ATM Firmware for DROPS

Uwe Dannowski
Uwe.Dannowski@inf.tu-dresden.de

Dresden University of Technology

July 1999

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

The bottleneck in current PCI-based servers is no longer CPU performance but I/O. Current server systems transfer data from the source I/O device to main memory and from there to the destination I/O device. Thus all data passes the PCI host bridge at least two times — usually more often for activities such as checksumming.

The Dresden Real Time Operating System (DROPS) project aims at providing applications with "Quality of Service" (QoS) support from the operating system. To this end, a multiserver environment based on the L4 μ-kernel is being developed. The DROPS project attempts to develop design techniques for the construction of distributed real time systems where each component can guarantee a certain quality of service to applications. From the start of the DROPS project on, memory usage, CPU usage and interrupt latency were the aspects to give guarantees for. Recent research [Sch98] has shown noticeable influences from I/O activity on these three aspects. There is a demand of predictable I/O operations as well as a need for optimized data paths.

L4ATM together with the PCA-200E driver using FORE's firmware offer ATM connectivity to DROPS applications. Even under heavy load, like the AIM multi-user benchmark running under L4Linux, this protocol stack can guarantee promised connection characteristics for real-time connections. But, that setup does not behave well with unexpected heavy traffic arriving from the ATM network. In that case CPU utilization increases although packets are probably dropped by L4ATM due to buffer shortage. Even worse, real-time connections may experience data losses because of receive queue congestion or receive buffer outage in the PCA-200E driver and firmware.

For the DROPS networking components, especially for the PCA-200E ATM device driver, an advanced firmware addresses some of the issues. Expected benefits are:

- per virtual channel (VC) buffering
  Current firmware provides only two distinct receive buffer pools. Receive buffers are allocated in a hardly predictable order forcing the system to do copy/map operations. Per VC buffering would allow a zero copy receive path up to the application. The effects of buffer overruns due to non conforming connections would be limited to the single connection.

- data routing decisions in firmware
  Having per VC buffering, it is possible to locate receive buffers either in host memory or in PCI attached memory, depending on the further processing steps.

- drop unwanted cells earlier
  Transferring knowledge of connection characteristics (traffic parameters) to the firmware moves the point where non conforming data can be discarded from the device driver to the firmware. This saves valuable PCI/host memory bandwidth and CPU cycles.

- reduced host resource usage
  By reducing and limiting the usage of the hosts resources like memory bandwidth and CPU cycles, predictability can be improved.

- priorities
  Real-time connections should have priority over non real-time connections. That way, without much effort both connection types can be used concurrently.

This work aims at providing a firmware for the PCA-200E ATM network adapter that provides best service by utilizing a minimum of the host's resources. The result should be a fast, flexible and maintainable firmware that suits the DROPS project.

1.2 Synopsis

This thesis is organized in six main chapters. Following this introduction, Chapter 2 briefly describes the terms being required for understanding this work. Furthermore, related projects are introduced in that chapter. In Chapter 3 the design goals are presented, a process model is derived from that and the different processes are described. Chapter 4 points out certain aspects of the implementation and discusses performance. Finally, Chapter 5 summarizes this work and discusses the scope for further work.

Chapter 2

Basics

An extremely short and surely incomplete overview of ATM is given in the first section of this chapter. Thereafter, a few related projects are introduced: the PCA-200E drivers for Linux and DROPS, the L4 μkernel, and the DROPS project and its ATM component L4ATM. This chapter closes with a description of the PCA-200E hardware.

2.1 What is ATM ?

Of course, this section is not intended to be a complete description of ATM, but it tries to give a short explanation of the terms used in this work.

Asynchronous Transfer Mode (ATM) is a connection-oriented packet switching network technology based on asynchronous time division multiplexing. In contrast to traditional networks, ATM networks can give certain guarantees regarding transfer delay, bandwidth, loss ratio, etc. This enables using the same network for LAN/WAN data traffic and multimedia communication services.

Cells

Packets in ATM networks are called cells and have a constant size of 53 octets. At first sight, 53 is a rather unconventional choice for the size of a basic data unit. But this results from two contrary efforts: Enlarging the data unit reduces overhead due to headers. Minimizing the data unit reduces the delay of single octets. Thus, 53 octets seemed to give the best delay/payload ratio for transmission of both data and PCM-coded voice signals. The first five octets of a cell make up the cell header; the remaining 48 octets are referred to as the cell payload (Figure 2.1). The format of the cell header at the UNI is defined in [ATM94].

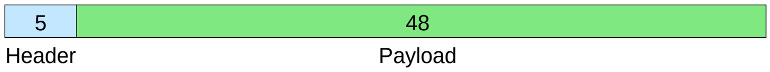

Figure 2.1: ATM Cell

Virtual Connections

As shown in Figure 2.2, the cell header has a well-defined structure. It contains the Virtual Path ID (VPI) and Virtual Channel ID (VCI) fields. A Virtual Channel (VC) is a unidirectional communication channel, potentially with associated traffic parameters. All cells of a VC have the same VCI value. Several VCs can be grouped in a Virtual Path (VP), marked by a common VPI value. The concept of Virtual Channels and Virtual Paths enables asynchronous multiplexing of several logical connections on a single physical link.

bit 7	6	5	4	3	2	1	0	octet
GFC				VPI (4...7)				1
VPI (0...3)				VCI (12...15)				2
VCI (4...11)								3
VCI (0...3)				PTI			CLP	4
HCS								5

GFC	Generic Flow Control	VPI	Virtual Path Identifier
VCI	Virtual Channel Identifier	PTI	Payload Type Indicator
CLP	Cell Loss Priority	HCS	Header Checksum

Figure 2.2: ATM Cell Header

The physical structure of an ATM network (see Figure 2.3 for an example network) is a set of ATM devices connected via point-to-point links (segments). Usually, ATM end systems (A, B, C, D) have a single ATM interface, whereas ATM switching devices (X, Y, Z) have several ports (X1, X2, Y1 ... Y3, Z1 ... Z3).

A way from one ATM device to another can be described by an ordered list of intermediate switches and the respective outgoing ports. Before cells can be sent from A to C a virtual connection must be established along the path. This can be done by manual configuration (permanent virtual circuit — PVC) or using signalling protocols to establish switched virtual circuits (SVC). Connection setup involves path discovery (A $\rightarrow$ X2 $\rightarrow$ Y3 $\rightarrow$ Z2 $\rightarrow$ C) and selection of exclusive VPI/VCI pairs. Since it would be a very bad idea to reserve the same exclusive VPI/VCI pair along the whole path, these

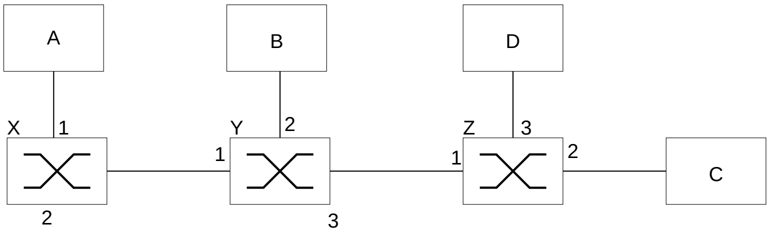

Figure 2.3: Example ATM Network

pairs are assigned for single segments only.

There is kind of a routing table in the intermediate switching entities. For each open connection it contains an entry with output port and new values for VPI and VCI. Thus the cell header is changed when passing an intermediate switch. Due to the constant packet size and the simple header structure this can almost be done in hardware.

