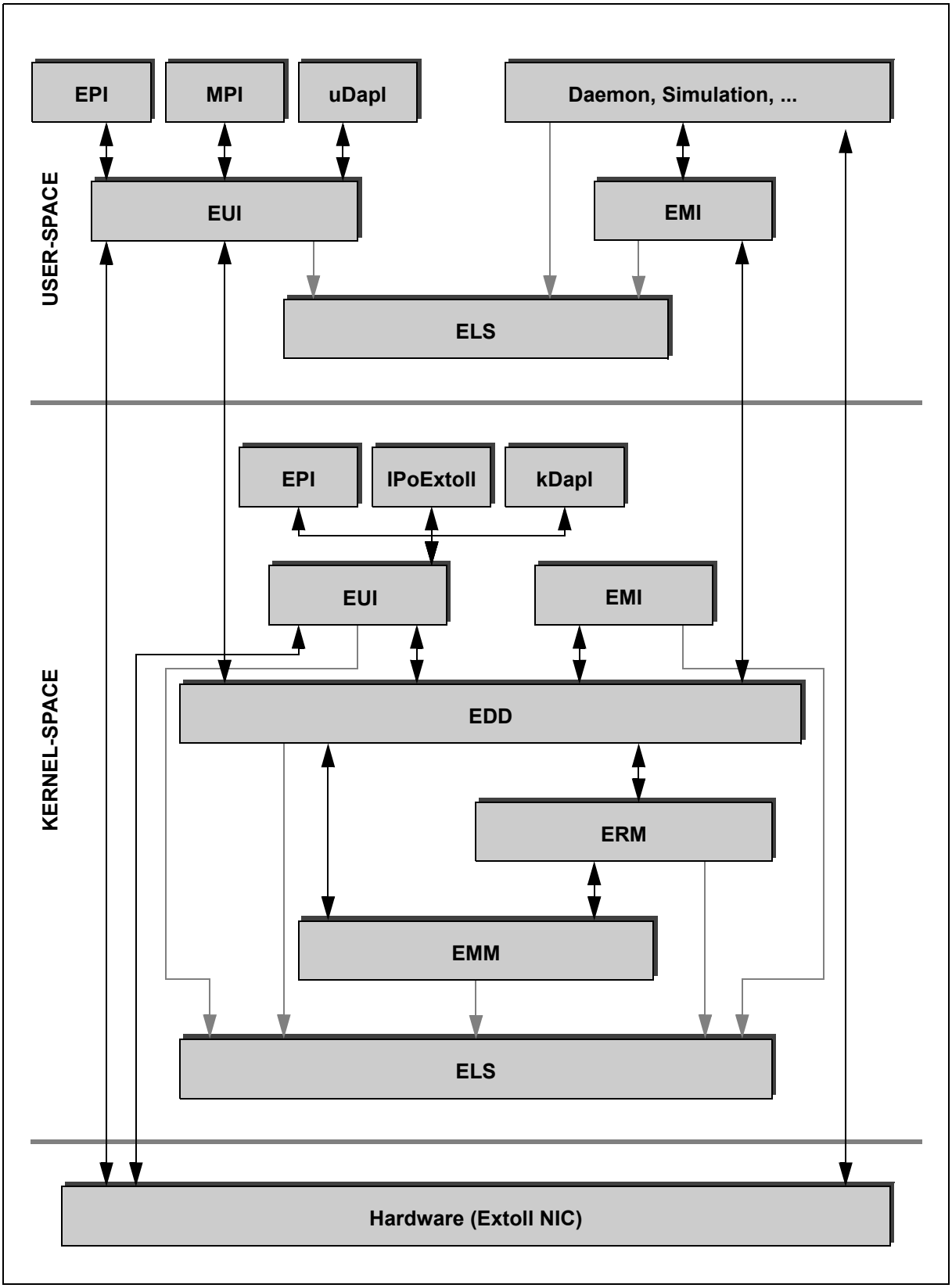


Figure 4-3 : ESS Design

4.2 Extoll Logging System

4.2.1 Requirements

The Extoll Logging System has the following requirements:

- The Extoll Logging System must be fast to avoid any performance impacts to the application or whole system.
- The Extoll Logging System must be usable in a development and a productive environment.
- The Extoll Logging System must be simple and abstract from the underlaying logging backend (e.g file, stdout).
- The Extoll Logging System must work in kernel- and user-space with the same user interface.
- The Extoll Logging System should provide logging to different mediums.

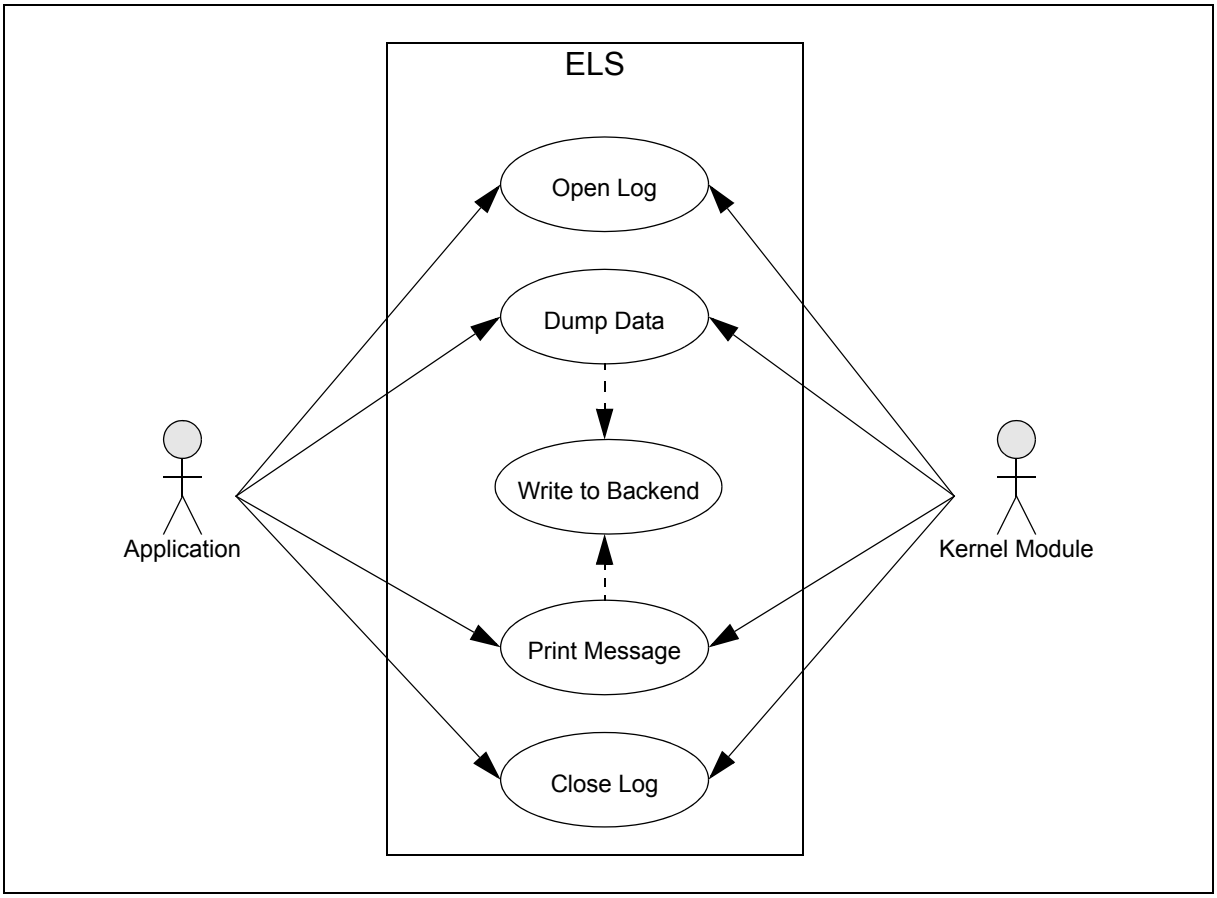


Figure 4-4 : ELS Use Case Diagram

As shown in Figure 4-4 there are two customers for the Extoll Logging System, applications that are running in user space and kernel modules that are running in kernel space. The functionality that must be offered by the Extoll Logging System are the creation and deletion of logging objects and corresponding logging functionality.

4.2.2 Design

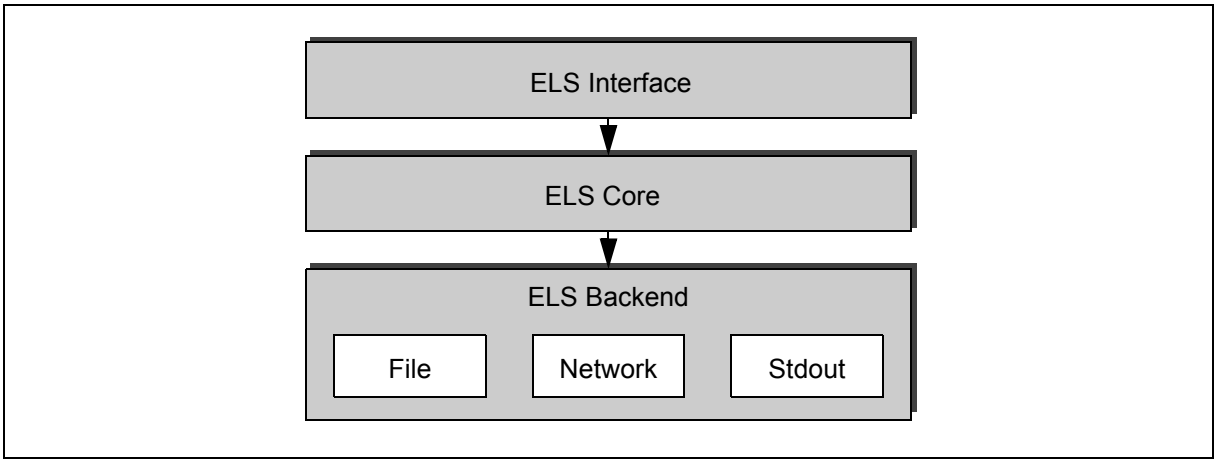


Figure 4-5 : ELS Design

The ELS component consists of the following modules:

- **ELS Interface**. The ELS interface consists of the functions that are offered to the customers.
- **ELS Core**. The ELS core is responsible for the pre-processing of the log information, before they are transferred to the backend.
- **ELS Backend**. The ELS backend is responsible to transfer the log data to the log medium.

4.3 Extoll Memory Manager

4.3.1 Requirements

The Extoll Memory Manager has the following requirements:

- The EMM must guarantee enough memory for at least a certain amount of VPs.
- The EMM should be able to support different usage patterns (e.g. a few VPs are opened/ closed very infrequently versus a lot of VPs are opened and closed very often).
- The EMM should work well with the Linux memory sub-system.
- Need only run in kernel mode.
- The memory that is returned by the EMM must be physical continuous.
- The EMM should be able to support different allocation mechanisms.

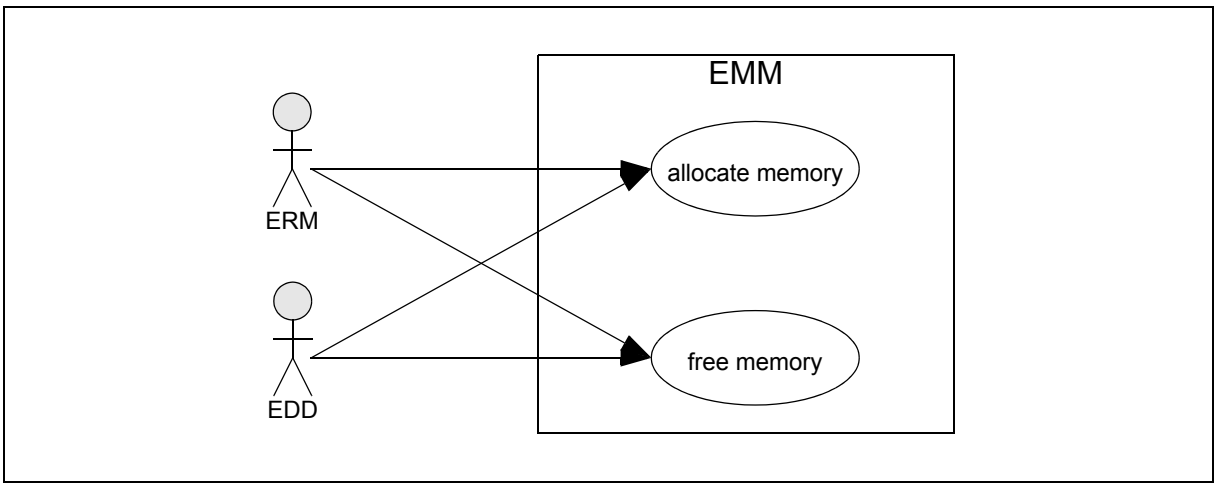


Figure 4-6 : EMM Use Case Diagram

As shown in Figure 4-6 two customers exists for the EMM, the ERM (See "Extoll Routing Manager" on page 79) and EDD (See "Extoll Device Driver" on page 80). The functionality that must be offered by the EMM is the allocation and the release of memory blocks.

4.3.2 Design

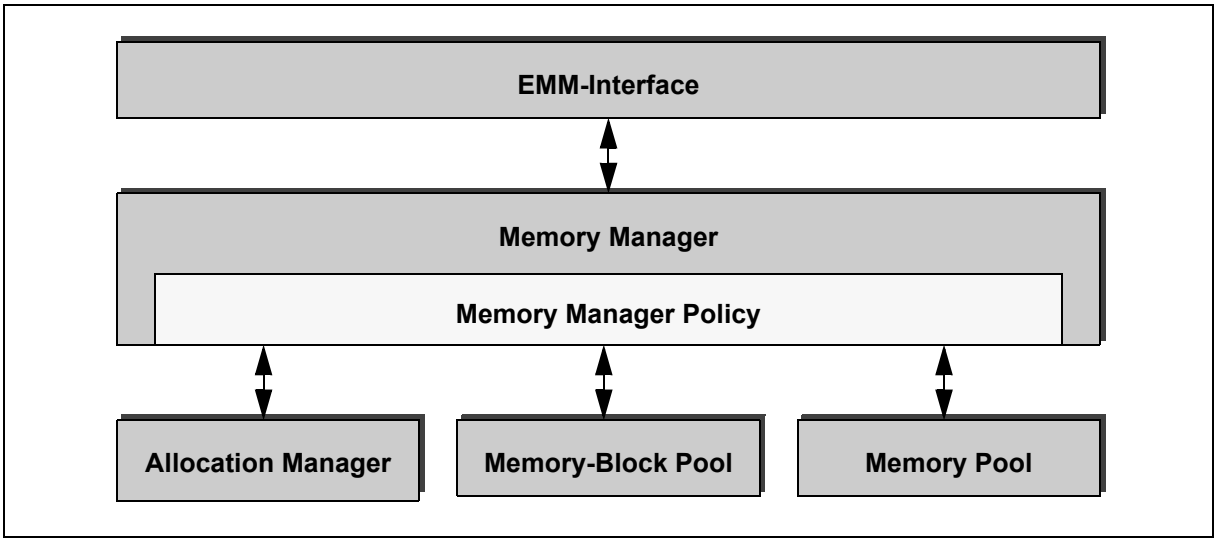


Figure 4-7 : EMM Design

Definition 4-3 : **Extoll Memory Block Object**. An Extoll Memory Block Object is an abstract software object that represent a memory region. The memory region can be either RAM or I/O mapped device memory.

As shown in Figure 4-7 the EMM consists of the following components:

- **EMM Interface**. The EMM interface consists of the functions that are offered to the customers of the EMM.

- **Memory Manager**. The Memory Manager is the core of the EMM, that is responsible for managing the memory requests of the customers. The way in which the memory is managed is defined by the Memory Manager Policy (e.g. the Memory Manager Policy decides if it is worthy to split a cached memory block into smaller chunks or directly allocate new memory from the system).
- **Allocation Manager**. The Allocation Manager is responsible for the memory allocation/ release from the host system.
- **Memory Block Pool**. The Memory Block Pool is responsible for the allocation/release of memory block objects.
- **Memory Pool**. The Memory Pool is used to cache memory blocks that are not currently used by a customer.

4.4 Extoll Routing Manager

4.4.1 Requirements

The ERM requirements are:

- The ERM must be able to handle multiple routing tables.
- The ERM must manage the routing tables.
- The ERM is not responsible for the calculation of the routing strings.
- The ERM must be able to associate several routing strings with the same routing target.

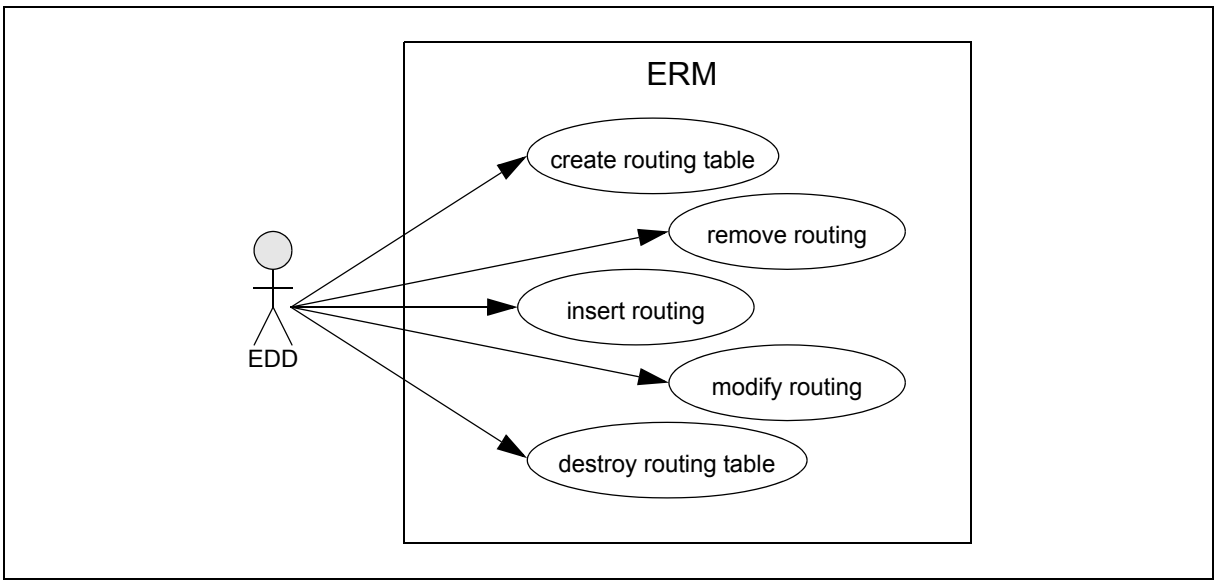


Figure 4-8 : ERM Use Case Diagram

As shown in Figure 4-8 the ERM only has the EDD as customer. The ERM must offer functions to create and destroy routing tables, because for each available device an extra routing table is required. For each routing table, functions to insert, delete and modify the routing strings must be offered.

4.4.2 Design

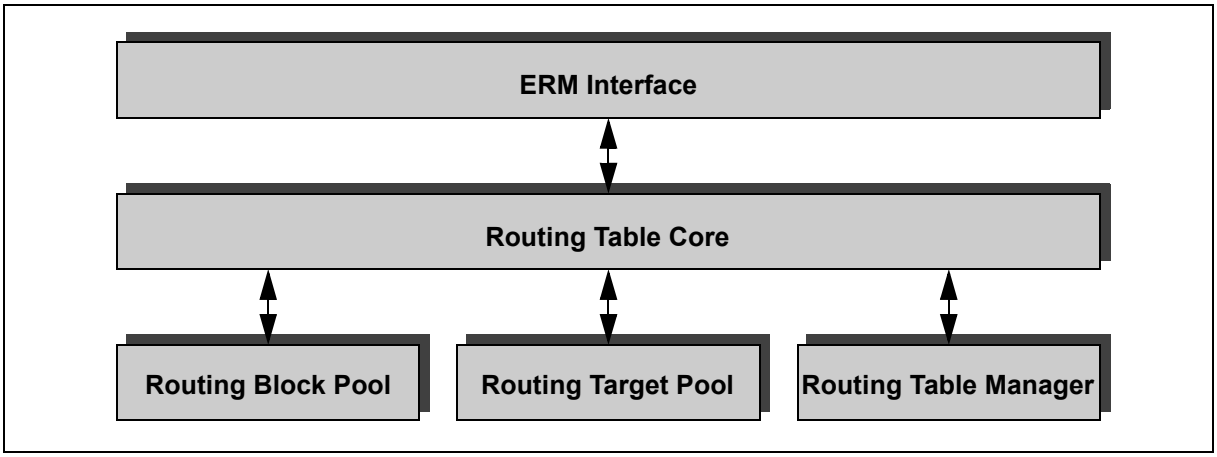


Figure 4-9 : ERM Design

Definition 4-4 : **Routing Target Object**. A routing target object is a software object that describes a routing target.

Definition 4-5 : **Routing Block Object**. A routing block object describe a part of the routing space that contains a routing string.

As shown in Figure 4-9 the ERM consists of the following components:

- **ERM Interface**. The ERM interface consists of the functions that are offered to customers.
- **Routing Table Core**. The Routing Table Core is responsible for handling the customer requests.
- **Routing Block Pool**. The Routing Block Pool is a dynamic cache of routing block objects. If there are no free routing block objects the Routing Block Pool will automatically allocate new routing block objects.
- **Routing Target Pool**. The Routing Target Pool is a dynamic cache of routing target objects. If there are no free routing target objects the Routing Target Pool will automatically allocate new routing target objects.
- **Routing Table Manager**. The Routing Table Manager performs the low level routing table manipulations of a routing table.

4.5 Extoll Device Driver

4.5.1 Requirements

- The EDD must be able to support multiple NICs.
- The EDD must support user and super user access methods to the available resources.
- The EDD must take care about the allocated resources and is responsible that under normal circumstances no allocated resources get lost.

- The EDD must be able to support the select/poll/epoll interface to allow the usage of the Extoll resources together with other resources of the operating system.
- The EDD should be as independent as possible of the I/O bus between the host system and the Extoll NIC.

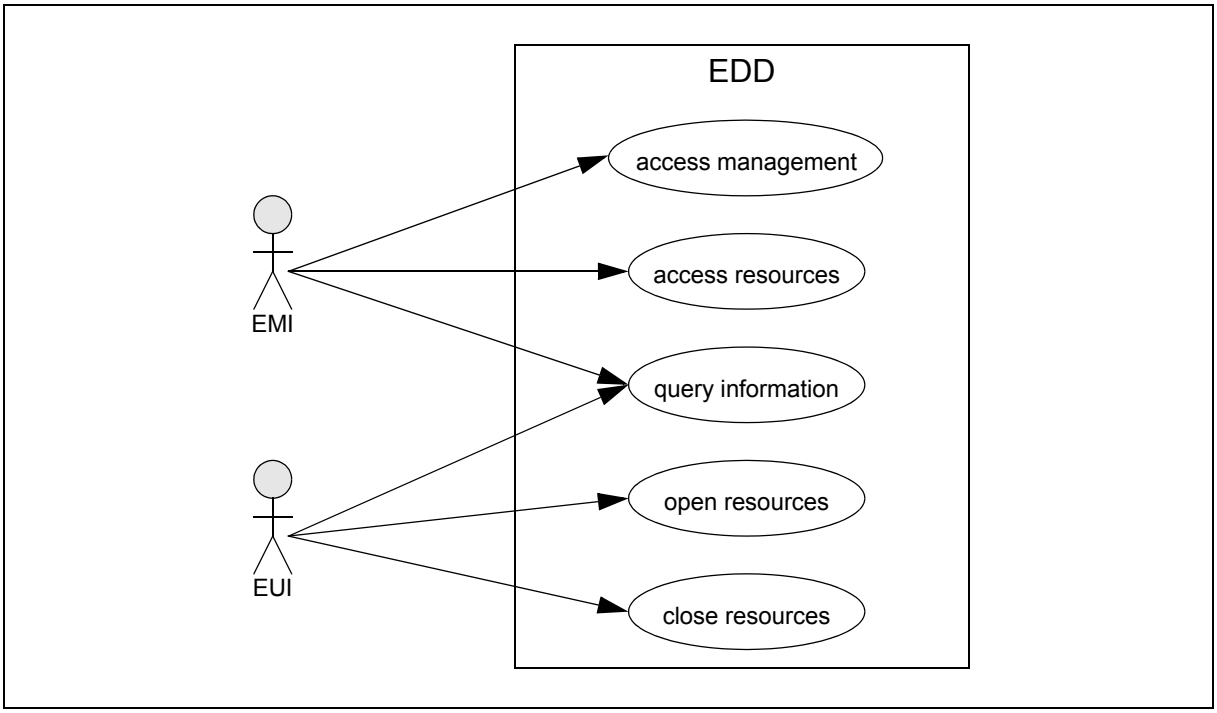


Figure 4-10 : EDD Use Case Diagram

As shown in Figure 4-10 the EDD has the EMI and EUI as customers. For the EMI customer EDD must offer functionality to access all available resources and all management functions. In the case of the EUI customer a less privileged access to the resources must be offered.

4.5.2 Design

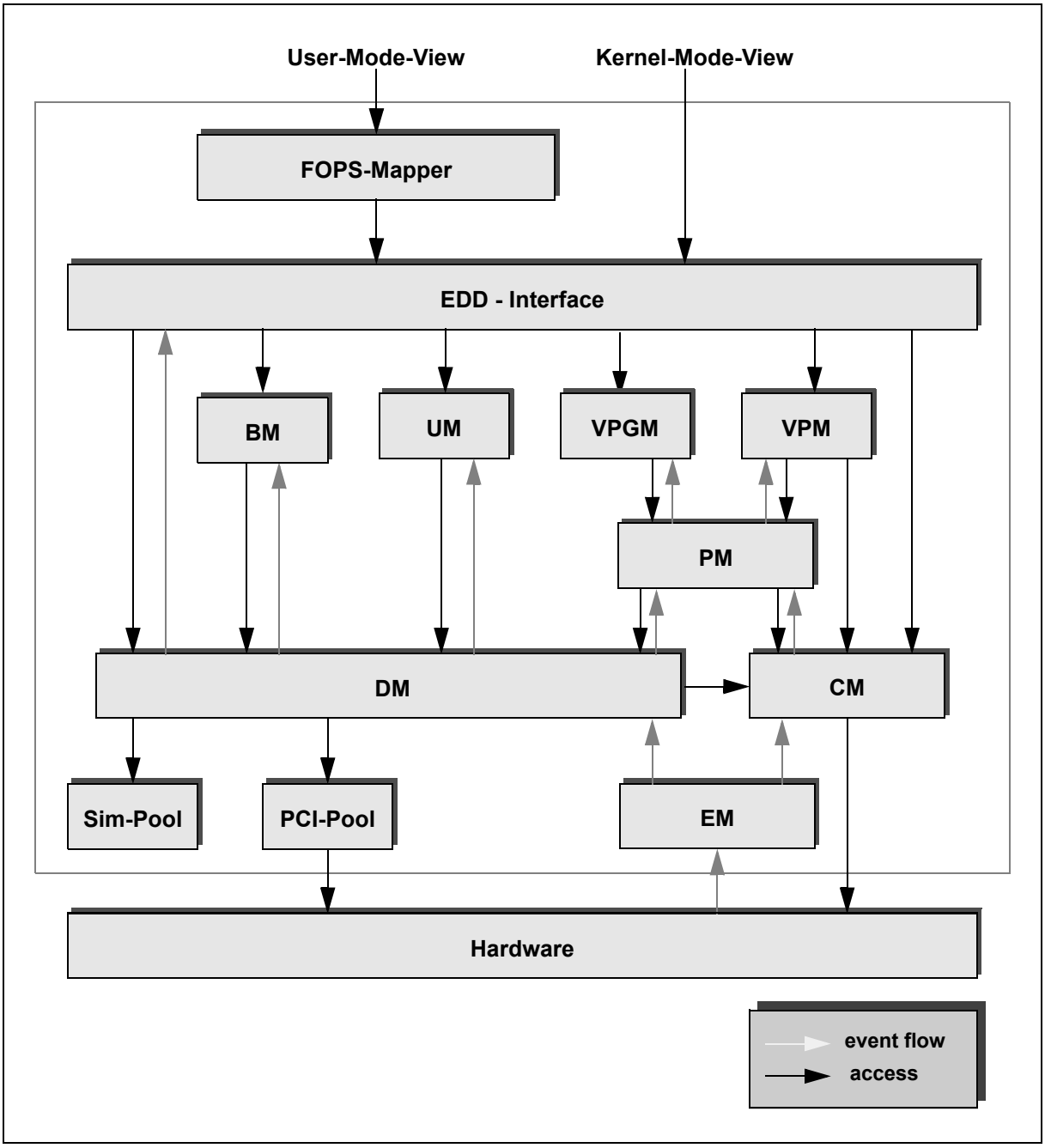


Figure 4-11 : EDD Design

Definition 4-6 : **EDD Device**. An edd device is an abstract software object that is used to describe an Extoll device inside the EDD. All sub-modules of the EDD, except the Device Pools, operate on EDD Device objects. Therefore the biggest part of the EDD is independent of the real Extoll device adapter (e.g. PCI or simulation adapter). This approach has the advantages that new Extoll adapters (e.g. via HyperTransport) can be introduced without the

need to change the EDD (only a new device pool needs to be added).

Definition 4-7 : **Device Pool**. A device pool is a collection of all devices that are connected via the same I/O bus, e.g. the PCI device pool contains all PCI(X) Extoll NICs. The Device Pool is responsible to find all devices that are connected to the corresponding I/O bus and create corresponding EDD device objects.

Definition 4-8 : **EDD Process**. For each user process that opens an Extoll resource a corresponding Extoll Process is created. An Extoll Process collects information about all Extoll resources that have been associated with the user process.

Definition 4-9 : **EDD Connection**. An EDD connection is an abstract software object that represents a routing information (routing offset and routing length). There is no corresponding equivalent in the hardware. An EDD connection is a special object that allows the EDD to track which process use which routing entries. This is necessary when e.g. a process crashes, then the EDD is able to update the reference count of the corresponding routing entries.

Definition 4-10 : **Virtual Port Group (VPG)**. The Virtual Port Group is an abstract object that has no representation in the Extoll hardware. The Virtual Port Group allows the user to group several VPs together and wait until one of the VPs in the VPG received a new notification.

The design of the EDD module is shown in Figure 4-11. The EDD module consists of the following sub-modules:

- **Event Manager (EM)**. The event manager is responsible for the processing of the events that are generated from the devices (See “Event System” on page 34).
- **Cache Manager (CM)**. The cache manager is responsible for TLB management. This management includes the address translation from virtual to physical addresses. If the address cannot be translated immediately because the page is swapped out the Cache Manager is responsible to swap the page in.
- **Device Manager (DM)**. The device manager is responsible for the access to all available devices in the system. The available devices are collected from the device pools.
- **Simulation Device Pool (Sim-Pool)**. The simulation device pool contains a set of software devices. The device of the Simulation Device Pool does not have a representation in real hardware. The missing I/O memory of a real device is replaced by RAM memory blocks.
- **PCI Device Pool (PCI-Pool)**. The PCI Device Pool contains all Extoll devices that are connected via a PCI/PCI-X bus.
- **Process Manager (PM)**. The process manager is responsible for managing Extoll Processes.
- **Virtual Port Manager (VPM)**. The Virtual Port Manager is responsible for the management of the VPs that belong to a device.

- **Virtual Port Group Manager (VPGM)**. The Virtual Port Group Manager is responsible for the management of the VPGs.
- **Ultra Manager (UM)**. The Ultra Manager is responsible for the management of the Ultra Ports that belong to a device.
- **Barrier Manager (BM)**. The Barrier Manager is responsible for the management of the Barriers that belong to a device.
- **EDD Interface**. The interface that is offered to the customers.
- **FOPS-Mapper**[1]. The FOPS-Mapper maps the EDD-Interface from the kernel-space into the user-space.

4.6 Extoll User Interface

4.6.1 Requirements

The requirements of the EUI are:

- The EUI should offer a simple and easy to use interface for accessing all user relevant features.
- The EUI should have the same interface for kernel- and user-space.
- The EUI is an abstract layer that hides the low-level details and the differences of the kernel- and user-level access to the EDD from the user.

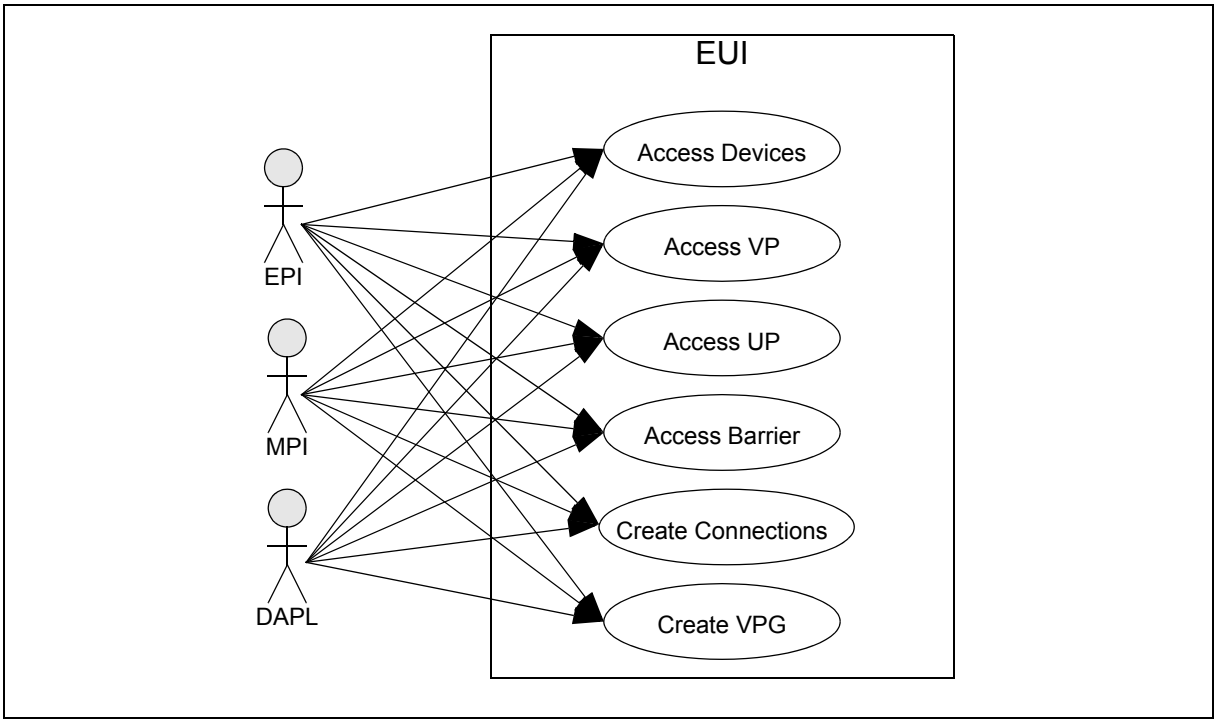


Figure 4-12 : EUI Use Case Diagram

1. FOPS means “file operations” and is a structure that consists of function pointers to functions that can be performed on a file (e.g. open, close, write). For a complete description refer to [LDD].

Figure 4-12 shows that the EUI has three possible customers. All customers require access to all resources that are accessible for the user.