ATM Adaption Layers

Independent of the type of data, the tiny ATM cells are not very handy for data transfer. Corresponding to the demands of different service types, a set of ATM Adaption Layers (AAL) was defined. An AAL specifies the mapping of high level data units and control mechanisms to the ATM layer (SAR - segmentation and reassembly). Initially, five AAL types were defined:

- real-time aware services with constant bit rate (AAL1)
- real-time aware services with variable bit rate (AAL2),
- non-real-time connection-oriented (AAL3) or connectionless (AAL4) services, merged into AAL3/4 later
- AAL5, a simplification of AAL3
- AAL0, in fact no AAL at all, direct cell access

AAL5

Since AAL5 seems to be the most often used AAL for mass data transfer, a short overview is presented here. The AALs are specified in ITU-standards I.363.1 ... I.363.5. AAL5 provides an unreliable transport service for protocol data units (PDUs) of 1-65535 octets. Unreliable means in-order delivery of undamaged PDUs. Damaged PDUs are discarded and no retransmission occurs. Prior to transmission, a padding area (0...47 octets) and a trailer (8 octets) are appended to the PDU data (Figure 2.4).

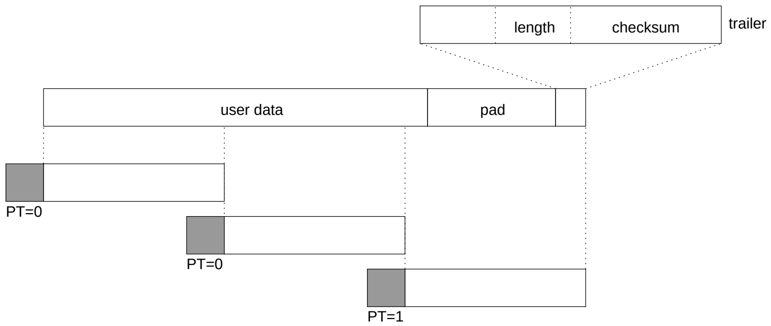

Figure 2.4: AAL5 PDU segmentation

The size of the padding area is determined to make the whole PDU a multiple of 48 octets. The resulting block is split into groups of 48 octets (segmentation) that are placed in the payload field of cells. The payload type field of these cells is set to 0; except for the last cell, where it is set to 1 indicating the end of the PDU. Integrity of a PDU is checked by means of CRC checksumming. During reception, the cell payloads are concatenated (reassembly) and the padding area and trailer are discarded to form the original PDU content.

Traffic types

Different types of service imply different traffic characteristics. During connection setup, the connection parameters are negotiated. The user specifies a set of parameters describing the connection characteristics (peak/minimum/sustainable cell rate (PCR, MCR, SCR), maximum burst size (MBS), maximum cell transfer delay (maxCTD), peak-to-peak cell delay variation (CDV), cell loss ratio (CLR), etc.). Depending on the ATM network's capacities and capabilities, the connection is accepted or rejected. If the connection is accepted the network guarantees to meet or exceed the confirmed Quality of Service (QoS) for the lifetime of the connection. Several service categories have been defined with respective parameter sets:

- UBR — Unspecified Bit Rate (PCR)
- CBR — Constant Bit Rate (PCR, CDV, maxCTD, CLR)
- rt-VBR — real-time Variable Bit Rate (SCR, MBS, PCR, peak-to-peak CDV, maxCTD, CLR)
- nrt-VBR — non real-time Variable Bit Rate (SCR, MBS, PCR, CLR)
- ABR — Available Bit Rate (PCR, MCR), feedback mechanism

For every service category, algorithms are specified for checking the conformance of cells to the negotiated parameters.

Network Resource Management

Unfortunately, resources like bandwidth and buffer space in networks are limited. Non conforming connections could cause congestion by over-utilizing assigned resources. In [ATM96] a set of traffic and congestion control functions is described. ATM networks can implement one or a combination of these functions in order to meet QoS objectives of compliant connections.

- Connection Admission Control
- Usage Parameter Control — monitor connections, detect violations and take appropriate actions like
  - Cell Tagging
  - Cell/Frame Discarding
  - Traffic Shaping
- Flow Control

Cell or frame discard limits the rate of a connection by dropping cells or whole frames. In contrast, traffic shaping is a work-preserving way to force a connection to meet the connection parameters. Provided that sufficient buffer space is available, data is buffered and forwarded at the negotiated rate.

Summary

The design of ATM enables building networks that can give real-time guarantees. Due to the fixed sized tiny cells, switching can be done almost in hardware. This enables very low latencies and high bandwidth. Generally, ATM networks seem to be the optimal infrastructure for transmission of multimedia and real-time sensitive data.

2.2 Linux

Linux is a freely-distributed UNIX-like operating system, originally created by Linus Torvalds. Developed under the GNU General Public License (GPL), the source code for Linux is available to everyone. With the freedom to create and adapt Linux for a wide variety of platforms, Linux has become quite popular for business and personal use worldwide. Lots of developers worldwide contribute to the Linux project by creating applications and extending the kernel (device drivers, protocol stacks, etc.).

From the system developers view Linux is an operating system with a monolithic kernel. In [Tan90] a monolithic kernel is defined to be a collection of procedures with each procedure being allowed to call any other procedure if needed. This results in a quite intricate structure making isolation of a single component difficult. Since there is no memory protection inside the Linux kernel itself, a failure in one procedure may influence others, and hence, the whole kernel might become instable due to a single failing component.

2.3 PCA-200E Linux Driver

The PCA-200E Linux driver was the first step towards the DROPS ATM protocol stack. It plugs into ATM-on-Linux[Alm99] , the ATM suite for Linux. ATM-on-Linux was started by Werner Almesberger in 1995. Since then, many people contributed to the project by writing or porting device drivers and management tools. Currently still available as a separate source distribution, ATM-on-Linux is going to be integrated into the mainstream Linux kernel soon. ATM-on-Linux supports the following protocols:

- "raw" unreliable ATM transport without AAL ("AAL0")
- "raw" unreliable ATM transport over AAL5
- ATMARP (RFC1577; client and server) for PVCs and SVCs
- LAN Emulation (client and server) for SVCs
- LANE Version 2 (client), Multi-Protocol Over ATM (MPOA, client)
- Arequipa (Application REQuested IP over ATM)
- ATM Name Service (ANS, client and server)

Since the existing drivers did not seem to fit easily into the ATM-on-Linux Device Driver Interface [Alm96b], the PCA-200E driver was written from scratch. It has a two layered structure; the lower layer controls the firmware executing on the PCA-200E board — the upper layer is the glue between the Device Driver Interface and the lower layer's functions. This results in a fairly portable hardware dependent lower layer and a system dependent upper layer.

The driver was developed on x86-based Linux systems. But, the well-chosen structure of the PCA-200E Linux driver allowed porting to Power PC Linux and SPARC Linux within a few days only.

Information on the current state of the driver can be found in [Dre99b].

2.4 L4

The L4 μkernel was developed by Jochen Liedtke at the National Research Center for Information Technology (GMD) and IBM Watson Research Center. The initial version of L4 runs on Intel's IA-32 — meanwhile there are implementations available for DEC-Alpha from the Dresden University of Technology and for MIPS from the University of New South Wales. Furthermore, there is FIASCO, an IA-32-based re-implementation of L4 in C++.

As the name "μkernel" implies, L4 offers only a minimal set of functionality:

- fast, message-based interprocess communication (IPC)
- page based memory management
- priority based scheduling with hard priorities
- tasks as security domains

External pagers allow implementation of almost any desired memory management outside the kernel. This minimalistic design offers highest flexibility at high speed.

The following definitions for L4 primitives are an excerpt of [Lie96]:

IPC: Interprocess communication in L4 is message based and takes place between exactly two threads. For a message to be transferred both parties must agree to the transfer. Message transfers happen always synchronously. There are mainly two types of messages — short messages and long messages. A short message consists of words where (the size of a word and) the maximum number of words is architecturally dependent. Short messages are transported entirely in the processor's register set. Hence, short messages provide the fastest communication path. Long messages use a message descriptor (message dope) located in memory. By use of long messages any number of words can be copied between the IPC partners, either located in the message dope itself or referenced in the message dope. When a message is marked having Flexpages, the words of the message are interpreted as Flexpage descriptors.

Address spaces: An address space is a mapping which associates each virtual page to a physical frame or marks it non-accessible. Address spaces can be manipulated by sending Flexpages in IPC messages. L4 supports recursive creation of address spaces outside the kernel. But, to prevent corruption of address spaces all changes must be controlled by the kernel.

Flexpages: Flexpages are regions of the virtual address space, consisting of all pages mapped in this region. Sending a Flexpage by means of IPC adds all the pages currently mapped in this Flexpage to the destination's address space.

Threads: A thread is an activity, being characterized by some kind of state information (registers, instruction and stack pointer, priority, ...) and an associated address space. L4's scheduling works on thread level.

Tasks: A task is the entirety of an address space and all threads (active or inactive) executing in this address space. Moreover a task is also a protection domain for IPC.

Interrupts: In L4 hardware interrupts are translated into an IPC message to a certain thread. A thread can register with a hardware interrupt to be notified when that interrupt occurs.

2.5 DROPS

The Dresden Real-Time Operating Systems Project[1] is a research project aiming at the support of applications with Quality of Service requirements.

Although much research has been done on networking support for continuous-media applications, very few projects tackle related operating system issues, such as scheduling and file system support for bounded response time. The DROPS project attempts to find design techniques for the construction of distributed real time operating systems whose every component guarantees a certain level of service to applications.

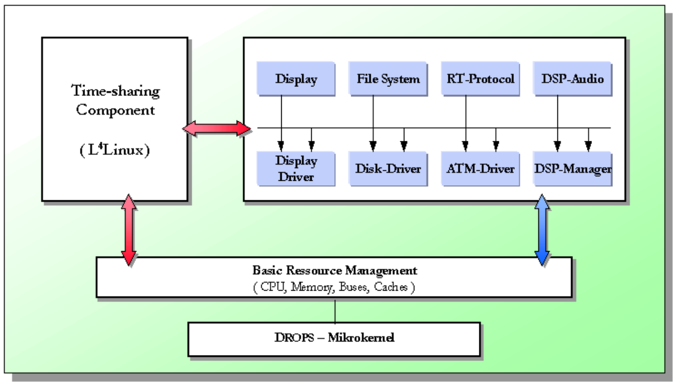

Figure 2.5: DROPS architecture

[1]The project is supported by DFG (Deutsche Forschungsgemeinschaft, SFB 358, Teilprojekt G2).

A key component is L4Linux, the Linux server on top of the L4 μ-kernel; it serves standard Linux applications. In addition, separate real time components – designed from scratch — provide deterministic service to real time applications (Figure 2.5). At the moment, an ATM protocol component, a real time file system, and a presentation component are available. The real-time components of DROPS are connected via the DROPS real-time streaming interface, an interface designed for the transport of jitter constrained streams.

More information regarding the DROPS project can be found in [Dre99a].

2.6 L4ATM

L4ATM is the ATM protocol component in DROPS. In its current state, it offers a narrow subset of the Linux ATM API [Alm96a] — only PVCs are implemented. L4ATM runs as a stand-alone L4 task, using the PCA-200E driver (Section 2.7) to access the hardware. Clients can use L4ATM through a client library hiding the complexity of the IPC protocol. The library provides the well-known BSD-style socket interface as well as functions for a "zero-copy" data path between the client and the ATM protocol server (`get_ncp_page()`, `ncp_read()`, `ncp_write()`). A stub driver exists for L4/Linux presenting the L4ATM protocol component as an ATM network device.

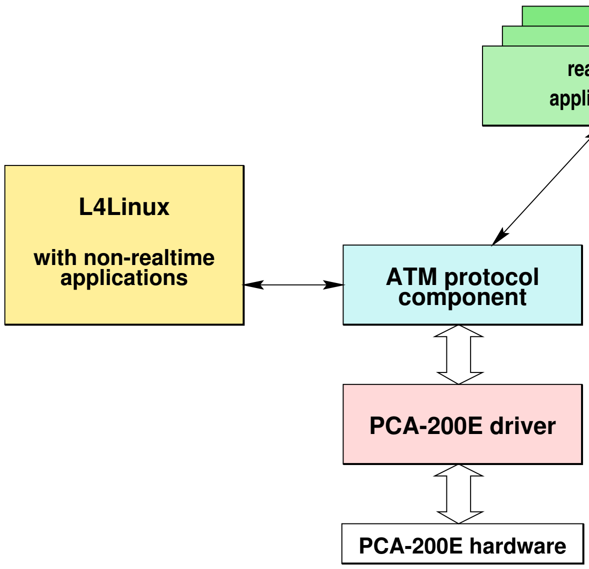

Figure 2.6: L4ATM in DROPS

Two shared memory areas — one for transmit, the other for receive — are established between the client and the L4ATM protocol server. The physical addresses of the mapped pages in the shared memory areas need to be known and fixed over the lifetime of the mapping. In other words this is pinned memory. The size of the shared memory areas is determined by L4ATM from the QoS parameter set specified with a `setsockopt()` call after socket creation and prior to connection establishment.

At connection setup time, a dedicated worker thread per connection is created in L4ATM. This thread handles transmit requests from the client and pushes received data to the client. On transmit, this thread may block to enforce the negotiated transmit rate (traffic shaping). If received data is available, it is pushed to the client at the negotiated receive rate.

A “pseudo interrupt thread” waits for messages from the driver’s interrupt thread and copies the received PDU in the buffer of the respective connection. If no free buffer space is available, the received PDU is dropped. Since buffer sizes are determined from QoS parameters, this indicates a non conforming traffic source. At that point the PDU was already transfered into main memory and processed by the driver and by the ATM protocol. With the current implementation of L4ATM and the PCA-200E device driver this path is very CPU intensive, which is even more annoying in an overload situation.

To point this out again, L4ATM manages exclusive receive buffers and a dedicated worker thread per VC. The receive buffer size is determined from the QoS parameter set of the connection.

L4ATM is described in much more detail in [Bor99].

2.7 PCA-200E Driver for DROPS

The PCA-200E driver for DROPS (PCA-200E/L4) is a port of the PCA-200E Linux driver. The lower layer of the Linux driver — the portable hardware dependent part — was moved into a separate L4 task. To reuse the code of the PCA-200E Linux driver almost unmodified, an emulation of the Linux kernel functions used by the driver was required. This was the case for Linux’ memory management (`kmalloc()`, `kfree()`, `get_dma_pages()`), time management (`jiffies` counter) and interrupt management (`request_irq()`, interrupt handling semantics). Additionally, a server stub implements the IPC interface of the driver (Figure 2.7). A client library offers the low layer’s functions, effectively hiding the IPC interface in between.