4.6.2 Design

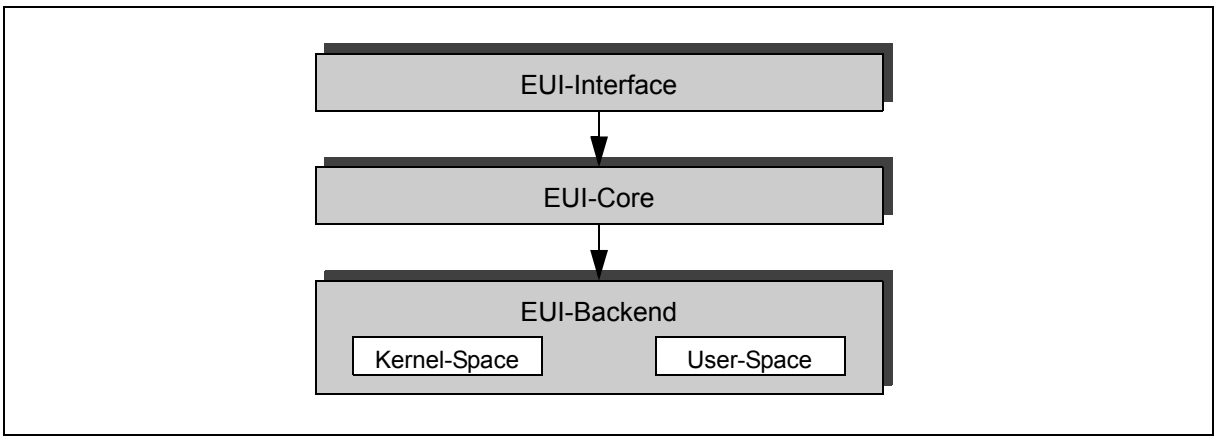


Figure 4-13 : Design EUI

- **EUI Interface**. The EUI interface consists of the functions that are offered to the customers.
- **EUI Core**. The EUI Core layer consists of all backend independent functionality.
- **EUI Backend**. The EUI Backend layer is responsible for dealing with the EDD interface either via the kernel- or the user-interface. This layer hides all differences between the different backends.

4.7 Extoll Management Interface

4.7.1 Requirements

The requirements of the EMI are:

- The EMI should offer a simple and easy to use interface to access all super user features.
- The EUI should have the same interface in kernel- and user-space.
- The EUI is an abstraction layer that hides the low-level details and the differences of the kernel- and user-level access to the EDD from the user.

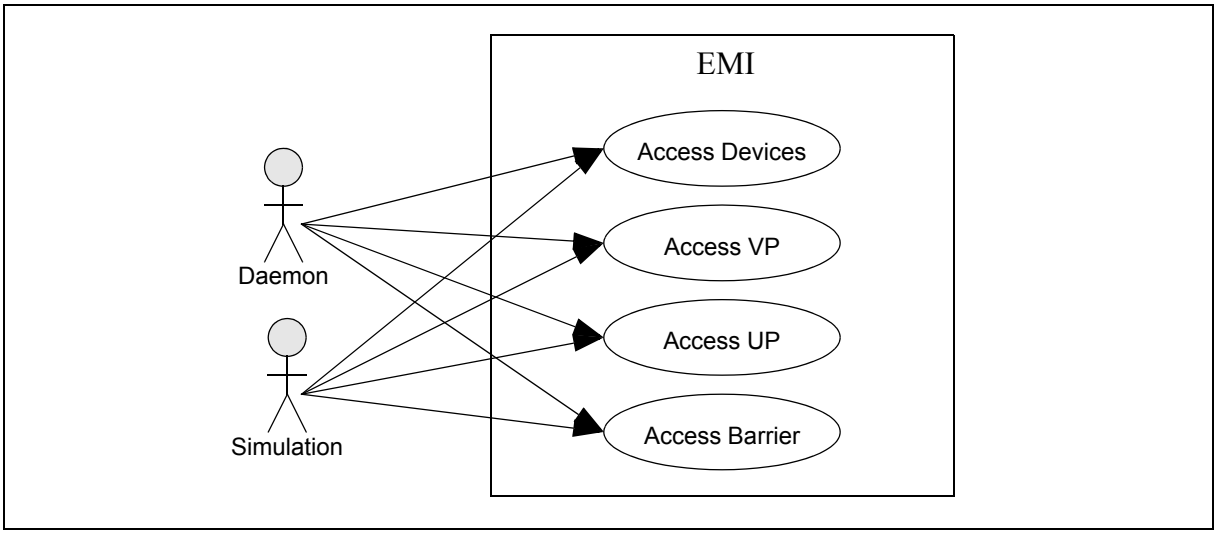


Figure 4-14 : EMI Use Case Diagram

As shown in Figure 4-14 the EUI has two possible customers. All customers require access to all resources that are accessible for the super user. The EMI must offer functionality to access all resources of a Extoll device.

4.7.2 Design

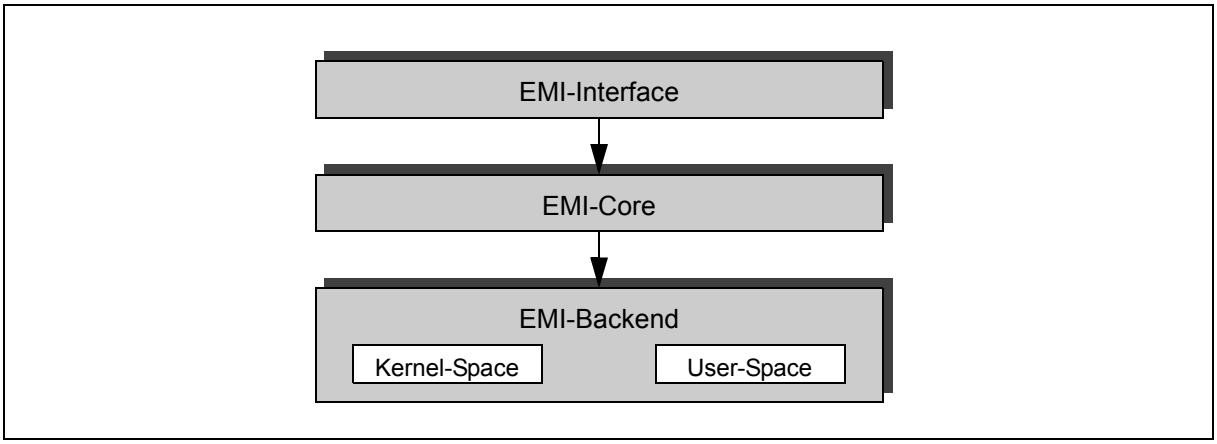


Figure 4-15 : Design EUI

- **EMI Interface**. The EUI interface consists of the functions that are offered to the customers.
- **EMI Core**. The EUI Core layer consists of all backend independent functionality.
- **EMI Backend**. The EUI Backend layer is responsible for dealing with the EDD interface either via the kernel- or the user-interface. This layer hides all differences between the different backends.

4.8 Extoll Programming Interface

4.8.1 Requirements

The EPI has the following requirements:

- The EPI must be fast and efficient.
- The EPI must have a small and easy to use interface.
- The EPI must offer all the functionality of the Extoll NIC to the user.

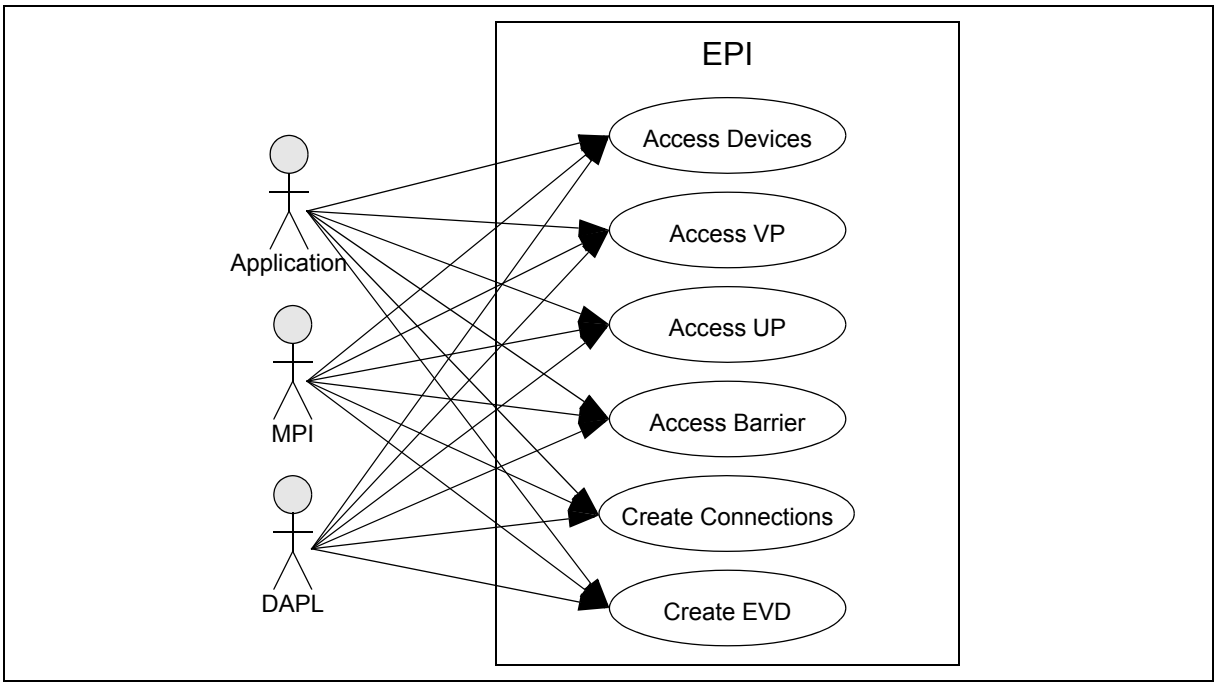


Figure 4-16 : EPI Use Case Diagram

As shown in Figure 4-16 the EPI has several customers that all need access to the resources for normal users. Only a small set of possible customer is shown in Figure 4-16. MPI and DAPL can be ported to the EUI or the EPI interface. Both approaches have their advantages and disadvantages.

4.8.2 Design

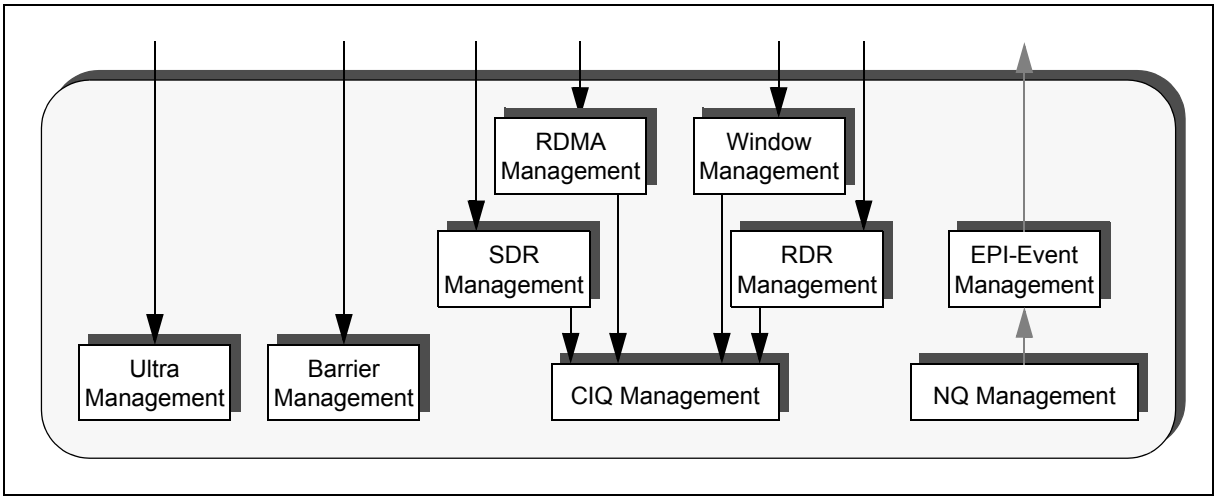


Figure 4-17 : EPI Design

As shown in Figure 4-17 the EPI consists of the following sub-units:

- **Ultra Management**. The Ultra Management is responsible for the management of the access and control to the Ultra ports.
- **Barrier Management**. The Barrier Management is responsible for the management of the access and control to the Barrier ports.
- **RDMA Management**. The RDMA Management is responsible for the creation of RDMA specific VCIs which are passed to the VCI management.
- **Window Management**. The Window Management is responsible for the creation, modification and deletion of memory windows.
- **SDR Management**. The SDR Management is responsible for the management of the SDR in the case of DMA send and posted sends.
- **RDR Management**. The RDR Management is responsible for the management of the RDR in the case of DMA receive and posted receives.
- **CIQ Management**. The CIQ Management is responsible for the management of the CIQ and the triggering of VCIs.
- **NQ Management**. The NQ Management is responsible for the management of the NQ.
- **EPI-Event Management**. The EPI-Event Management converts notifications into software events. The EPI-Event Management must reorder the notification if they arise out-of-order.

4.9 Extoll Daemon

The Extoll Daemon is special management system that is running on every node in the network. The Extoll Daemon has similar tasks as the Atoll Daemon ([NUS03]). The main tasks of the Extoll daemon are:

- **Topology Exploration**. The Extoll Daemon is responsible for the determination of the current network topology.

- **Routing Determination**. Based on the network topology the Extoll Daemon is responsible for the calculation of the routing strings for each host.
- **Error Detection/Correction**. If errors occur in the network or the local NICs the Extoll Daemon is responsible to solve the problem and put the system back into a working state. To fulfil this task the Extoll Daemon has to work very closely with the EDD.
- **Barrier Management**. Based on the network topology the Extoll Daemon is responsible for the calculation, configuration and management of the barrier trees (See “The Barrier” on page 60).

The design and development of the Extoll Daemon is not part of this thesis.

Implementation of the ESS

CHAPTER 5

This chapter gives an overview of the implementation of the Extoll Software Stack. The first part of the chapter covers the general aspects of the implementation. In addition to the global implementation approaches the testing mechanism is presented.

After the general design has been discussed a more detailed description of the ESS implementation is given. For each ESS component a description of the implementation of the most important features is given. For a complete overview of the ESS interfaces refer to the ESS Reference Manual ([STORK05]).

5.1 General

5.1.1 Symbol Resolving

As a consequence of module stacking the problem occurs that one module needs the address information of symbols that are located in modules. As show in Figure 5-1 two possibilities to solve this problem exists, either by an automatic or a manual symbol resolving mechanism.

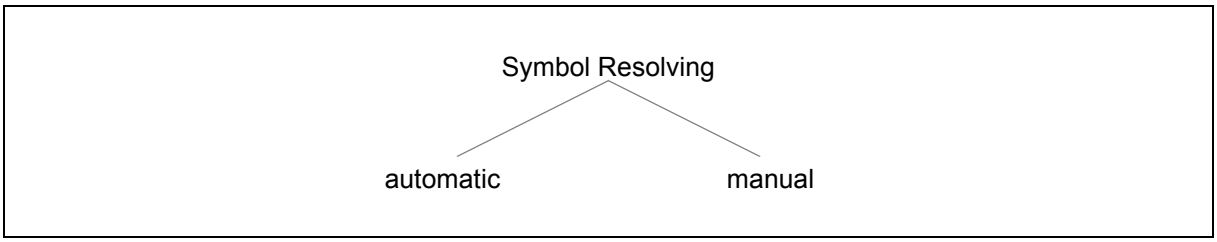


Figure 5-1 : Symbol Resolving Decision Tree

5.1.1.1 Automatic Symbol Resolving

The automatic symbol resolving is done by the standard mechanism of module loading. If a module is loaded which has unresolved symbols the mechanism tries to find the required symbol in one of the other installed modules. If the symbols could be found in another module this module will automatically be loaded else the whole module loading mechanism will be cancelled with an error.

Advantages

- No extra code to resolve the symbols is required.
- Dependent modules are automatically loaded (if they are installed).
- The Kernel knows which modules depend on each other.

Disadvantages

- During the linking of the kernel modules compiler warnings will be created for each unresolved symbol.

5.1.1.2 Manual Symbol Resolving

In the case of a manual symbol resolving the module must itself search for the required symbols during the initialization. If the required symbols cannot be found the initialization breaks with an error and the module will not be loaded.

Advantages

- No compiler warnings.

Disadvantages

- Dependent modules will not automatically be loaded.
- Extra code to find the symbols must be written.
- The kernel does not know which modules depend on each other.
- There must be an indirect step for each symbol (e.g. function pointer).
- The mechanism of how symbols can be searched can changed between different kernel versions.

5.1.1.3 Conclusion : Symbol Resolving

The ESS will use the automatic approach because of the following reasons . The automatic approach require no extra code, especially it looks and feels like a library mechanism. This leads to the fact that differences between kernel- and user-level implementation will be reduced. The automatic approach is the default mechanism of the Linux system.

5.1.2 Extoll Testing Framework

To verify the correct implementation and behaviour of the ESS a testing mechanism was required. The currently available unit testing frameworks do not fit for the ESS, because the ESS is running in kernel- and user-space and the available unit testing frameworks work only in one part, either kernel- or user-space. Therefore a new testing framework for the ESS had to be developed. The developed testing framework for the Extoll is called Extoll Testing Framework (ETF).

5.1.2.1 Design Goals

The ETF was designed with the following design goals:

- Testing environment must be able to testing kernel- and user-space test cases.
- The ETF should be fast and efficient.
- The ETF must be able to run the tests automatically and provide a short and informative report to the user. The report should be as small as possible and as informative as necessary to allow the user a fast overview.
- The ETF must be simple to use, manage and to extend.

- In the case that some test cases fail the testing framework must guaranty that everything that belongs to the failed test case will be removed from the system.

5.1.2.2 Structure

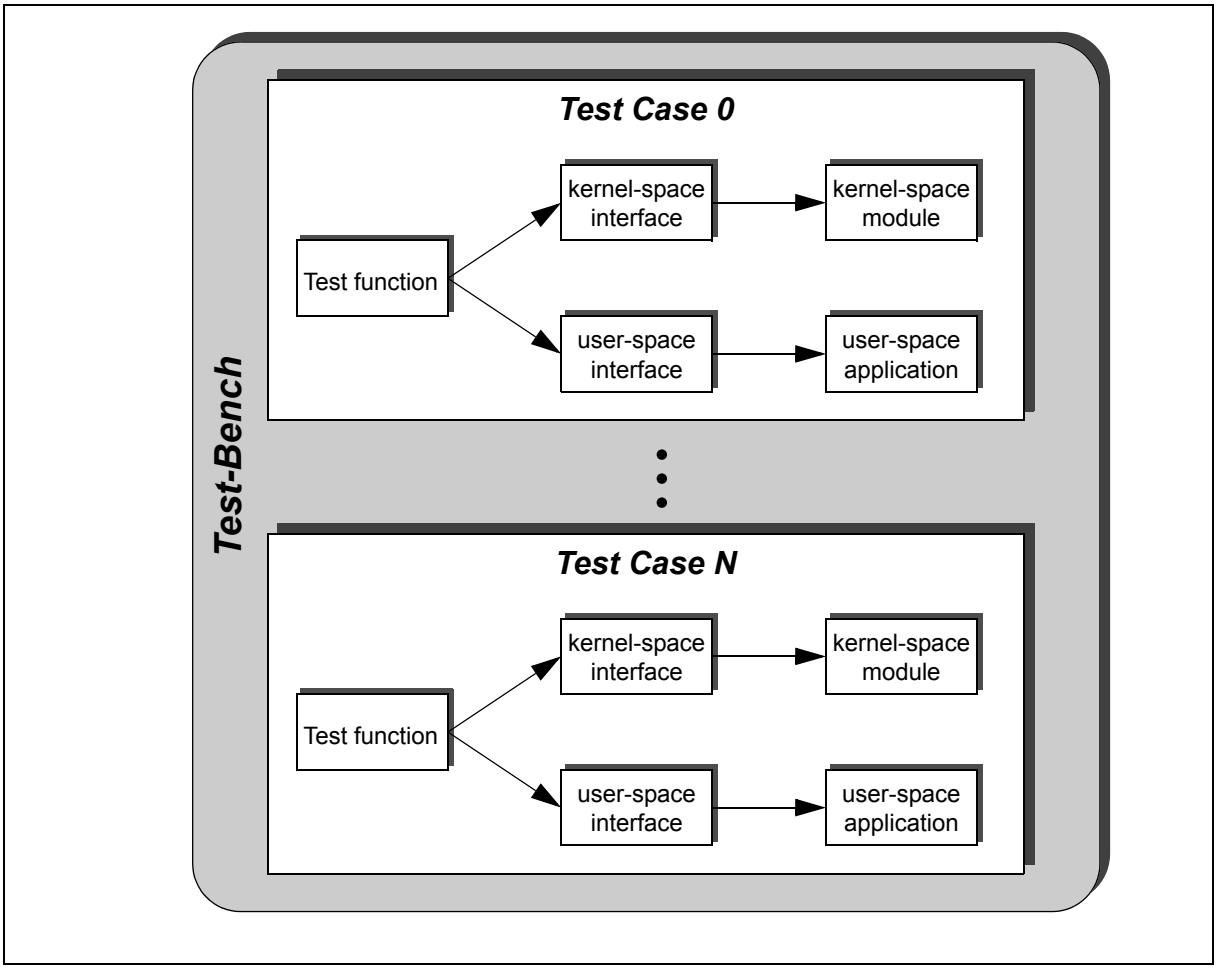


Figure 5-2 : Test-Bench Structure

To achieve the required goals the ETF has the general structure that is shown in Figure 5-2. Every test-bench consists of several test cases. A test case is responsible for testing one particular part of the implementation. A test case is able to test the property in kernel-space and user-space. As shown in Figure 5-2 every test case has a single test function. The test function is a C function that performs the whole testing (initialisation, testing, cleanup). The test function returns 0 on success and -1 on an error.

Because the interfaces of the ESS modules are equal for kernel- and user-space the test function is independent and therefore can be used for testing in kernel- and user-space. To execute the test function in user space the test function is compiled and linked with the user-space interface. The user-space interface consists of a main function that executes the test function and use the return value of the testing function as exit code. For the kernel space the test function is compiled and linked with the kernel-space interface. The kernel-space interface consists of module init- and exit-function ([LDD]). The module init function will execute the test function and use the corresponding return value as return value of the module initialization.

All test cases are executed in a sequential order by the test bench. The test-bench is a shell script that either executes the user-level program or tries to load the kernel-level module. The failure of the test function can be detected by the exit value of the started test case. If the executed test application or the loading of the test module returns 0 the test has been passed. In the case that the exit value is not equal 0 the test has failed. Listing 5-1 shows an example of an test bench run.

```
========================================================================
==              Test Bench for the Extoll Routing Manager             ==
========================================================================
Running ERM Interface Test ..................................... [passed]
Running Routing Pattern 1x1 Test ............................... [passed]
Running Routing Pattern 4x4 Test ............................... [passed]
Running Routing Pattern 8x8 Test ............................... [passed]
Running Routing Pattern 16x16 Test ............................. [passed]
Running Routing Pattern 32x32 Test ............................. [passed]
Running Routing Pattern 64x64 Test ............................. [passed]
Running Routing Pattern 128x128 Test ........................... [passed]
Running Routing Pattern 256x256 Test ........................... [passed]
Running Routing Pattern 512x512 Test ........................... [passed]
Running Routing Pattern 1024x1024 Test ......................... [passed]
========================================================================
  Runned Tests   : 11
  Passed Tests   : 11
  Failed Tests   : 0
========================================================================
```

Listing 5-1 : Output Test Bench of ERM

5.1.3 Configuration

5.1.3.1 Compiletime Configuration

All values/parameters in the ESS are initialized with reasonable default value. While many of the parameters are changeable during loadtime and/or runtime, there exists some parameters that are fixed and can only be changed by re-compilation (e.g. the number of maximum supported devices).

5.1.3.2 Loadtime Configuration

Every kernel module of the ESS supports module parameters. The module parameters are passed when the module is loaded into the kernel. The available module parameters can be determined with the *modinfo* command.

5.1.3.3 Runtime Configuration

All kernel modules support the ability to change some parameters during runtime. To view and modify the runtime parameters the *extollctl* tool (see “extollctl” on page 136) can be used. The mechanism of the runtime parameters works via the Linux sysfs ([LDD], [MAURER]).

5.2 ELS

As described in “Extoll Logging System” on page 76 the ELS system consists of 3 layers. The ELS systems supports a per log priority threshold. With this threshold it is possible to filter out low priority messages. This priority threshold can be changed during runtime and therefore offers the ability to use the ELS system in a development and productive environment. This approach also has the advantage that when in a productive environment an errors occurs, the threshold can be changed back to the lowest possible priority and the full debugging capabilities can be accessed to search for the error.

5.3 EMM

As described in “Extoll Memory Manager” on page 77 the real management is specified by the memory management policy. The policy mechanism is realized via a set of function pointers to functions that implement the corresponding functionality. In the current implementation a simple memory management policy has been implemented. The simple memory management policy directly allocates and releases the memory from the host system without any caching of the memory buffers. The policy can be changed at loadtime of the EMM module.

5.4 ERM

5.4.1 Routing Table Management

As described in “Extoll Routing Manager” on page 79, the ERM supports several routing tables. The ERM supports a number of routing tables up to the maximum device count. The current implementation of a routing table is presented in Figure 5-3.

As shown in Figure 5-3 every routing table contains a physical continuous routing area where the routing information is stored. This routing area is subdivided into smaller areas, called routing slots, that either contain routing information or are unused. Each routing slot of the routing area is described by a routing block object. All routing blocks are connected in a list. This mechanism allows the ERM easily to merge several free routing slots that are continuous into one big free routing slot. To speed-up the determination of free routing slots each routing table manages a list of all free routing blocks in the routing table. The last part of a routing table is a target set that contains all routing target objects of the routing table. To speed-up the search operation for a specific target, the target set is implemented as a hash table that uses the target device ID as input for the hash function. Each routing target has fixed amount of routing slots that can be associated with the routing target object. This allows the implementation to use several routing strings per target.

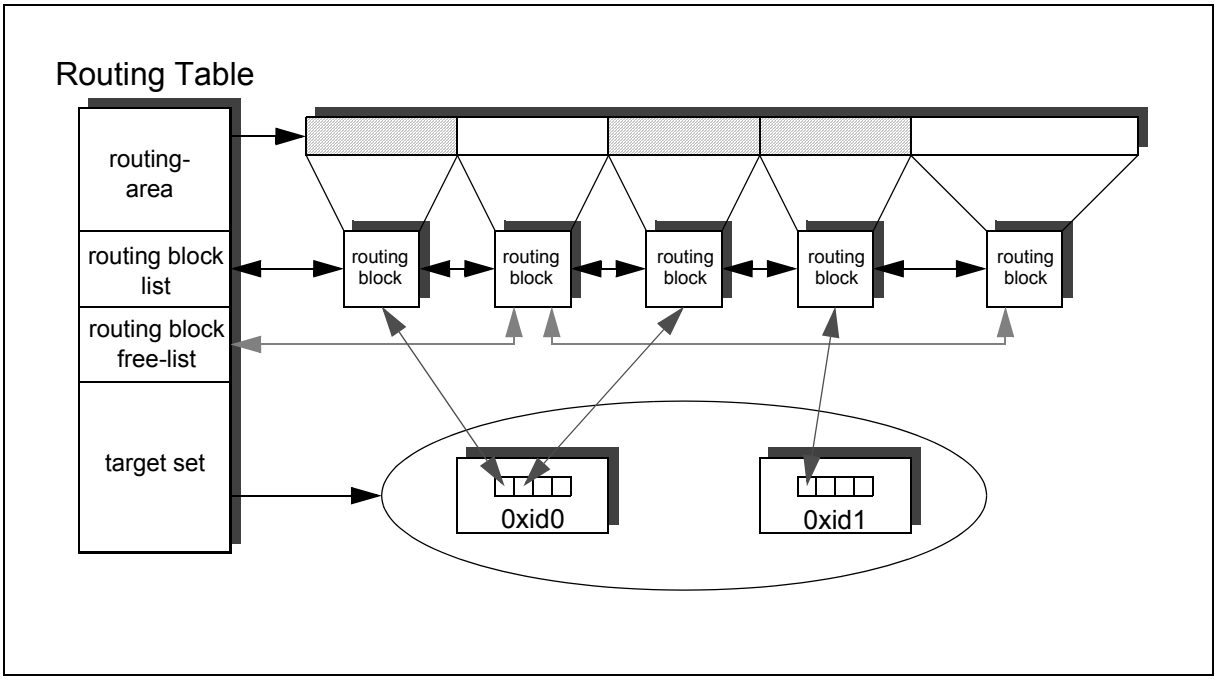


Figure 5-3 : ERM Routing Table Implementation

5.4.2 Routing Failure

In the case that a certain routing string becomes invalid (e.g. a network link breaks) the user will receive a notification with the corresponding error. After the user receives a notification with such an error the user needs to disconnect the old connection (release the routing string) and then the user needs to create a new connection to the target (request a new routing string). With this new connection the user is able to restart the failed operation.

5.5 EDD

5.5.1 Process Management

To perform a fast look up for an EDD process all EDD processes are managed in a hash table. The input for the hash function is the address of the Linux task structure of the Linux process.

5.5.1.1 Threads

Under Linux every thread is represented by a task in the Linux kernel and a process is represented by a group of tasks ([MAURER]). Each task group has one distinct main task (main thread), the so called group leader. When the group leader ends all other tasks in the same task group will be terminated too, while the other tasks (threads) in the task group can be created and terminated in any order without any effect to the whole task group. Therefore each EDD process is always associated with the group leader task of a task group.

5.5.2 Device management

As described in “Extoll Device Driver” on page 80 the EDD device is the central object of the EDD. The EDD device mainly consists of 2 Extoll memory objects. The first memory object describes the Extoll Device Resource Area and the second describes the Extoll Device Trigger Page Area. Additionally each EDD device contains information about the device and the available resources on the device. A unique number is associated with each EDD device that is used to identify the device inside the system.

5.5.3 VP Management

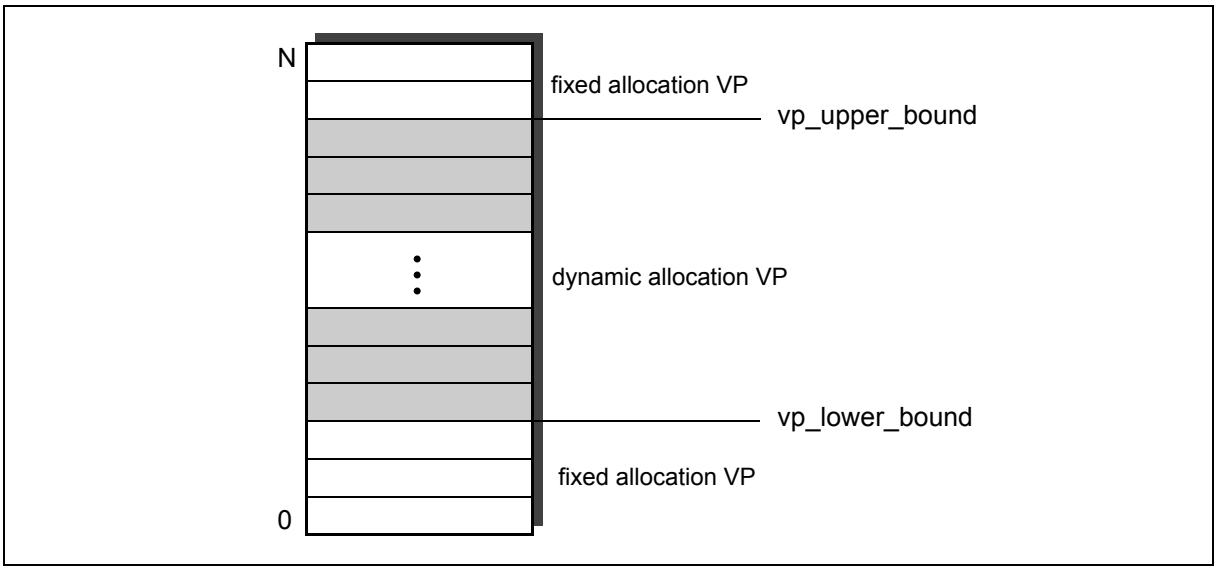
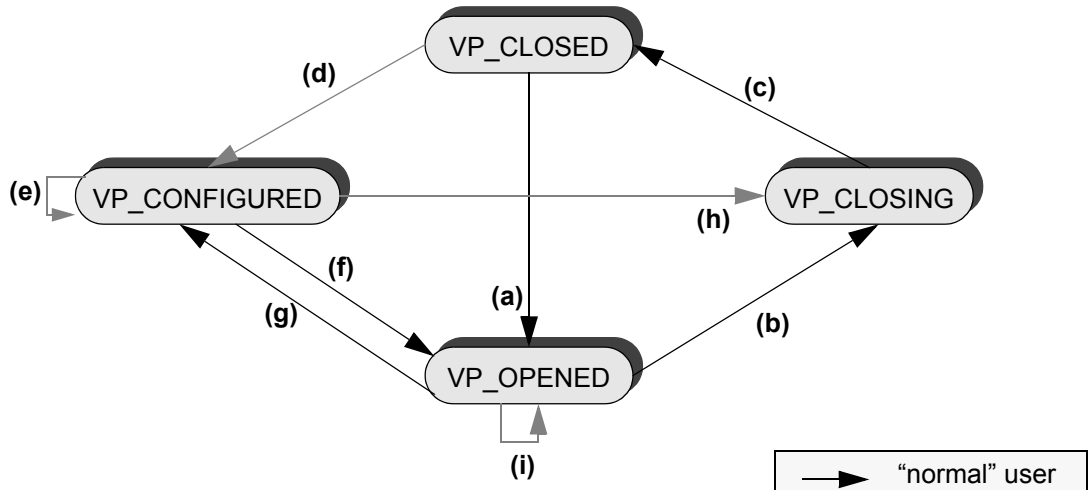


Figure 5-4 : VP Allocation Map

To open a VP the user must specify the corresponding VPID of the VP that should be opened. Many general purpose user (e.g. MPI) do not require a specific VPID. Such general purpose users can use the symbolic constant EXTOLL_VP_ID_ANY to force the VP manager to dynamically open any VPID that is currently unused. The VPIDs that are dynamically managed by the VP manager are located in the area between the value of *vp_lower_bound* and *vp_upper_bound* (see Figure 5-4). This approach has the advantage that the first VPIDs can be reserved for users that explicitly request these VPIDs and are not associated to a general purpose user by accident.

As mentioned in “Extoll Device Driver” on page 80 the EDD differentiates between a normal users and super users. A normal user is only to able open resources that are actually not opened by a normal user. The super users are able to access all resources independently whether the resources have already been opened by a normal user or not. It is also possible that multiple super users access the same resource at the same time bit it is important to note that only normal users are able to use the opened resources for communication.



(a) A normal user opens a closed/free virtual port.

(b) The normal user closed his virtual port and there were no "super" users simultaneously accessing the same virtual port.

(c) All resources and operations of the VP have been ended/freed.

(d) A "super" user opened an unused/free port. There is no "normal" user at the moment.

(e) A "super" user opened a virtual ports that is only opened by "super" users and no "normal" user.

(f) A virtual port that has been opened by a "super" user has been requested by a "normal" user.

(g) A "normal" user closed its virtual port, but there are still "super" users using the resources.

(h) A virtual port has been closed by the last "super" user.

(i) A "super" user has opened the VP.

Figure 5-5 : VP States

The support for normal and super users leads to the following 4 states for a VP (see Figure 5-5):

- **VP_CLOSED**. The state VP_CLOSED means that no normal and no super user has opened the corresponding VP. Because there are no users of the VP there are no resources associated with the VP.
- **VP_OPENED**. The state VP_OPENED means that a normal user has opened the VP. Beside the normal user there can be any number of super user accessing the VP at the same time. Because the VP has at least one user, the VP has a complete set of resources.
- **VP_CONFIGURED**. The VP_CONFIGURED state is reached when at least one super user has accessed the VP but no normal user. Because there exists at least one user the VP has a complete set of resources.
- **VP_CLOSING**. Like the VP_CLOSED state a VP in the VP_CLOSING state has no users, but unlike in the VP_CLOSED state a VP in the VP_CLOSING state has still some resources. The VP_CLOSING is a kind of parking state for a VP that needs to wait until all resources could be freed.

Every time when a normal user opens a VP (enters state VP_OPENED) or the normal user closes the VP again (leaves state VP_OPENED) functions to enable and disable the VP are called (see Listing 5-2). Both functions are responsible to update the corresponding VP descriptor and to flush the possibly existing old entries from the caches. The *edd_vp_disable* function additionally must ensure that no resources of the VP are used any longer by the Extoll NIC.