There are two threads in the PCA-200E driver for DROPS: a service thread that handles transmit and control requests and an interrupt thread that pushes received data to the client. To avoid data loss, the client should be ready to accept received data anytime. Since there is only one client in DROPS, the driver does not implement demultiplexing. Although the driver is capable of enqueueing several requests, the ATM protocol component described in the next section requires a synchronous transmit path. As a proof

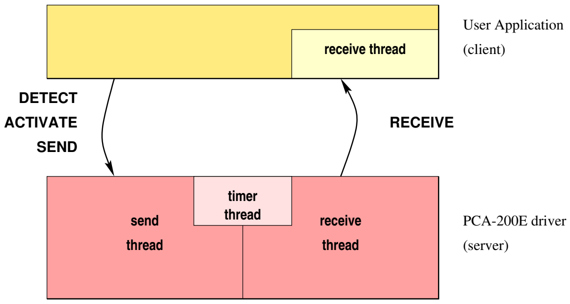

Figure 2.7: PCA-200E DROPS driver overview

of concept, the former upper layer of the Linux driver remained as a stub in the L4/Linux kernel together with the client library (Figure 2.8).

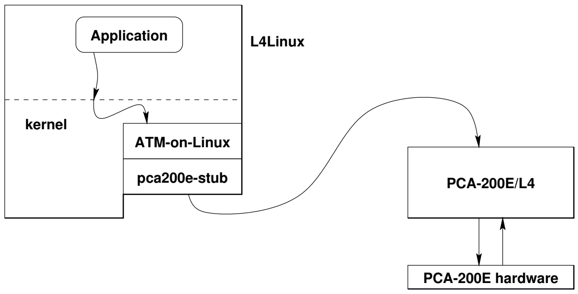

Figure 2.8: PCA-200E/L4 and L4/Linux

A more extensive description of the PCA-200E driver for DROPS is given in [Dan98].

2.8 PCA-200E Hardware

The PCA-200E is the PCI based member of FORE's *FORERunner 200E* 155Mbps ATM network adapter series. Figure 2.9 shows a schematic diagram of the PCA-200E.

All members of the 200E series share a common part and a host-bus specific interface. The common part consists of Intel's i960-CA processor, 256 KB RAM local to the i960, an ESP-ASIC (Enhanced SAR Processor) with 128KB RAM and PMC Sierra's SUNI-155/Lite User Network Interface. Either an UTP interface or a fiber-optical interface is attached to the UNI. Host side interfaces are available for SBus, EISA Bus, GIO Bus, PCI Bus, VME bus and Micro Channel Bus.

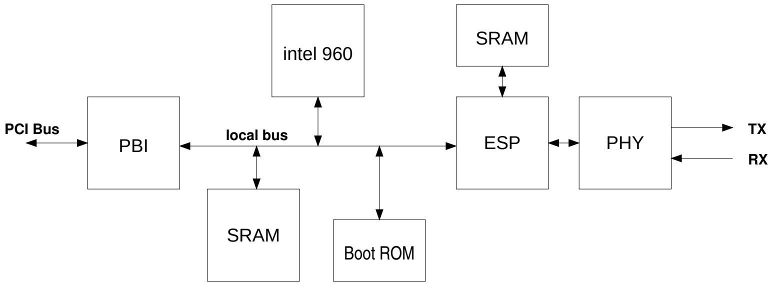

Figure 2.9: PCA-200E Structure

2.8.1 CPU

Intel's i960-CA processor is a general-purpose 32 bit RISC processor, mainly targeting at the embedded systems market. Its architecture features:

- load-store architecture
- sixteen 32 bit global registers, sixteen 32 bit local registers
- 4GB flat address space, no virtual memory
- fast call-and-return mechanism, saves/restores local registers
- local register cache, speeds up calls and returns
- bus control unit, pipelined burst 100MB/s (at 25 MHz)
- priority interrupt unit, 32 priorities, 248 interrupt vectors
- simple supervisor protection model
- 1KB fast internal RAM
- sustained two instructions per cycle

A complete description is given in [Int94]. On the 200E series adapters, the CPU is responsible for the segmentation and reassembly of PDUs. Furthermore, it has to perform all data movement.

2.8.2 Host Interface

The PCI Bus Interface (PBI) hides the complexity of the PCI-Bus Protocol from the i960 (CPU). It implements the PCI Bus Protocol as specified in [PCI95]. Access to the world behind the PCI bus is offered to the i960 via two FIFOs, each 128 words deep. A write access is caused by writing destination address and word count to the OUT FIFO, followed by the actual data words. To generate a read, source address and word count are written to the IN FIFO. After that, the data words can be pulled from the IN FIFO. Special modes for fixed size transfers (1, 4, 8, 11, 12, 22, 24 words) exist. Handshaking is done by means of FIFO word counters. To avoid FIFO overflows, the PBI can hold the i960 temporarily.

From the host's point of view, the PBI exports a 2 MB memory area. In this 2 MB area the first 1 KB is unused; in the next 255 KB the i960's local ram can be accessed. The Host Control Register (HCR) resides at an offset of 1 MB.

The PCA-200E can act as a PCI Master (activity caused by the i960) and Target (host accesses shared memory) and can generate interrupts which have to be acknowledged by the host in the HCR.

A detailed description of the PBI is given in [Ben95b].

2.8.3 User Network Interface

The *FORERunner 200E* boards are designed for SONET/SDH links. According to [PMC96], PMC Sierra's SUNI-155/Lite chip implements the ATM Forum User Network Interface Specification [ATM94] and the ATM physical layer. It performs the complete SONET/SDH framing process, HCS generation/correction, and insertion and removal of idle/unassigned cells. In fact, it translates a stream of 52-octet blocks (4 octets cell header without HCS, 48 octets payload) to the line protocol and vice versa. Thus, it hides the complexity of the line protocol from the other components on the board.

2.8.4 Enhanced SAR Processor

The Enhanced SAR Processor (ESP) is the decoupling element between the CPU and the User Network Interface. The main objective of the ESP is to provide an easy-to-use interface to the UNI for the CPU. Inside the ESP, CRC generators for AAL3/4 and AAL5 checksumming, four transmit FIFOs and four receive FIFOs, transmit rate generators, event counters and a timer are implemented. The CPU accesses the ESP through memory mapped registers in several regions.

FIFOs are implemented in a 128KB SRAM attached to the ESP. The ESP logic maintains head and tail pointers for each FIFO, that are updated whenever cells are inserted into or pulled from a FIFO. Data can be moved between the PBI and the ESP's FIFOs by "fly-by"-transfers. When the CPU performs a read from a special region of the ESP, a read from the ESP receive FIFO occurs while a write is performed simultaneously to the PBI's OUT FIFO. Similarly, a read from another region of the ESP performs a read of the IN FIFO while writing to the ESP transmit FIFO. These "fly-by" regions support burst mode which should be used to maximize the transfer rate. Avoiding the CPU to touch all data twice, this enables 100MB/s on-board transfer rates (at 25MHz) — that is approximately three times the full duplex OC3 speed and a nearly saturated PCI bus (32 bit, 33MHz).

CRC checksums are calculated when data is transfered to or from the ESP's FIFOs. As an example, to generate the trailing checksum for AAL5, the AAL5 CRC register is set to an initial value. Then, after all data (user data, pad bytes, partial AAL5 trailer without CRC) is written to the transmit FIFO, the AAL5 CRC register holds the correct CRC checksum which is simply written to the FIFO, too. The same works during receive: the CRC register holds the calculated checksum which has to be compared with the received value.

While transmitting, cell headers must be written on cell boundaries to a cell header register which is tightly coupled with the transmit FIFO. Header coalescing is a neat feature to reduce the overhead caused by cell headers. It allows to set a cell header once and keep it for all following cells until a new header is written. On the receive path, the cell header is pulled from the cell header register together with the number of cells having the same header. Consequently, cell payloads can be transfered between the PBI and FIFOs in blocks of more than a cell payload, further reducing the CPU cycles per cell.

The transmit FIFOs are connected to a cell scheduler/multiplexer. FIFO can either have high or low priority. A per FIFO token generator generates tokens (permission to emit a cell) at a certain rate which can be controlled by the CPU. Whenever a FIFO has a token and a cell available, the cell scheduler can transfer the cell from the FIFO to the UNI. This can be used to force outgoing connections to a certain cell rate (traffic shaping). The scheduler serves the high priority FIFOs first in a round robin manner; if no high priority FIFO has token and data available, the low priority FIFOs are served round robin.

Which receive FIFO to place an incoming cell in is determined by to the two least significant bits in the VCI field of the cell header. This fixed assignment can reduce the number of connections per FIFO, effectively increasing the probability of back-to-back cells with the same header (header coalescing). Obviously, this relies on VCI values being allocated in densely ascending/descending manner. For most ATM switches this assumption is valid; the VCI value is continuously increased on each new connection, wrapping around at the end of the VCI space.

The ESP is described in detail in [Ben95a].

2.9 PCA-200E Firmware

After power-on, the i960 CPU on the PCA-200E runs mon960, a standard debug/monitoring tool from Intel. The mon960 code is located in a small boot ROM on the board. It can be used to download and start the firmware. The PCA-200E firmware is part of the drivers for the operating systems supported by FORE. It is to be loaded onto the PCA-200E during system start.

After initialization the firmware implements the AALI [FOR97]. The AALI defines a command queue, a transmit queue, a receive queue and up to four buffer queues. Queues are simply an array of queue entries. Each queue forms a logical ring list by wrapping around after the last array element to the first. Queue lengths are configurable at initialization time. The queues in detail are:

Command queue: The command queue is used to send commands to the firmware: `activate_vci` (open), `deactivate_vci` (close), `request_stats`, `zero_stats`, etc.

Transmit queue: To send data, a Transmit PDU Descriptor (TPD, containing location/length of data in host memory, ATM cell header, ...) located in host memory is filled and then its address is written to the transmit queue.

Receive queue: The receive queue holds references to Receive PDU Descriptors (RPD). These RPDs are located in host memory and are written to by the firmware on reception of data from the network. An RPD holds references to buffers in host memory containing the received data.

Buffer queues: Via the buffer queues new receive buffer descriptors are supplied to the firmware. There are up to two receive buffer pools, each with up to two buffer sizes, giving a maximum of four buffer queues.

When opening a new VC with the `activate_vci` command, an associated receive buffer pool can be selected. Unfortunately, there is no way to directly specify receive buffers per VC. This prevents implementation of zero-copy receive data paths. The receive buffer pools and the receive queue are shared by all connections, allowing monopolization of these resources by one connection while other connections may lose data.

2.10 The U-Net Project

The U-Net architecture developed at Cornell University, provides low-latency and high-bandwidth communication over commodity networks for workstations and PCs. It achieves this by virtualizing the network interface such that every application can send and receive messages without operating system intervention. With U-Net, the operating system is no longer involved with the sending and receiving of messages. This allows communication protocols to be implemented at user-level where they can be integrated tightly with the application. In particular, the large buffering and copying costs found in typical in-kernel networking stacks can be avoided and feed-back to the application about flow-control and packet loss is facilitated.

The key aspects of U-Net are:

- U-Net defines a virtual network interface for commodity networking hardware and operating systems.
- The U-Net virtual network interface preserves the traditional protection boundaries between processes. Multiple applications can use U-Net at the same time without interfering.
- U-Net uses commodity operating systems (Unix and Windows/NT) and commodity networks (Fast Ethernet and ATM).

U-Net supports the Myrinet PCI and SBus interfaces, DECChip 21140 Fast Ethernet PCI interface, and the FORE Systems PCA-200/SBA-200 ATM interfaces. Supported operating systems include Linux, SunOS 4.x, Solaris 2.x, and BSDI.

U-Net/MM is an extension to the U-Net user-level network architecture, allowing messages to be transferred directly to and from any part of an application's address space. This is achieved by integrating a translation look-aside buffer into the network interface and coordinating its operation with the operating system's virtual memory subsystem. This mechanism allows network buffer pages to be pinned and unpinned dynamically.

The source of U-Net related information is [Cor96].

2.11 The RIO Subsystem

The RIO subsystem [KSL99], a project at Washington University, enhances the Solaris kernel to enforce the QoS features of the The ACE ORB (TAO) endsystem and provide end-to-end QoS. That is achieved by using early demultiplexing and schedule-driven protocol processing.

The Solaris default network I/O subsystem processes all packets sequentially at the same priority, regardless of the destination user thread. That can lead to priority inversion easily, preventing high-priority connections to meet their QoS requirements. To overcome this, RIO supports priority-based queueing — instead of enforcing strict FIFO order, packets destined for high-priority applications are delivered ahead of low-priority packets. Connections are assigned a priority which is determined from the connection characteristics given with TAO's QoS specification. Priorities map to RIO queues, which are served by an associated in-kernel thread with respective scheduling priority. Incoming packets are inspected by a packet classifier in the network driver, which decides whether processing takes place in interrupt context (for low-delay connections) or the packet is enqueued in the appropriate queue.

To summarize, the RIO subsystem tries to preserve end-to-end priorities by separating resources and applying priority-based protocol processing.

Chapter 3

Design

L4ATM together with the PCA-200E driver using FORE's firmware offer ATM connectivity to DROPS applications. Even under heavy load, like the AIM multi-user benchmark running under L4Linux, this protocol stack can guarantee promised connection characteristics for real-time connections. But, that setup does not behave well with unexpected heavy traffic arriving from the ATM network. In that case CPU utilization increases although packets are probably dropped by L4ATM due to buffer shortage. Even worse, real-time connections may experience data losses because of receive queue congestion or receive buffer outage in the PCA-200E driver and firmware.

To overcome these problems, this work aims at providing a smarter firmware for the PCA-200E. Due to unavailability of the firmware source code, the firmware has to be designed and implemented from scratch. On one hand, this means re-implementing parts that were acceptable before; but on the other hand, it gives the opportunity to build a firmware that suits the DROPS design. Furthermore, it enables one to apply changes in case of future insights. However, it seems to be a chance to overcome the limitations of the current ATM protocol stack.

This chapter identifies the design goals. From that the process model of the firmware is derived and its implications are described. After inspecting the data paths, the individual processes are described while certain aspects are investigated in more detail.

3.1 Design Goals

Among others, the aspects listed below have been identified to deserve increased interest when building real-time systems. The firmware design should try to minimize these and make them as predictable as possible.