.

```
extoll_error_t edd_vp_enable(edd_vp_t *vp)
extoll_error_t edd_vp_disable(edd_vp_t *vp)
```

Listing 5-2 : VP enable/disable Prototypes

5.5.4 Barrier Management

The barrier management supports no dynamical allocation of barriers. Therefore the user that opens a barrier needs to know which barrier he wants to join. Like in the VP only one normal user is able to open a barrier at a time while several super users are allowed to open a barrier simultaneously.

5.5.5 UP Management

The UP Management is similar to the VP management. Because an UP has no dynamic resources (except the receive UP) and the UP are fixed hardware resources that are associated with a process the management of the UP is much simpler. As for a VP only one normal user is allowed to open a VP at once while several super users are allowed to open simultaneously.

5.5.6 Connections Management

There are no special management issues for the connection management. The user can create new connections as long as there is enough memory available. Every connection that has been created by the user must also be freed again.

5.5.7 VPG Management

There are no special management issues for the VPG management. The user can create new VPG as long as there is enough memory available. Every VPG that has been created by the user must also be freed again. Before a VPG can be freed the user must first remove all previous inserted VPs.

5.5.8 FOPS-Mapper

The FOPS-Mapper is a mechanism that maps the EDD interface that is offered by the EDD kernel module to the user space. The mapping is realized by using device files ([LDD]). All device file that belong to the EDD are locate in the “/dev/extoll” sub-directory. Unlike in the Atoll implementation the Extoll device files are hierarchically organised (see Figure 5-6). The naming of the device files is self explanatory. Device files that are prefixed with an ‘s’ are for super user access. For the creation of the device files the *extoll_mknod* tool can be used (see “extoll_mknod” on page 133).

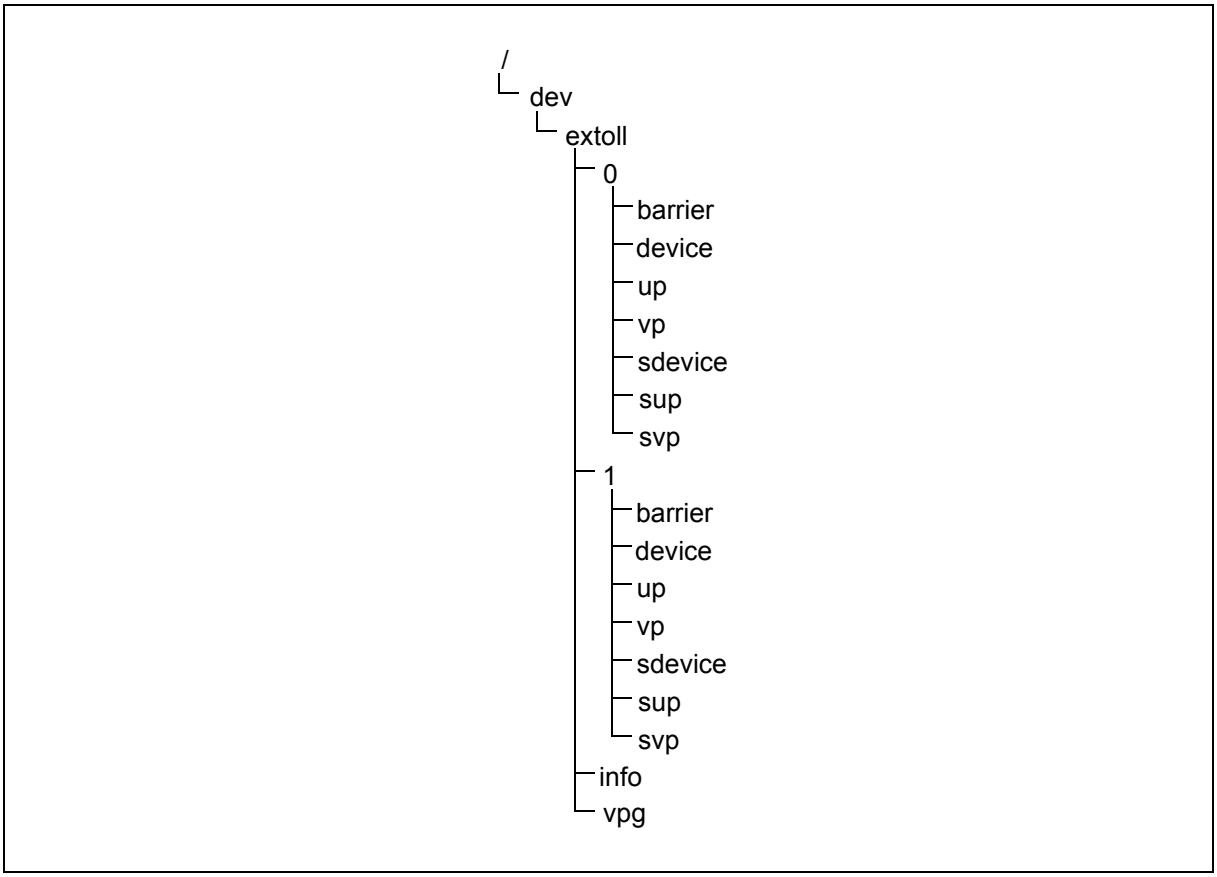


Figure 5-6 : FOPS-Mapper Device Files

The partitioning of the minor device number is shown in Figure 5-7 and Table 5-1. The device is the number of the EDD device. The mode of the minor number specifies if the accessor is normal user or super user. The mode encoding is shown in Table 5-3. The resource part of the minor device specifies which device resources to open. The resource encoding is shown in Table 5-2.

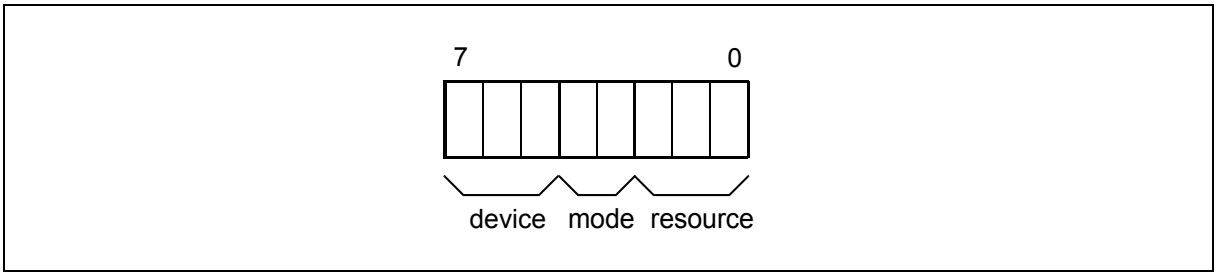


Figure 5-7 : Device file minor number partitioning

Minor Number Bits	*Description*
7 - 5	The device number. This value represents the number of the Extoll NICs.

Table 5-1: Minor device number partitioning

Minor Number Bits	*Description*
4 - 3	The mode. This value describes how the resources should be accessed (e.g. user or super user,).
2 - 0	The function-unit. This value identifies which resource to open, e.g. the device, a virtual port, a barrier, ...

Table 5-1: Minor device number partitioning

Coding	*Name*	*Description*
000_2	info	Provide access to general information.
001_2	device	To open a device.
010_2	vp	To open a VP.
011_2	barrier	To open a Barrier.
100_2	up	To open a Ultra port.
101_2 - 111_2	reserved	Reserved.

Table 5-2: Minor number resources

Code	*Privileges*	*Description*
00_2	none	Nobody is allowed to access this device. Not used.
01_2	user	Only user functionality is allowed.
10_2	super user	Super user functionality is allowed.
11_2	invalid	This combination is not allowed.

Table 5-3: Minor number privileges

For the memory mapping of the different memory buffers and regions a half-automatic mapping system has been implemented. First the user must query how many objects are mapped for a given object and then the user must query which objects (type and size) are mapped. With this system the order and the size of the mapped object can be dynamically evaluated.

5.5.9 Cache Management

The Cache Management mainly consists of the management of the TLB (“TLB - Translation Lookaside Buffer” on page 57). The two main tasks of the Cache Management are the virtual to physical address translation for the Extoll NIC (see Figure 5-8) and the maintenance of the TLB consistence when memory remappings occur (see Figure 5-9).

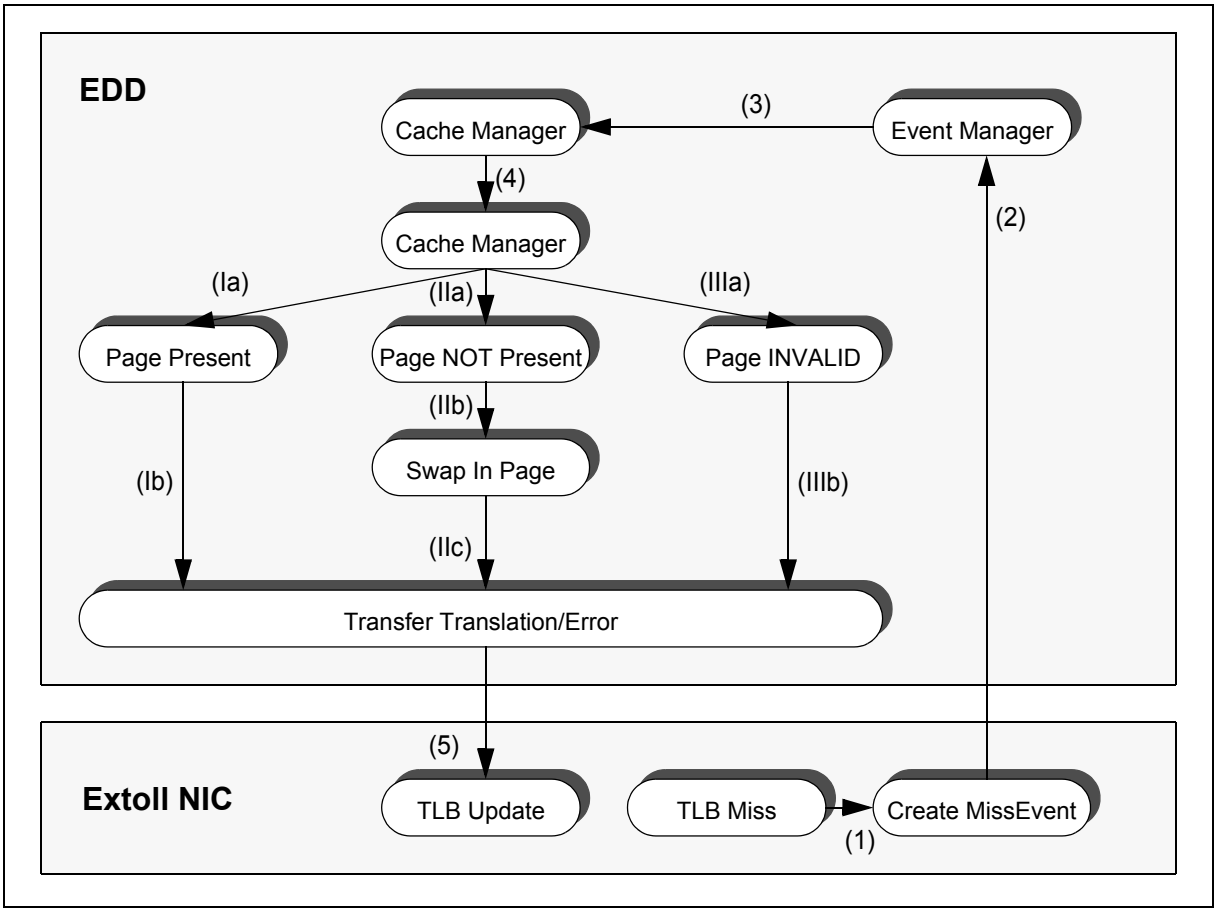


Figure 5-8 : TLB Miss Handling

Every time when the TLB has a miss, a corresponding TLB miss event will be created (Figure 5-8 (1)). The event will be received by the event manager of the EDD (Figure 5-8 (2)) and will be delivered to the cache manager (Figure 5-8 (3)). The cache manager tries to perform the requested translation (Figure 5-8 (4)). If the corresponding page is available in the main memory (Figure 5-8 (Ia)) the translation information will be transferred to the Extoll NIC and the TLB will update his information. If there exits a valid memory mapping for the requests address translation but the corresponding page is actually swapped out (Figure 5-8 (IIa)) the cache manager thread will be activated to swap the corresponding page in (Figure 5-8 (IIb)). An portable and easy way to cause the swap in of valid user pages can be achieved with the *get_user_pages* function ([LDD]). After the page has successfully been swapped in the address translation will be transferred to the Extoll NIC where the TLB will update his information. In the case that there is no valid memory mapping (Figure 5-8 (IIIa)) a special address translation will be transferred to the Extoll NIC to signal the TLB that there is no valid memory mapping.

The memory mapping of a process can always be changed (e.g. swapping, mmap, munmap, ...). Every time the memory mapping changes, the problem can occur that the Extoll NIC will use address translations which are no longer valid. Let us consider the case that the Extoll NIC has an address translation for a page that translates the virtual page address X to physical page address Y and exactly this page will be swapped out. After the page has been swapped out the page will be associated with a process or the operating system. If the Extoll would still use the old translation information the wrong processes could be damaged by over-writing the memory. Therefore the Extoll NIC must be consistent with the memory data structures of the processes.

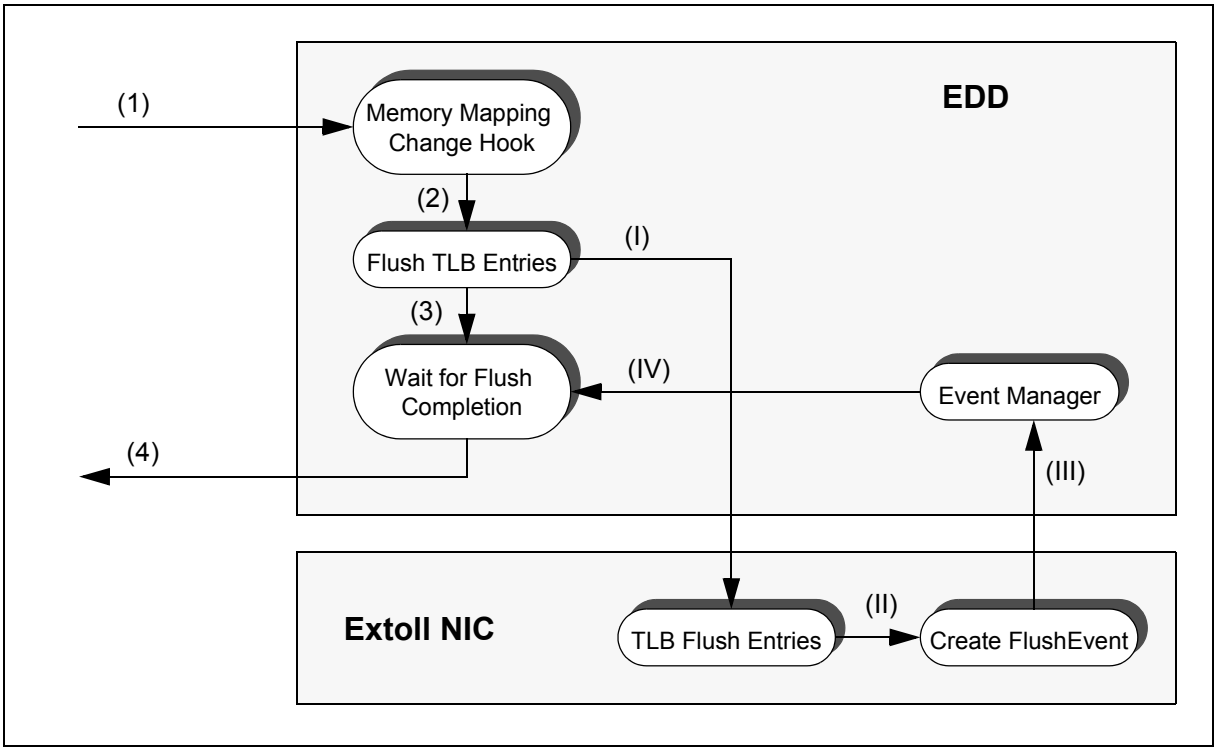


Figure 5-9 : Memory Remapping Handling

In Figure 5-9 the principle approach of a memory mapping modification is shown. Every time the mapping of a process is changed a callback function in the EDD cache manager will be called with all necessary information that describe the memory mapping modification (Figure 5-9 (1)). This hook function will determine if and which entries of the TLB are affected and flush them from the TLB (Figure 5-9 (2)). After the flush operation has been initialized the function will wait until the flush operation has been completed. The Extoll NIC receives the flush request and removes the TLB entries form the TLB. After the Extoll NIC ensured that none of the flushed translations is used by any other resource of the Extoll NIC a flush completion event is generated (Figure 5-9 (II)). The flush completion event is received by the event manager (Figure 5-9 (III)) which will wake up the hook function. After the hook function received the completion of the flush operation the hook function will return and therewith return the control back to the system. The standard Linux kernel has no support for the mentioned callback mechanism. Therefore a kernel patch either of Quadrics ([QUADRICS]) or Thomas Schlichter ([SCHLICH03]) must be applied to add the described callback functionality to the Linux kernel.

5.5.10 Poll/ePoll Support

As mentioned one of the biggest drawbacks of the Atoll design was that only polling has been supported. This means that every time a process was waiting for e.g. a new message the process had to poll in the replicator page. This busy polling wastes valuable CPU time that could be used by another process to make some progress. Therefore the Extoll will support a non busy waiting. If a process needs to wait for e.g. the arrival of a new message the process will go to sleep and the Extoll NIC, or more precisely the EDD, will wake up the thread on the arrival of a new message.

With the select/poll/epoll functions (see “Linux Select/Poll/Epoll” on page 129) Linux offers a standardised way to wait for actions on different resources. The idea behind this functions is to bundle several different file descriptors (e.g. sockets, files, futexes, ...) together and wait until something happens on at least one of the file desriptors. To integrate the ESS with the standard Linux system the ESS will also support the Linux poll mechanism.