- memory bus utilization

- I/O bus utilization
- interrupt frequency
- host CPU utilization

Zero-copy data paths are a common approach to keep the host's CPU utilization low for protocol processing, even at very high data rates. This can be achieved by the use of per-VC transmit and receive buffers. Furthermore, per-VC receive queues and receive buffers minimize influences of concurrent connections and isolate misbehaving connections (early demultiplexing). I/O bus utilization can be minimized by placing data structures local to the activity that most often accesses them. Interrupt frequency can be reduced and bound by generating a single interrupt for a group of events. That way, the host CPU utilization caused by network activities can be limited even under worst case conditions.

In order to provide the capabilities offered by the ATM network to the applications, further design goals must be met:

- minimize mutual influences of concurrent connections
- minimize host's CPU utilization for malicious connections
- provide support for shaping outgoing traffic

Practice shows that AAL5 is the most often used AAL. The firmware should implement segmentation and reassembly for AAL5. Nevertheless, the design must not prohibit later implementation of other AALs or even higher level protocols in the firmware.

3.2 Process Model

At a first glance, there are three main tasks the firmware has to work on in parallel:

- Transmit data to the network
- Receive data from the network
- Handle control requests from the host

This calls for a multitasking system. An interrupt-based approach is unlikely to perform well here. The FIFO mechanisms would require extensive locking to prevent concurrent accesses. Opposed to that, cooperative multitasking gives a task full access to all the FIFOs at runtime and avoids any locking.

Tasks are forced to save their state between invocations. The system can be reduced to a list of functions that are executed one after another repeatedly, as the following piece of pseudo code illustrates:

```
while (true)
{
  transmit();
  receive();
  handle_commands();
};
```

Execution times of these functions should be minimized and bound to reduce overall latency and to avoid data loss due to FIFO overflows. Whenever a function would block on a certain operation, it should save its current state and return to its caller. The current state of a process is stored in a set of descriptors — per-VC receive descriptors hold the state of the receive process, per-PDU transmit descriptors are used for the transmit process state.

3.3 Transmit

The transmit process covers transmit queue handling with transmit FIFO selection, refill of the transmit FIFOs and event handling for completed transmissions. In this section, an overview of the transmit data path is given first, followed by notes on the FIFO selection scheme. A description of the transmit process completes this section.

3.3.1 Transmit Data Path

In DROPS, an application wishing to communicate over ATM uses L4ATM. The upper interface of L4ATM offers two slightly different methods to transmit data: the copying `write()` function call and the "no-copy" function `ncp_write()`. The function `get_ncp_page()` requests a buffer in the memory area shared between L4ATM and the application. The application copies its data into the buffer and calls `ncp_write()` to hand over the buffer to L4ATM for transmission. `write()` is called with the address of a buffer private to the application. The client library then passes the buffer contents to L4ATM by a copying IPC operation. The result of both methods is similar: transmit data (PDU) is accessible in L4ATM's address space and the respective physical address is known. The worker thread in L4ATM performs protocol processing — including invocation of traffic management algorithms — and conveys the PDU to the device driver.

The driver must inform the firmware of the transmission request and pass all information required to transmit the PDU to the firmware. The firmware

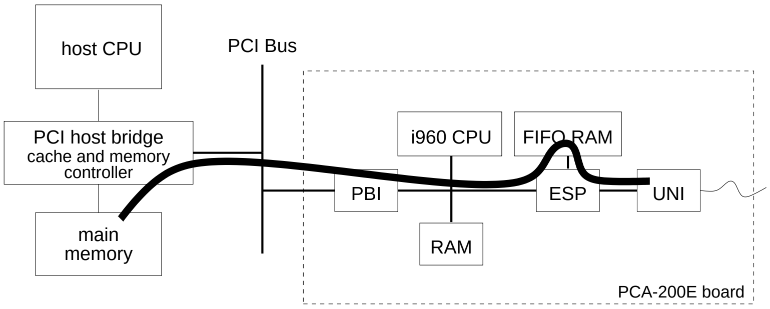

Figure 3.1: Data Path

selects a transmit FIFO according to the PDU description. While eventually performing the associated AAL processing (header/trailer, checksum generation) the PDU data is transfered from the host's memory via the PBI into the transmit FIFO as FIFO space is or becomes available. Due to the limited size of the PBI's IN FIFO and the ESP's transmit FIFO this can take a quite long time, depending on the PDU size — especially at very low transmit rates. A plot of the physical data path is given in Figure 3.1.

3.3.2 Transmit FIFO Selection

DROPS favours the idea of running real-time applications and non real-time applications on the same machine. The same is valid for communication: there are two different qualities of network connections — real-time connections and best-effort connections. Naturally, best-effort connections have a lower priority than real-time connections. As the name "best-effort" implies, these connections can utilize only resources not used by real-time connections. With the hardware given in the ESP, this scheme is applied easily: one FIFO is set to low priority, the remaining three FIFOs are set to high priority. Real-time connections use one of the high-priority FIFOs to shape their traffic. Remembering the FIFO scheduling scheme described in Section 2.8.4, the fourth i.e. the low-priority FIFO can only emit a cell when no high-priority FIFO has data available. Hence, all best-effort connections using the low-priority FIFO share the remaining bandwidth.

The rate of a FIFO's token generator can be changed anytime. This implies somehow that the rate must not be changed when the FIFO holds transmit data, since this would change the rate of the currently processed PDU. As long as the number of concurrent real-time connections does not exceed the number of high-priority FIFOs, these FIFOs can be used exclusively by a certain connection. Other scenarios are discussed in Section 4.6.

3.3.3 Transmit Process

At first sight, the firmware needs at least the following information to transmit a PDU: the PDU location (physical address), PDU length, VPI and VCI number and AAL type. From L4ATM's point of view, there are real-time connections and best-effort connections. Real-time connections are established with a certain transmit rate; best-effort connections are intended to transmit as fast as possible. Hence, transmit requests for real-time connections can be mapped to shaped transmissions while transmit requests for best-effort connections cause non-shaped transmissions. Having this in mind, it seems sufficient to add a rate descriptor to the list above in order to have a distinction between real-time connections and best-effort connections even in the firmware. An invalid rate descriptor then indicates a non-shaped (best-effort) request. The elements of a transmit PDU descriptor are shown in Table 3.1.

field name	field size in bits
physical address	32
PDU length	16
VPI	≤ 8
VCI	≤ 16
AAL	3
transmit rate	16
handle	32

Table 3.1: TPD Elements

Ttransmit PDU descriptors (TPD) reside in host memory. The firmware maintains a transmit queue in the board's local RAM; this queue is a logical ring list of addresses of TPDs. The driver sets up a TPD and writes this TPD's physical address to the current transmit queue entry. The firmware processes the request and informs the driver either by an event or by simply marking the descriptor as completed. This distinction enables the driver to reduce the interrupt frequency.

The transmit queue resides in the board's local RAM in order to reduce PCI-Bus activity. That way, a single PCI write access from the host is sufficient to issue a transmit request. The firmware polls the current entry of the transmit queue periodically, which does not involve any PCI Bus transfers. If a new request (the address of a TPD) is found, the TPD is fetched into the local RAM, the transmit queue entry is invalidated and the reference to the current entry is advanced. The fields of the TPD are validated and an internal TPD is set up. An internal TPD contains the fields listed in Table 3.2.

Internal TPDs are enqueued per transmit FIFO. Which of the four transmit

field name	description
address	address of remaining data
pdu_length	PDU length
data_words	remaining words for data
pad_words	remaining pad words
partial_crc	temporary CRC checksum
tx_function	AAL-specific transmit function
complete_function	function to call on completion
tx_rate	
handle	

Table 3.2: Elements of an internal TPD

FIFOs is primarily used depends on the traffic type (see Section 3.3.2). The transmit process handles the first TPD in each FIFOs queue by calling the associated transmit function. The transmit function implements the AAL processing and data transfer to the FIFO. If the FIFO is filled up before all data from the TPD is used the TPD is updated and the next FIFO is served. This way a nonblocking operation can be achieved. Before the FIFO has been drained the firmware can work on other tasks and is back to fill the FIFO again. If a TPD has been completed — i.e. the PDU has been transmitted completely — the associated completion function is called and if available the next TPD is served. The firmware design reqires that the completion function runs for a fairly short time only.

3.4 Receive

Reception of data from the network involves several stages: identify incoming cells, either drop cells or apply AAL processing, transfer data into host buffers and notify the driver. This section gives an overview of the receive data path followed by notes regarding identification of incoming cells. An explanation of the necessary changes to L4ATM's receive buffer scheme and the description of the receive process complete this section.

3.4.1 Receive Data Path

Incoming cells are put in one of the four receive FIFOs, no matter if there is an open connection for that VPI/VCI pair. If in any of the four FIFOs are more than a configurable number of cells, a flag is set in a control register (An interrupt could be requested too). For the sake of low latency the cells should be pulled from the ESP's receive FIFO as soon as possible. Cells for which a connection was registered are processed by the respective receive function for reassembly. All other cells are dropped. The receive function implements AAL processing and reassembly. Obviously, this is the best point to apply early demultiplexing. Receive buffers for a VC are installed at connection setup. Buffer space is provided by L4ATM; the buffer size is determined from the QoS parameter set. The receive function transfers the reassembled data directly from the ESP's receive FIFO into the per VC buffers in host memory. This way a zero-copy receive data path to L4ATM can be implemented.

L4ATM's client library exports two functions an application can use for receiving data. The `read()` function copies the received data into a buffer specified by the application and returns the length. In contrast to that the function `ncp_read()` provides a pointer to the buffer and returns the length. While the implementation of `read()` copies data from L4ATM's address space (the receive buffer area) to the applications address space by means of IPC, `read_ncp()` only copies a reference to the data located in the shared memory area and thus implements a zero-copy receive data path to the application.

3.4.2 Identification of Incoming Cells

A major point of interest is the identification of incoming cells. First of all, a decision has to be made on whether to discard the cell(s) or not. Only cells of open inbound connections are important. All other cells should be removed from the ESP's receive FIFO as soon as possible.

The receive hardware provides four different receive FIFOs. When the use of all four FIFOs is not disabled, the receive hardware selects the appropriate FIFO (0..3) by inspecting the two least significant bits of the VCI field in the cell header. Depending on the way the FIFOs are served, cells with different VCI values could outstrip each other. This is legitimate, as long as cells of the same VC are kept in order. The same could also happen in a switch featuring per VC queuing.

The firmware must maintain information on all open VCs. When the cell header is pulled from the receive cell header FIFO, a connection descriptor — if a connection exists for this VPI/VCI pair — must be looked up. This operation should be fair in that it is equally expensive for all open connections; furthermore, the operation should be very cheap when no associated connection can be found. Implementation details are discussed in Section 4.4.

The result of the lookup operation is a pointer to an internal receive descriptor as described in Section 3.4.4. If the incoming cells do not belong to an open connection, the pointer is invalid and the cells are discarded.

3.4.3 Receive Buffers

DROPS' ATM protocol component — L4ATM — manages a dedicated receive buffer area per VC. This area is physically contiguous and the physical address is known. The size of the buffer area is determined by L4ATM on connection setup from the QoS parameter set. As for now, L4ATM's pseudo-interrupt thread selects the appropriate receive buffer and copies the received PDUs into the buffer at page boundaries. If the PDU would not fit into the free space at the end of the buffer it is placed at the start of the buffer (In [Bor99], this is called early wrap-around). An entry describing the PDU (location in buffer, size) is added to the list of pending PDUs for this connection.

Obviously, this has to change when demultiplexing is implemented in the firmware. Therefore, the buffer management is moved to the firmware as well. That means the firmware manages the list of received PDUs and exports it to L4ATM. The firmware reassembles received PDUs directly into the per-VC receive buffer. Due to hardware limitations, a PDU always starts at an address that is a multiple of four. In contrast to early wrap-around implemented by L4ATM, in the case of insufficient space at the end of the buffer the PDU continues at the start of the buffer (see Figure 3.2).

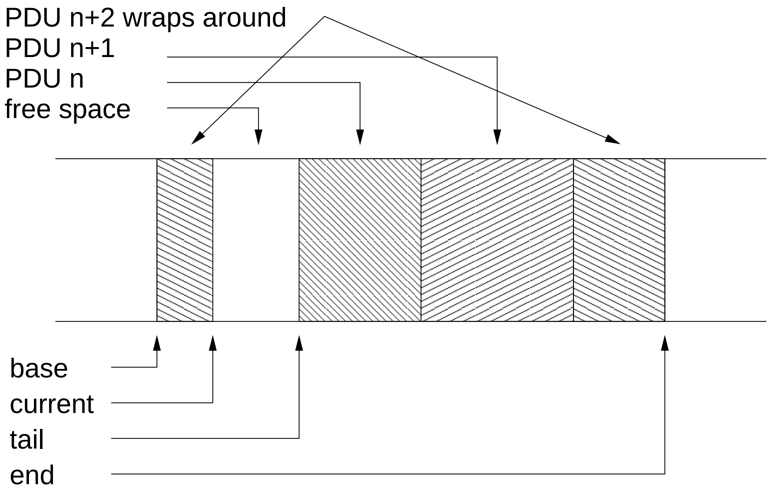

Figure 3.2: per-VC Wrap-Around Receive Buffer

Unfortunately, using this buffer model a PDU can happen to be split into two segments. Neither the API of L4ATM nor the ATM-on-Linux API support scattered buffers. To overcome this, the start of the buffer could be mapped behind the end of the buffer — assuming the buffer is aligned to a page boundary and a multiple of pages in size. That way a split PDU can be accessed as a contiguous block of virtual memory.

An internal buffer descriptor (Table 3.3) holds the required information to manage a receive buffer. It contains the general parameters of the buffer necessary for implementing wrap-around. The current pointer holds the address where received data is to be written to. A fallback pointer is used whenever a receive error occurs.

field name	description
current	where to write next data
fallback	restart here if PDU was damaged
base	start of buffer
end	end of buffer
tail	tail pointer updated by the driver

Table 3.3: Internal Receive Buffer Descriptor

A receive queue is maintained by the firmware for each connection. The firmware writes information about the received PDUs to the queue. The dedicated worker thread in L4ATM pulls entries from the receive queue, optionally applies traffic management algorithms, pushes the PDU to the application and adjusts the tail pointer for the buffer and the receive queue afterwards.