The first thing that needs to be supported is a file descriptor, because the poll mechanism works only on file descriptors. This can be done without any extra effort because the user already obtained the file descriptor when a specific resource has been opened in user space. This also has the advantage that with the file descriptor automatically specifies the resource. The file descriptor that is passed to the user must not be closed by the user because this would lead to the release of the specified resource.

To support the poll mechanism the EDD must implement the FOPS poll function. The functionality of the FOPS poll function is to check for the specified file whether new actions can be performed or not (e.g. reading, writing, ...). If there are new actions for a file the FOPS poll function will return the corresponding actions via a bit mask. For a detailed description of the FOPS poll function refer to Linux Device Driver ([LDD]). The EDD will only support a wait functionality for VPs and VPGs. The condition when a VP can make progress is reached when the VP has a new notification in his NQ. To determine where to check for a new notification the read offset of the NQ is required. Unfortunately it is not possible to transfer this information via select/poll/epoll functions. Therefore the EDD must extract this information from the corresponding file descriptor. To solve the problem each VP has a so called config page. The config page is a single memory page that is associated with each VP. At the beginning of the config space an extoll_vp_t object (see Figure 5-10) is stored. The extoll_vp_t object represents all information that is necessary to manage a VP. The user **must** initialize and manage the extoll_vp_t object and is responsible to keep the data structures up to date. A kernel user is able to use the extoll_vp_t object directly while an user-level user needs to map the config page. In Figure 5-11 both cases are visualized.

```
typedef struct extoll_vp_t {
   extoll_vp_id_t id;
   extoll_queue_t sdr;
   extoll_queue_t rdr;
   extoll_queue_t ciq;
   extoll_queue_t nq;
   extoll_queue_t wdt;
   extoll_ptr_t   trigger_page;
} extoll_vp_t;
```

Figure 5-10 : Definition extoll_vp_t

Now the EDD has all necessary information to implement the poll functionality. When the EDD needs to check for a certain VP if there is a new notification the EDD will extract the current read_offset from the config page that is associated with each VP.

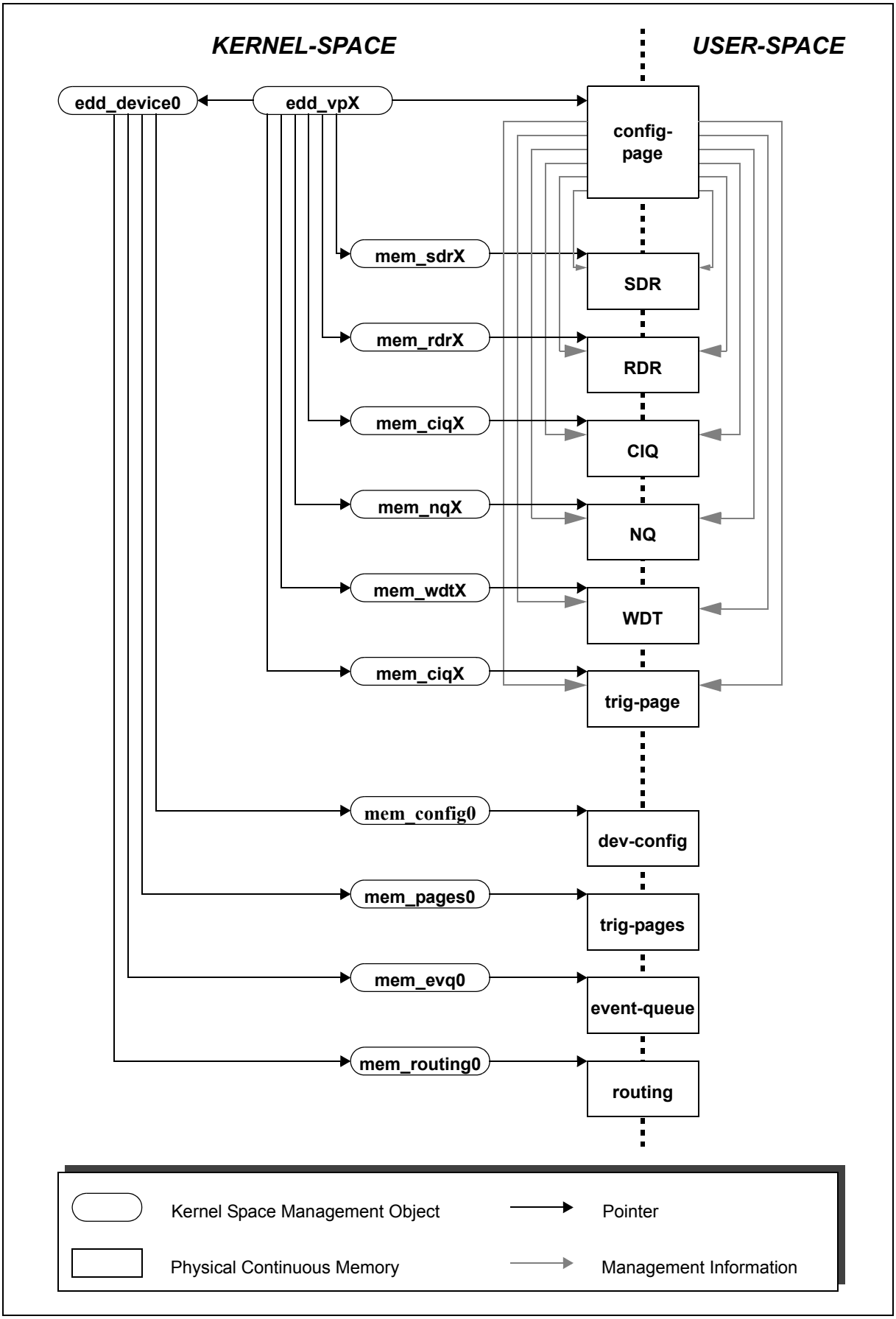


Figure 5-11 : extoll_vp_t Object with memory mapping

5.6 EUI

As mentioned the EUI is an abstraction layer that hides the differences of the kernel- and user-space access of the resources from normal users. The implementation is straight forward and consists mainly of a mapping of functions to kernel- and user-space access methods.

5.7 EMI

As mentioned earlier the EMI is an abstraction layer that hides the differences of the kernel- and user-space access to the resources for super users. The implementation is straight forward and consists mainly of a mapping to functions for kernel- and user-space access methods.

5.8 EPI

In this section the word event does not refer to the events that have been introduced in "Event System" on page 34. In this section the word event refers to EPI event ("EPI Events and Event Dispatcher" on page 107).

5.8.1 Structure

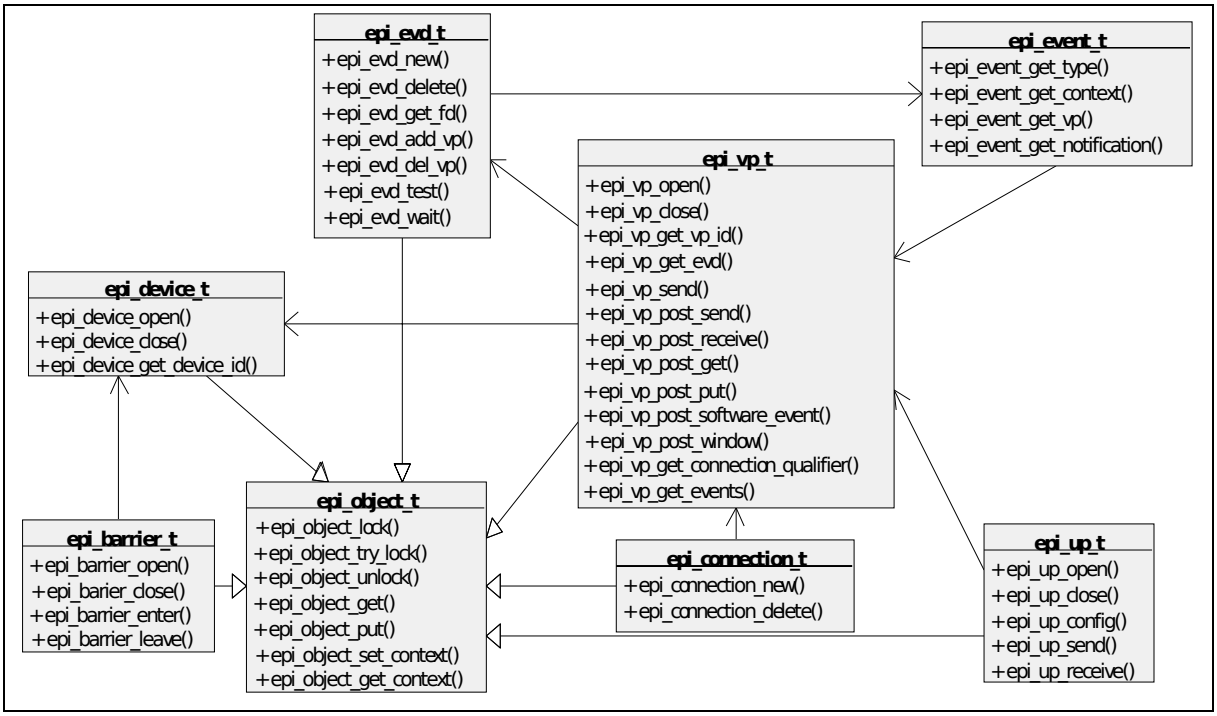


Figure 5-12 : Class Diagram of EPI

The structure of the EPI implementation is shown in Figure 5-12. As shown above there exists one object for each resource of the Exoll. The event and evd object are new and are described in "EPI Events and Event Dispatcher" on page 107. The EPI, like the whole Extoll, uses an

asynchronous communication model. The user starts an operation and the hardware will asynchronously perform the requested operation. When the execution of an operation has finished the user will be informed (see Figure 5-14).

5.8.2 EPI Events and Event Dispatcher

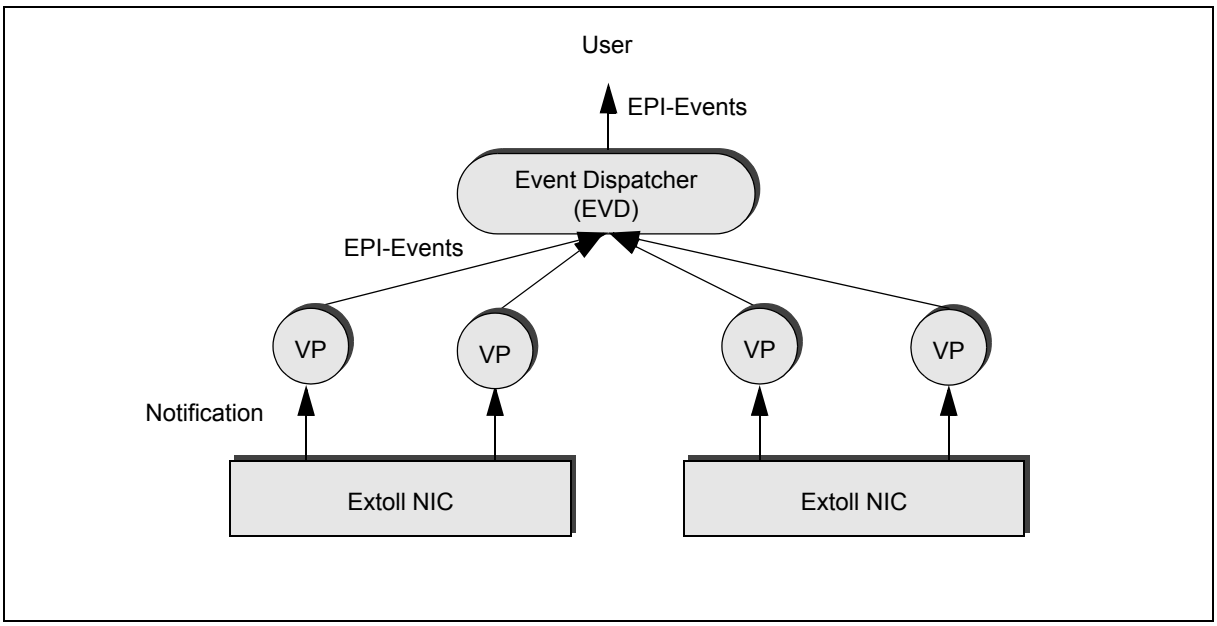


Figure 5-13 : EVD

Definition 5-1 : EPI Events. The EPI events are not related to the events that are generated by the Extoll for the EDD. The EPI Events are pure software events that consist of a notification, a context value and several management information. The EPI Events are events that are generated by the EPI layer to inform the user about completed operations (see Figure 5-13).

Definition 5-2 : Event Dispatcher (EVD). An event dispatcher is a software object that allows the user to bundle several VPs together and receive events from them.

As shown in Figure 5-13 the EVD is an object that allows the user to bundle several VPs together and receive events from them. If there are no events in the associated VPs it is possible to sleep on a EVD until at least one of the associated VPs has a new event. The introduction of the EVD is required to allow the user to have a single object to wait on for new events. If there was not an EVD the user would have to poll on all opened VPs in a round robin fashion. The user is able to receive a file descriptor that represents the EVD. This file descriptor can be used with the poll/epoll functions. This allows the user to wait simultaneously on different resources (e.g. futex, socket, file, ...) without polling. The EVD is also able to work on VPs of different NICs.

5.8.3 Polling and Waiting

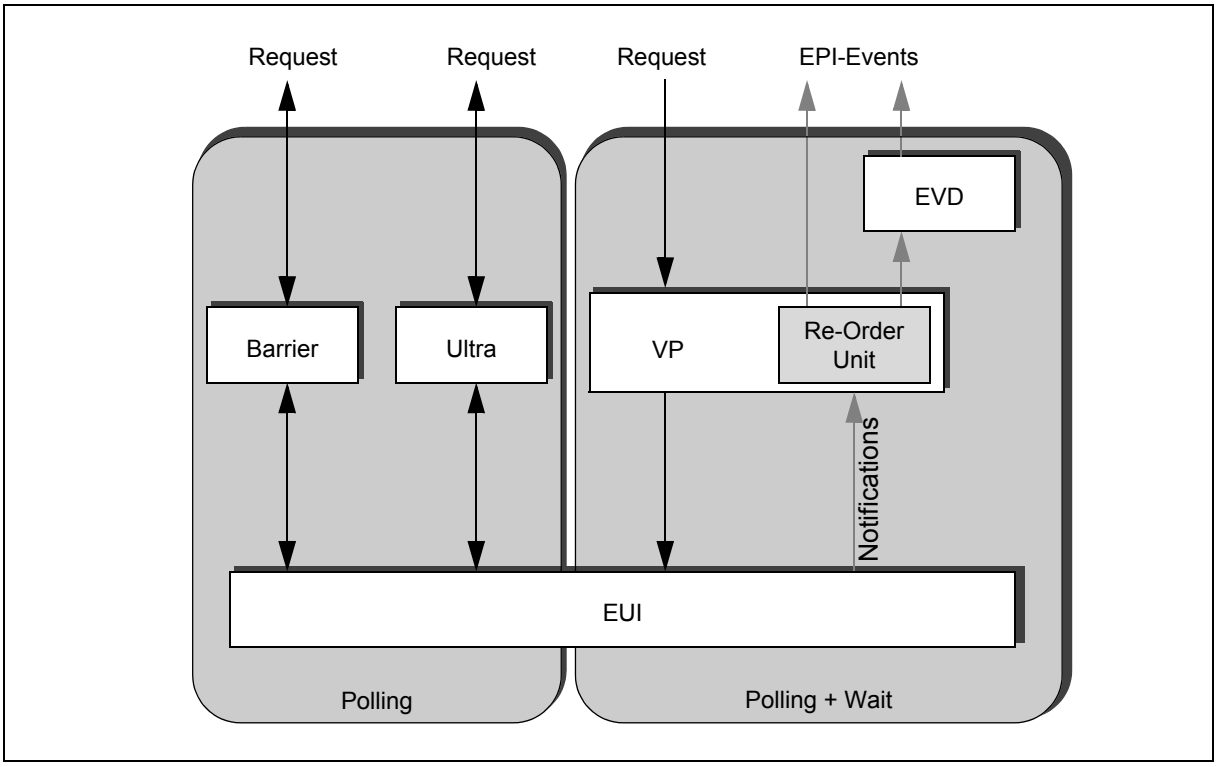


Figure 5-14 : EPI polling and waiting parts

As mentioned earlier the Extoll supports a polling and a waiting approach for detection of completed operations. As shown in Figure 5-14 barriers and Ultra ports only support the polling approach. This makes sense because both components are optimized for low latency. For VPs and EVDs it is possible to poll for new EPI events or to wait until there are new events available.

5.8.4 VP window management

Like the communication operations are the window management operation asynchronously. This means that the user posts a request operation and, later after the operation has completed, the user will receive a corresponding event. The following kind of window operations are available:

- **Window Create**. This operation allows the user to create a new memory window. The user is able specify the window ID of the window that should be created or specify EXTOLL_WINDOW_ID_ANY to create a window with the next available free ID. The user must not use the window until the corresponding window-created-event has been received.
- **Window Modify**. With window modify operation it is possible to change the configuration of an already opened window.
- **Window Delete**. The window delete operation frees the specified window. The user must not use the window after the corresponding delete operation has been posted.

For all operations it is important to flush the corresponding entry from the window descriptor cache to avoid wrong behaviour of the hardware in the case that there is an old version cached.

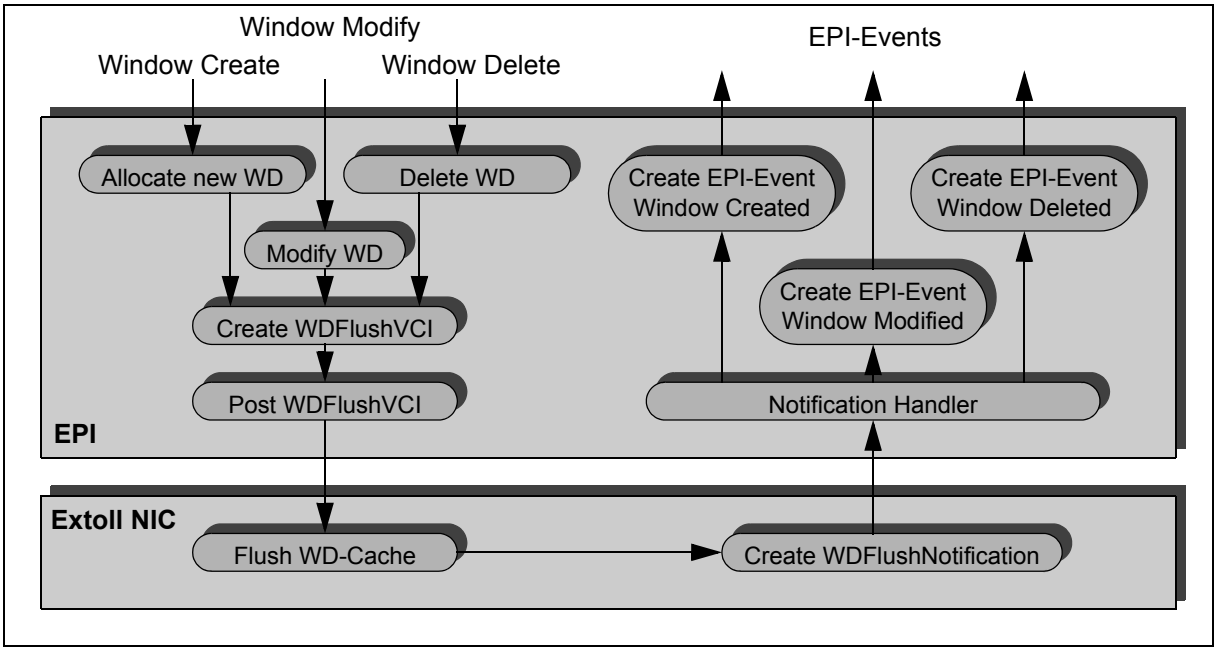


Figure 5-15 : Window Management

5.8.5 Thread Safety

Definition 5-3 : **Co-operative thread safety**. To avoid expensive locking operation inside the EPI the EPI supports co-operative thread safety. This means that EPI is per default not thread safe. If different threads want to access the EPI simultaneously each thread has to lock all objects that are passed via parameter to the functions that are called.

For performance reasons the EPI will only support co-operative thread safety.

5.8.6 Context Value

The context value is a user specific value (64 bit word) that can be associated with each EPI object and every communication operation that is triggered (excepted the Ultra port communication). This context value has no internal meaning and therefore can contain any value. The context value is a mechanism for the user to manage asynchronous communication. When for example a user starts a send operation the pointer of the corresponding request or connection object can be used as context value. After the operation is completed the user will receive the corresponding event and therewith the context value. This allows the user an efficient and fast matching of events to his own data structures.

Conclusion & Outlook

CHAPTER 6

6.1 Conclusion

The goal of this diploma thesis was to evaluate and design the software interface of the Extoll device by using the hardware-software co-design approach. As result of this thesis, a complete specification of the hostport software interface for the currently specified features has been designed. In parallel to the specification of the hostport software interface a complete software environment with the name Extoll Software Stack (ESS) has been developed. The ESS consists of seven specialized modules which are working in user- and kernel-space. The ESS modules can be grouped into the Device Driver Framework (EMM, ERM and EDD) and the Extoll Software Developer Kit (EUI, EMI and EPI).

The device driver framework supports all necessary functionality for management and maintains all of the Extoll resources currently specified. The Extoll SDK enables a user to access the Extoll resources either in user- or kernel-space. All the implemented features have been tested as far as the current development allowed. Each of the ESS modules is itself sub-divided into smaller modules. This modular approach makes the whole ESS very stable, maintainable and flexible. Therefore the ESS modules can be easily adopted and extended to future Extoll features.

Name	*Core*	*Examples*	*Tests*	*Sum*
COMMON	3610	90	170	3870
ELS	1200	180	310	1690
EMM	2590	170	280	3040
ERM	3020	260	21140	24420
EDD	9110	-	60	9170
EUI	3780	1990	790	6560
EMI	3660	1240	150	5050
EPI	4810	-	810	5620
Sum	31780	3930	23710	59420

Table 6-1: Quantitative Analyse of the ESS (lines of code)

The whole ESS consists of approx. 59000 lines of code[1] (see Table 6-1). Despite the fact that the code has been written in C, the whole design follows the object oriented design paradigms. The resulting libraries are able to be used in C/C++ programs. The whole code has been documented with an in-line API documentation system ([DOXYGEN]).

1. Measure with the word count utility wc.

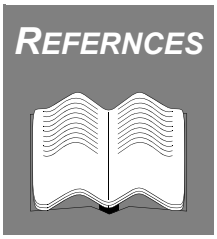
6.2 Outlook

As mentioned before this thesis covers only the Extoll resources that are currently specified. There are still some features under active development which need to be integrated. This applies to the whole cache system of the Extoll. The design and implementation of the Extoll cache system is part of the diploma thesis of Felix Rembor ([REMBOR06]). Therefore the mechanism of the address translation management are still missing in the EDD implementation. In the case of a process crash the EDD takes care that no resources are lost, but the detection of a process crash and the final cleanup are still missing in the implementation.

Due to the of lack of existing hardware or software simulation the communication mechanism in the ESS could not be tested. Therefore more testing of the communication mechanism is required at the time the necessary environments are available.

The ESS already covers a lot of the tasks that have been done by the former Atoll Daemon (e.g. routing management). Therefore an evaluation is required if an Extoll Daemon is still required and which tasks have to be performed by this daemon.

References



[ATOLL] The Atoll Homepage; http://*www.atoll-net.de*.

[ATOLL99] Lars Rzymianowicz, Ulrich Brüning, Jörg Kluge, Patrick Schulz and Mathias Waack; *ATOLL: A Network on a Chip*; 1999.

[ATOLL00] Markus Fischer, Ulrich Brüning, Jörg Kluge, Lars Rzymianowicz, Patrick Schulz and Mathias Waack; *ATOLL, a new switched, high speed Interconnect in Comparison to Myrinet and SCI*; 2000.

[ATOLL02] Ulrich Brüning, Holger Fröning, Patrick R. Schulz, Lars Rzymianowicz; *ATOLL: Performance and Cost Optimization of a SAN Interconnect*; 2002.

[ATOLL03] David Slogsnat, Patrick R. Haspel, Holger Froening and Ulrich Bruening; *ATOLL: Performance and Cost Optimization of a SAN Interconnect*; 2003.

[CSB] Lambet Schaelike, Alan L. Davis; *Design Tradeoff for User-level I/O Architectures*;

[DAT] Direct Access Transport Collaborative Specification; http://www.dat-collaborative.org.

[DOXYGEN] Doxygen documentation system Homepage; http://www.doxygen.org.

[DREPPER05] Ulrich Drepper; *Futexes are Tricky*; http://people.redhat.com/~drepper/futex.pdf; 2005.

[ELAN] Quadrics Ltd.; *Elan Programming Manual*; http://www.quadrics.com; 2005.

[FELDNER04] Ingo Feldner; *High Level Executable Specification Development of a high performance SAN chip*; Diploma Thesis, Computer Architecture Group at the Department of Computer Engineering, University of Mannheim; 2004.

[FOWLER] Martin Fowler; *UML Distilled Third Edition. A Brief Guide to the Standard Object Modelling Language*; Addision Wesley; 2003.

[FRANGER04] Dirk Franger; *A Multi-Context Engine for Remote Memory Access to Improve System Area Networking*; Diploma Thesis, Computer Architecture Group at the Department of Computer Engineering, University of Mannheim; 2004

[GCC] GNU Compiler Collection Homepage; http://gcc.gnu.org

[IBTA] Infiniband Trade Association Homepage; http://www.infinibandta.org.

[LDD] Jonathan Corbet, Alessandro Runini, Greg Kroah-Hartman; *Linux Device Drivers*; O'Reilly; 2005.

[LITZ05] Heiner Litz; *Advance hardware communication techniques*; Diploma Thesis, Computer Architecture Group at the Department of Computer Engineering, University of Mannheim; 2005.

[MAURER] Wolgang Maurer. *Linux Kernelarchitectur.* Hanser 2004.

[MPIFORUM] The MPI Standard; http//www.mpi-forum.org.

[MPI2] MPI-Forum; *The MPI-2 Standard*; Homepage http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.htm; 1997.

[MX] Myricom; *Myrinet Express (MX): A High-Performance, Low-Level, Message-Passing Interface for Myrinet*; http://www.myricom.com; 2005.

[MYRICOM] Myrinet Inc. Homepage; http://www.myrinet.com.

[NUS03] Mondrian Nuessle; *Design and Implementation of a distributed management system for the ATOLL high-performance network*; Diploma Thesis, Computer Architecture Group at the Department of Computer Engineering, University of Mannheim; 2003.

[PERF05] Holger Fröning, Mondrian Nüssle, David Slogsnat, Patrick R. Haspel, Ulrich Brüning; *Performance Evaluation of the ATOLL Interconnect*; 2005.

[PFISTER] Gregory F. Pfister; *An Introduction to the InfiniBand Architecture*.

[POSIX] *The POSIX standard*; http://www.opengroup.org/onlinepubs/009695399/toc.htm.

[QUADRICS] Quadrics Inc. Homepage; http://www.quadrics.com.

[REMBOR06] Felix Rembor; *Exploration, Development and Implementation of different TLB Function and Mechanism*; Diploma Thesis, Computer Architecture Group at the Department of Computer Engineering, University of Mannheim; expected 2006

[RZY97] Lars Rzymianowicz; *Designing Efficient Network Interfaces for System Area Networks*; Dissertation Thesis, Computer Architecture Group at the Department of Computer Engineering, University of Mannheim; 1997.

[SCHLICH03] Thomas Schlichter; *Exploration of Hard- and Software requirements for one-sided zero copy user-level communication and its implementation*; Diploma Thesis, Computer Architecture Group at the Department of Computer Engineering, University of Mannheim; 2005.

[SOH05] Richard Sohnius; *Creating an Executable Specification Using SystemC of a High Performance, Low Latency Multilevel Network Router*; Diploma Thesis, Computer Architecture Group at the Department of Computer Engineering, University of Mannheim; 2005.

[SPONER05] Timo Sponer. *Development, Verification and Integration of a Processing Unit in the Communication Function of a SAN Device in SystemC*; Diploma Thesis, Computer Architecture Group at the Department of Computer Engineering, University of Mannheim; 2005.

[TANENBAUM] Andrew S. Tanenbaum; *Operating System - Design and Implementation*;

[TOP500] The TOP 500 Homepage; http://www.top500.org.

[STE98] Richard Stevens. *UNIX Network Programming, Volume 1, Second Edition: Networking APIs: Sockets and XTI*. 1998.

[STORK05] Sven Stork. *ESS Reference Manual*. Internal Documentation, Computer Architecture Group at the Department of Computer Engineering, University of Mannheim; 2005.

[UMPI2] William Gropp, Ewing Lusk, Rajeev Thakur. *Using MPI-2 - Advanced Features of the Message Passing Interface*. The MIT Press. 1999.

[UML] James Rumbaugh, Ivar Jacobson, Grady Booch; *The Unified Modelling Reference Manual*; Addision Wesley; 1999.

[VERBS] Mellanox Technologies; *Mellanox IB-Verbs API*; http:// www.mellanox.com; 2001

Glossary

CHAPTER B

API

API is an acronym “Application Programming Interface”.

ATOLL

ATOLL is an acronym for “Atomic Low Latency”. The ATOLL NIC belongs to the family of SANs and is the direct predecessor of the Extoll NIC.

BSD

BSD is an acronym for “Berkley Software Distribution”.

CIQ

CIQ is an acronym for “Communication Instruction Queue”.

Cluster

A cluster consists of several independent nodes that are connect by a SAN. Computation task are distributed across the nodes of a cluster to increase the over all computation power.

CSB

CSB is an acronym for “Conditional Store Buffer”.

DAPL

DAPL is an acronym for “Direct Access Programming Library“.

DTO

DTO is an acronym for “Data Transfer Operation“.

EDD

EDD is an acronym for “Extoll Device Manager”.

ELS

ELS is an acronym for “Extoll Logging System”.

EMI

EMI is an acronym for “Extoll Management Interface”.

EMM

EMM is an acronym for “Extoll Memory Manager”.

EP

EP is an acronym for “Endpoint”.

EPI

EPI is an acronym for the “Extoll Programming Interface”. The EPI is the successor of the ATOLL-Palms.

EPU

EPU is an acronym for “Extoll Processing Unit”.

EUI

EUI is an acronym for “Extoll User Interface”.

ESS

ESS is an acronym for “Extoll Software Stack”.

ETF

ETF is an acronym for “Extoll Testing Framework”.

Event

An event is an well defined descriptor that is generated by the Extoll NIC to provide the device driver with information.

Extoll

Extoll is an acronym for “Extended ATOLL”. The Extoll NIC belongs to the family of SANs.

FU

FU is an acronym for “Function Unit”.

GM

GM is the acronym for “Glenn’s Messages”. GM is an older API for the Myricom SANs.

HCA

HCA is an acronym for “Host Channel Adapter”.

HT

HT is an acronym for “Hyper-Threading”.

IBA

IBA is an acronym for “Infiniband Architecture”.

IPoExtoll

IPoExtoll is a network device driver that tunnels IP packets through the Extoll SAN.

ISV

ISV is an acronym for “Independent Software Provider”.

LRU

LRU is an acronym for “Least Recently Used”.

MMU

MMU is an acronym for “Memory Management Unit”. A hardware resource that translate virtual addresses into physical addresses.

MPI

MPI is an acronym for “Message Passing Interface”.

MX

MX is an acronym for “Myrinet Express”. MX is the most current API for Myricom SAN solutions.

Notification

A notification is an well defined descriptor that is generated by the Extoll NIC and stored in the NQ of a VP.

NIC

NIC is an acronym for “Network Interface Controller”.

NQ

NQ is an acronym for “Notification Queue”.

PIO

PIO is an acronym for ”Programmed Input/Output”.

PRQ

PRQ is an acronym for “Posted Receive Queue”.

QP

QP is an acronym for “Queue Pair”.

RDR

RDR is an acronym for “Receive Data Region”.

SDR

SDR is an acronym for :Send Data Region”.

SAN

SAN is an acronym for “System Area Network”.

TLB

TLB is an acronym for “Translation Lookaside Buffer”.

ULTRA

ULTRA is an acronym for “Ultra Low Latency Transaction“.

UML

UML is an acronym for “Unified Modelling Language” and/or “User Mode Linux”.

User-Level-Communication

In the case of a user-level communication a process is able to trigger network operations directly by accessing some resources of the hardware instead of trapping into the operating system.

VCI

VCI is an acronym for “Virtual Communication Instruction“.

VP

VP is an acronym for “Virtual Port”.

VPD

VPD is an acronym for “Virtual Port Descriptor“.

VPID

VPID is an acronym for “Virtual Port Identifer“.

WD

WD is an acronym for “Window Descriptor“.

WDT

WDT is an acronym for “Window Descriptor Table”. The WDT is organized as an array of window descriptors.

Coding Style

CHAPTER C

C.1 Coding Styles

To avoid mistakes during the implementation and usage of the ESS libraries/modules the following coding-style rules have been introduced. These rules help developers and users of the ESS to become fast familiar with the modules and their usage. Some of these rules are necessary to guaranty the collaboration with other libraries and user applications.

C.2 Naming convention

- Each function must be prefixed with "<$MODULE>_<$OBJECT>" (see Figure C-1). The module specifier is the module the function belongs to (e.g. els, epi or extoll for the global scope). The object identifies the object that is manipulated by this function (e.g. vp, device, ...). The whole function name must be only consist of lower-case letters and underscores. This is necessary to avoid conflicts with already existing functions of existing software modules.

```
extoll_error_t epi_vp_open( ... );
extoll_error_t extoll_queue_empty( ... );
extoll_error_t els_log_open( ... );
```

Listing C-1 : Examples of function names

- All object are expressed by an structure or enum. Like for the function each object has the following naming convention <$MODULE>_<$NAME>_t. The module and name have the same meaning as for functions. The whole name must only consist of lower case characters and under scores. To simplify the usage all objects must have a *typedef* to their own name (see Listing C-2).

```
typedef struct epi_vp_t {
    <$MEMBER>
    <$MEMBER>
    <$MEMBER>
}epi_vp_t;

typedef enum extoll_barrier_id_t {
    EXTOLL_BARRIER_ID_00  = 0x00,
    EXTOLL_BARRIER_ID_01  = 0x01,
    EXTOLL_BARRIER_ID_02  = 0x02,
    EXTOLL_BARRIER_ID_03  = 0x03,
    EXTOLL_BARRIER_ID_04  = 0x04,
    EXTOLL_BARRIER_ID_05  = 0x05,
    EXTOLL_BARRIER_ID_06  = 0x06,
    EXTOLL_BARRIER_ID_07  = 0x07
} extoll_barrier_id_t;
```

Listing C-2 : Examples of object names

- All enum values or defines are prefixed with <$MODULE_>_<$OBJECT> and must only consist of capital characters and underscores. In the case of enums at least the first value must have a value assigned. In general ever value should have a value assigned (see Listing C-2).
- All environment variables that are used to control the behaviour are prefixed with “EXTOLL_” and must only contain capital letters and underscores (e.g. EXTOLL_LOG_LEVEL).
- All ordinary variable types[1] must have a typedef to an unique name that is prefixed with “extoll_”. This is necessary to avoid problems between the 32 Bit and 64 Bit architectures when different data types can have different sizes.

C.3 Source Code

- At the beginning of each file a copyright message must be present (see Listing C-5).
- All header files must encapsulated their context in an extern “C” extern statement. This is necessary to compile and link the ESS modules with C++. For readability reasons the “BEGIN_C_DECLS” and “END_C_DECLS“ macros have been defined (see Listing C-5).
- Each function must have a documentation header that describes the behaviour and the parameters of the function (see Listing C-3).

1. e.g. int, char, ...

```
/**
 * <$FUNCTION_DESCRIPTION>
 *
 * @param <$PARAM_NAME> <$PARAM_DESCRIPTION>
 * @param <$PARAM_NAME> <$PARAM_DESCRIPTION>
 * @param <$PARAM_NAME> <$PARAM_DESCRIPTION>
 * @return <$RETURN_VALUE_DESCRIPTION>
 */
```

Listing C-3 : Comment header template

- All source code files must have an indention 4. The indention is made by spaces and not by tabs to avoid problems with different editors.
- The return value of each function call must be checked for an error. This check can be omitted if the function has no return value or always returns success.

```
ex_err = eui_vpg(&vpg);
if ( ex_err != EXTOLL_SUCCESS ) {
    // do error handling
}
ex_err
```

Listing C-4 : Examples of correct error checking

- All header files are accessible by “extoll/<$MODULE>/<$HEADER>” (e.g. “extoll/common/extoll_list.h”).
- A template for a header file is shown in Listing C-5.

```
/************************************************************************
*                                                                       *
* (C) <$YEAR>, <$AUTHOR>, Computer Architecture Group,                  *
* University of Mannheim, Germany                                       *
*                                                                       *
* This program is free software; you can redistribute it and/or modify *
* it under the terms of the GNU General Public License as published by *
* the Free Software Foundation; either version 2 of the License, or     *
* (at your option) any later version.                                   *
*                                                                       *
* This program is distributed in the hope that it will be useful,       *
* but WITHOUT ANY WARRANTY; without even the implied warranty of        *
* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the         *
* GNU General Public License for more details.                          *
*                                                                       *
* You should have received a copy of the GNU General Public License     *
* along with this program; if not, write to the Free Software           *
* Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307 *
* USA                                                                   *
*                                                                       *
************************************************************************/

#include <extoll/common/extoll_defines.h>

#ifndef __EXTOLL__<$MODULE>_<$FILENAME>__
#define __EXTOLL__<$MODULE>_<$FILENAME>__

BEGIN_C_DECLS

<$INCLUDES>

<$DATATYPES>

<$PROTOTYPES>

END_C_DECLS

#endif /* !__EXTOLL__<$MODULE>_<$FILENAME>__ */
```

Listing C-5 : Header template

C.4 General

- All variables, structures and memory blocks must be initialised before they are used.
- The order of the parameters should be first the IN parameters then the OUT parameters. IN/OUT parameters are handled as OUT parameters.
- Inline functions should be used instead of macros, because in the case of inline functions the compiler is able to make type checking on the parameters and the return value while a macro is only a text replacement of the pre-processor. This helps to detect and to avoid mistakes already during the compilation.

- To reduce the name pollution as much as possible functions and global variables that do not belong to the interface must not be visible outside of the corresponding component.
- All function have to return an error of the type *extoll_error_t*.
- All objects that must have a certain size (e.g. descriptors) must be marked with the packed attribute ([GCC]) to ensure that the compiler will not optimize the memory layout of the structure (see Listing C-6).

```
typedef struct __attribute__((packed)) extoll_window_descriptor_t {
    <$MEMBERS>
} extoll_window_descriptor_t;

typedef struct __attribute__((packed)) extoll_uint48_t {
    extoll_uint64_t value:48;
} extoll_uint48_t;
```

Listing C-6 : Examples of a packed objects

- Instead of writing one big function that performs a big number of actions it is better to split a huge function into several smaller functions that only perform one specific task.
- As long as there is no need because of timing problems, the code should be optimized for readability, because code is written once but read several times.

Linux Select/Poll/Epoll

CHAPTER D

D.1 Motivation

In real world applications the user needs to read/write data from a file or socket. To perform this operations the user will call the standard posix read/write functions. These functions can work in a blocking or in a non-blocking mode. In the best-case these functions immediately transfer the requested amount of data before they return. The difference occurs in the case when there are not enough resources to transfer the whole requested amount of data. In the blocking mode the process will sleep inside the function call until the whole amount of data has been transferred. In the non-blocking case the function call will return if the process would need to wait.

.

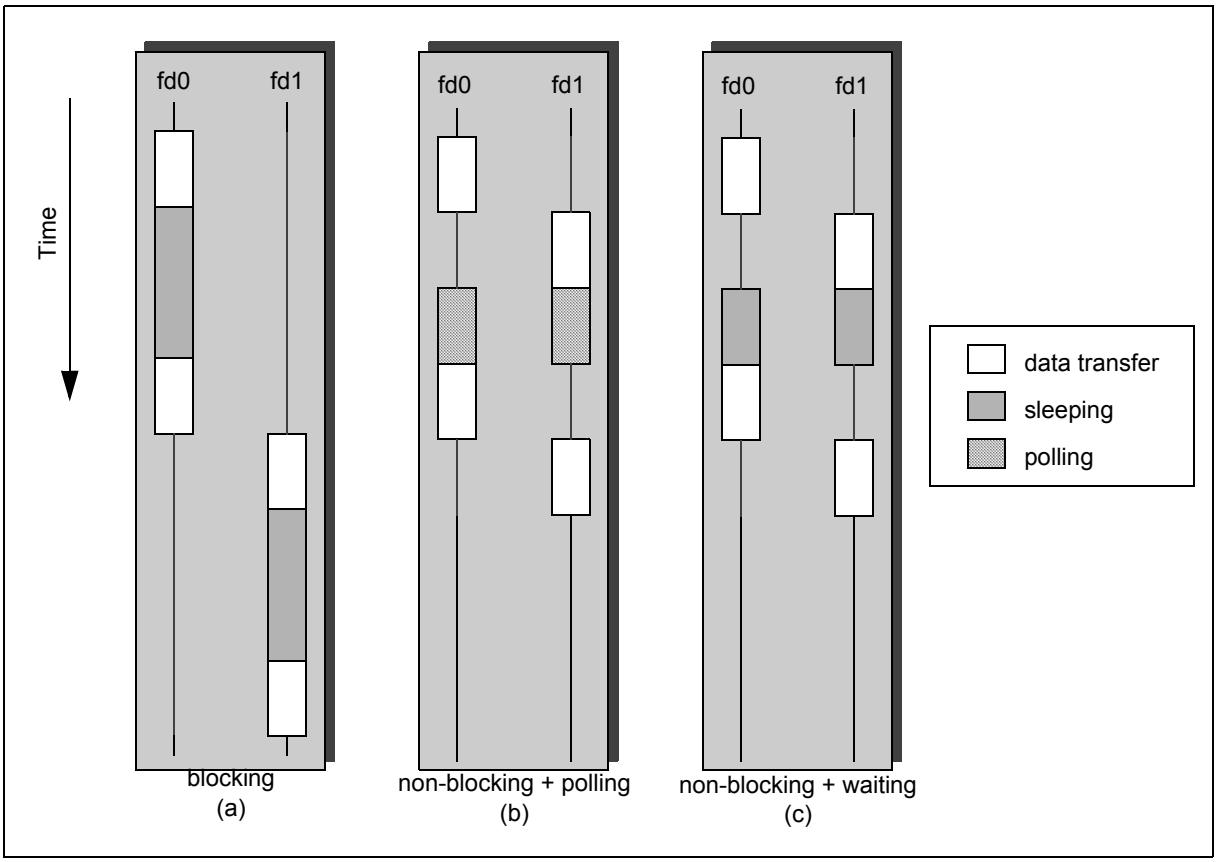


Figure D-1 : blocking vs. non-blocking

In the case that a user works with only one file descriptor there is no real problem with this semantic. The problem occur in the case when a process needs to transfer data on different file descriptors. If the user would use the blocking approach the user would need to wait until the whole data had been transferred on the first file descriptor before he could transfer data on a second descriptor. This is illustrated in Figure D-1 (a).

In the non-blocking case the user may transfer as much as possible data on the first file descriptor until the function would block, but instead of going to sleep the user could start transferring data on the second file descriptor until this file descriptor also would cause the function to block. If the user is not able to transfer data on any file descriptor the user would have to poll on all file descriptors to figure out when and which descriptor becomes ready next. This will lead to the case that the CPU does an active polling which waste a lot of CPU power. This behaviour is shown in Figure D-1 (b).
The blocking approach has the drawback that the process wastes lot of time sleeping in a function call on the first file descriptor while the user could transfer data on another file descriptor. The non-blocking approach solves this problem by introducing the problem, while at the same time the problem of active pollins is introduced in the case the that there are no descriptors ready for a data transfer. The behaviour that would be desired is show in Figure D-1 (c). The user tries to transfer as much data as possible on each file descriptor and when all file descriptors are busy the user decides to go to sleep until one of the file descriptors becomes ready again. For this purpose the select, poll and epoll mechanisms where introduced. To avoid that a process needs to wait forever all these mechanisms support timeout value.

D.2 Classical Approaches

D.2.1 Select

Select is the oldest mechanism that has been introduced to solve the described problem. The idea is that the user is interested in the state (or changing of state) of a set of file descriptors. The user creates a file descriptor set and adds the file descriptors into this set. As shown in Listing D-2 the select call supports 3 different kind of file sets, one for reading events, one for writing events and one for other events (e.g. connection events).

```
int select
(
    int             n,
    fd_set         *readfds,
    fd_set         *writefds,
    fd_set         *exceptfds,
    struct timeval *timeout
)
```

Listing D-1 : Prototype Select

D.2.2 Poll

The *poll* system call is a newer implementation that uses only one file descriptor set for all file descriptors. For each file descriptor the user can specify the events that should be observed (see Listing D-2).

```
int poll
(
    struct pollfd *ufds,
    unsigned int   nfds,
    int            timeout
)
```

Listing D-2 : Prototype Poll

D.2.3 Drawbacks

The drawback with both approaches is that both mechanism do not scale very well for a large amount of file descriptors. This comes from the fact that in both cases for every function call the kernel must copy all the information from the user-space to the kernel space and prepare the data structures for waiting. This must happen every time even when the user wants to wait for a file descriptor set even if there was no change in the file descriptor set. Because the amount of data that must be exchanged is linear to the amount of file descriptors this approach scales very bad for a huge amount of file descriptors.

D.3 New Approach

D.3.1 EPoll

The innovation of the new approach is to avoid the drawback of the old systems by splitting the mechanism in 2 parts. The managing and the wait functionality (see Listing D-3). The epoll system offers the possibility to modify the file descriptor set in the kernel with the *epoll_ctl* function. To wait on the file descriptor set the epoll system offers the *epoll_wait* function. Because the file descriptor set is pre-build in the kernel the *epoll_wait* function does not need to exchange all the descriptor information. In the case that the file descriptor set does not change the user needs only to call the wait function. In this case the wait is much more efficient.

```
int epoll_ctl
(
    int                 epfd,
    int                 op,
    int                 id,
    struct epoll_event *event
)
```

```
int epoll_wait
(
    int  epfd,
    struct epoll_event *events,
    int                 maxevents,
    int timeout
)
```

Listing D-3 : Prototypes of the epoll functions

A special feature of the epoll object is that it has itself a file descriptor that is useable inside epoll object (see Figure D-2).

.

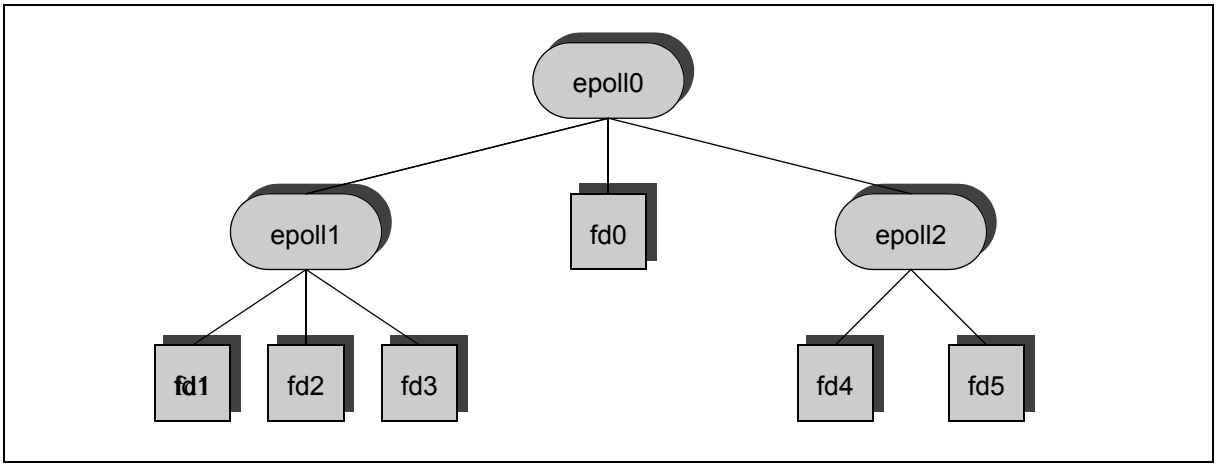


Figure D-2 : Epoll System

D.4 Driver Support

To support each of the described mechanisms the driver needs to support the “poll” function of the file operations structure (see Listing D-4).

```
unsigned int (*poll)
(
    struct file              *filp,
    struct poll_table_struct *table
)
```

Listing D-4 : Prototype FOPS Poll function

The implementation of this function has to check which kind of operation on the specified file will not block. Then the function must return a bitmask of all possible operations that can be performed on the file without blocking. Independent of the current state of the file the function also must specify a wait queue where the process can sleep on([LDD]).

Extoll Tools

CHAPTER E

E.1 extoll-config

E.1.1 Description

The *extoll-config* program offers access to all information that is necessary for compilation and linking to Extoll Software Stack components.

E.1.2 Parameters

Parameter	*Description*
--help	Display all available parameters.
--version	Display the current version of the installed ESS.
--prefix	Display the installation prefix directory of the installed ESS.
--cflags	Display the C/C++ flags that are necessary to compile modules/programs that used the ESS.
--els-libs	Display the C/C++ flags to link a programs against ELS.
--eui-libs	Display the C/C++ flags to link a programs against EUI.
--emi-libs	Display the C/C++ flags to link a programs against EMI.
--epi-libs	Display the C/C++ flags to link a programs against EPI.

Table E-1: Parameters extoll-config

E.1.3 Possible Error

none.

E.2 extoll_mknod

E.2.1 Description

This program creates all device files that are necessary for usage of the ESS. Before this program can be executed the EDD module must be loaded to obtain the corresponding major number. Per default the *extoll_mknod* creates device files for 4 NICs. Before the new device files are created possible old files will be removed.

E.2.2 Parameter

None.

E.2.3 Possible Errors

- The EDD module has not been loaded and therefore the program could not determine the major number.
  Solution : Load the EDD module.
- The device files cannot be deleted and/or created because of insufficient permission of the user that executed the program.
  Solution : The program must be executed by a user that has enough permissions to perform the deletion/creation of device files.

E.3 extoll_modules

E.3.1 Description

This program loads and unloads the Extoll Software Stack kernel modules.

E.3.2 Parameter

Name	*Description*
load	Load the kernel modules.
unload	Unload the kernel modules.

Table E-2: Parameter extoll_modules

E.3.3 Possible Errors

- The user that called the program does not have enough rights to load/unload modules.
  Solution : The program must be executed by an user with sufficient rights to load/unload kernel modules.

E.3.4 Example

```
linux:~ #extoll_modules load
Load els ..................................................... [OK]
Load emm ..................................................... [OK]
Load erm ..................................................... [OK]
Load edd ..................................................... [OK]
Load eui ..................................................... [OK]
Load emi ..................................................... [OK]
Load epi ..................................................... [OK]
linux:~ #
linux:~ #extoll_modules unload
Unload els ................................................... [OK]
Unload emm ................................................... [OK]
Unload erm ................................................... [OK]
Unload edd ................................................... [OK]
Unload eui ................................................... [OK]
Unload emi ................................................... [OK]
Unload epi ................................................... [OK]
linux:~ #
```

Listing E-1 : Example of extoll_modules

E.4 extoll_info

E.4.1 Description

This program displays the configuration of the currently available devices in the system.

E.4.2 Parameters

None.

E.4.3 Possibly Errors

- The program cannot query the information because the EDD modules is not loaded.
  Solution : Load the EDD modules.
- The program is executed with insufficient rights.
  Solution : Execute the program as user with sufficient rights.

E.4.4 Example

```
linux:~ # extoll_info
Found 2 devices

Extoll
|
|--Device:
|   |-- Device Number      : 0
|   |-- Device ID          : [0xcafe0000]
|   |-- Virtual Port Count : 64
|   |-- Barrier Count      : 8
|   |-- Routing Table Len  : 4096
|   |-- Routing Slot Len   : 128
|   |-- Routing Slot Count : 4
|   |-- CIQ len            : 4096
|   |-- WDT len            : 4096
|   |-- SDR len            : 4096
|   |-- RDR len            : 4096
|   |-- NQ len             : 4096
|   |-- VP lower bound     : 0
|   +-- VP upper bound     : 24
|
+--Device:
    |-- Device Number      : 1
    |-- Device ID          : [0xcafe0001]
    |-- Virtual Port Count : 64
    |-- Barrier Count      : 8
    |-- Routing Table Len  : 4096
    |-- Routing Slot Len   : 128
    |-- Routing Slot Count : 4
    |-- CIQ len            : 4096
    |-- WDT len            : 4096
    |-- SDR len            : 4096
    |-- RDR len            : 4096
    |-- NQ len             : 4096
    +-- VP lower bound     : 0
```

Listing E-2 : Output of extoll_info

E.5 extollctl

E.5.1 Description

This program is used to query and change the runtime parameters of the Extoll Software Stack kernel modules.

E.5.2 Paramters

Parameter	*Description*
-w variable=value	Modify a runtime variable.
-a	Display all available runtime variables.

Table E-3: Parameters of extollctl

E.5.3 Possibly Errors

- The user has not enough rights to access/modify the runtime parameters.
  Solution : Execute the program as user with sufficient rights.
- There are no ESS components loaded.
  Solution : Load ESS components.

E.5.4 Example

```
linux:~ # extollctl -a
emm.config.mem_min_order = 0
emm.config.mem_max_order = 11
emm.config.policy = simple
emm.config.els_priority = ELS_PRIORITY_NONE
erm.config.slots_per_target = 4
erm.config.max_table_count = 4
erm.config.els_priority = ELS_PRIORITY_NONE
linux:~ #
```

Listing E-3 : Output of extollctl

Declaration of Honour

I assure that this diploma thesis came into being without the help of a third person and without the use of other sources and tools and that the used sources, if literal or in content, are marked.

The diploma thesis has not been submitted in this or in any other form to any authority of examination.

I am aware that a misstatement will have legal effects.

Mannheim, 11 January 2006

Sven Stork