3.4.4 Receive Process

Internal receive descriptors hold a connection's receive process state. The fields of an internal receive descriptor are given in Table 3.4.

field name	description
rx_function	receive function, depends on the AAL
disable	if active, drop all data — reset at end of PDU
buffer	reference to receive buffer descriptor
complete_function	function to call after end of PDU
partial_crc	temporary CRC checksum
rxq_base	start of receive queue
rxq_size	size of receive queue
rxq_current	current receive queue entry
rxq_tail	tail pointer updated by the driver
handle	

Table 3.4: Internal Receive Descriptor Elements

Having looked up the receive descriptor for the cell header, the receive function is called. The receive function implements AAL processing and data transfer from the ESP's receive FIFO to host memory. Due to header coalescing, payload of more than one cell can be transfered in one operation, but a huge PDU is likely to be broken into a few blocks. The CRC checksum is calculated as data is pulled from the receive FIFO. After the transfer of a block the partial CRC value is stored in the RPD and restored before the next block of this PDU is transfered. The field `current` in the associated receive buffer descriptor is adjusted as data is written into the buffer. If the resulting value is at the end of the buffer, `current` is set to the buffer's base address. When the free buffer space between `current` and `tail` is not sufficient, buffer shortage is signaled for that connection. The current PDU is damaged due to cell loss; reassembly for that VC is suspended (transfer of data belonging to this PDU is disabled). With the end of the PDU, `current` is set to `fallback` and reassembly for that VC is resumed. In the preferred scenario the PDU is reassembled completely, the completion function is called and the internal receive descriptor is initialized for a new PDU (reset partial CRC value, set fallback pointer to current position).

The `tail` pointer resides in the board's local RAM. That fact is of increased importance when data arrives on a VC at a higher rate than expected. Probably, the receive buffer of that connection gets filled up, preventing further reception. The firmware polls the `tail` pointer on every arriving cell (or cell block) of that connection. These are local accesses that cause no PCI-Bus action. A single access from the driver to the board's local RAM is enough to get the `tail` pointer update. To minimize PCI-Bus utilization further, the host driver should cache the tail pointer and write back its value at the end of its receive processing for the specific connection.

The chosen process model requires timely termination of the receive process, even if there are still cells in the receive FIFO. Otherwise, transmit FIFOs could eventually drain off violating guaranteed transmit rates.

3.5 Events

Events are the means of communication from the firmware to the driver executing on the host CPU. A host resident event queue is filled by the firmware with event entries (Table 3.5).

field name	description
handle	specified by the driver, passed uninterpreted
type of event	indicates type of event

Table 3.5: Event Queue Entry Elements

While the event type enables easy classification of the event in the driver, the handle is used to identify the source of the event. These handles are specified when opening a connection, transmitting data or requesting control processing. Events can be generated for the following reasons (event types):

- transmit request completed
- control request completed
- PDU received
- receive queue full
- receive queue over threshold

Events can be **silent** or **interruptive**. Whenever an interruptive event is written to the event queue, an interrupt is generated on the PCI bus. Furthermore, an interrupt is generated when a receive queue gets filled up (error condition due to a non conforming connection) or the number of silent events has reached a threshold value. That way, connections with relaxed timing requirements (as best-effort connections are) can receive or transmit data without causing an interrupt for each single PDU. Hence, silent events are the means of reducing the interrupt frequency.

3.6 Control Requests

Besides the receive and transmit processes described in the previous sections, a third mechanism is required to control the firmware. This covers initial configuration, opening and closing connections, and status inquiries. The firmware maintains a command queue in the board's local RAM. The command queue is a list of physical addresses of command descriptors. The firmware polls the current entry of the command queue — since the queue is in the local RAM, no PCI transfers are required for that. An entry can be written by the driver in a single PCI transfer. In the following paragraphs the different request types are described, command descriptor structures are presented in the Appendix section.

Configuration: Global data structures are initialized with the configuration command. The length of both the command queue and the transmit queue, location and length of the event queue as well as the number of significant bits for VPI and VCI (i.e. the maximum number of connections) are configurable. The result of the configuration command are the addresses of the command queue, the transmit queue and the tail pointer of the event queue in the board's local memory. Obviously, the configuration command cannot be written to the command queue. Instead it is to be written to a well-known address in the board's local memory.

Open a connection: Opening a connection means enabling the receive process for the specified VPI/VCI pair. For that purpose, a receive buffer and a receive queue must be specified (base addresses and sizes). On success, the result consists of two addresses — one for the buffer tail pointer, the other for the receive queue tail pointer. Both addresses are offsets in the board's local memory. These locations are to be updated by the driver whenever it frees receive buffer space or receive queue entries. The driver should try to minimize the number of accesses. Additionally, a handle can be specified that is returned in the event queue as a reference.

Close a connection: Whenever a connection is closed, further reception of data on the specified VPI/VCI pair is disabled. The driver must not write to the tail pointer locations any longer. Furthermore, it should check the receive queue of the connection after completion of this command.

Status/Statistics: The current design does not require any status inquiries to be implemented. However, there may be need for in the future.

Parameters common to all control commands are a handle and an address where the result shall be written to. The size of the result buffer is implicitly given by the requested operation.

Chapter 4

Implementation and Performance Evaluation

In this chapter, some aspects of the implementation and their impact on performance are discussed. A few important data structures are described as their structure settled during the implementation process.

4.1 Modularization

For the implementation a modular design approach has been chosen. This means separating functional units of the firmware into the following modules:

- PBI initialization, ESP initialization, entry code, main loop and control request handler
- transmit data movement, receive data movement
- AAL5 segmentation and reassembly
- internal transmit descriptors (allocator, queue handling)
- internal receive descriptors (allocator, activate/deactivate, lookup)
- receive buffers, receive queues
- transmit process (transmit queue, FIFO selection), receive process
- events (interrupt generation)

That way e.g., applying a new receive buffer scheme involves only changing the receive buffer module.

4.2 Memory

The PCA-200E board has 256 KB of fast SRAM installed for storing the firmware code and data. The first KB is shadowed by the i960's internal RAM. During the implementation phase, the mon960 debugger is of much help (breakpoints, single stepping, etc.) — which takes about 20KB for its private data structures. So, there are about 234 KB available for the firmware. This space has to be shared for code and data. 32KB would be a very pessimistic estimate of the code size, leaving 200KB for data related to connection management. Targeting at a minimum of 1024 connections, this would result in nearly 200 bytes per-connection data (including receive descriptors, buffer descriptors, transmit descriptors, etc.). Taking into consideration that no data is to be buffered, that seems plenty of space.

4.3 Header Coalescing

Using the header coalescing feature when transmitting an AAL5 PDU, only two updates to the cell header register are required opposed to writing a cell header for each cell. Hence, the number of accesses to the ESP can be reduced which frees CPU cycles for other tasks. Figure 4.1 shows the ratio of required number of accesses with and without this feature enabled.

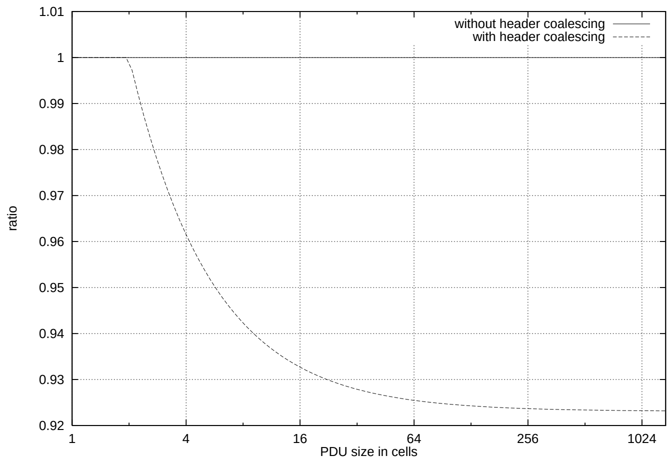

Figure 4.1: Effects of Header Coalescing

For a PDU size of 9180 bytes, header coalescing reduces the required number of accesses to the ESP to 92.3%. Hence, this feature reduces the transmit costs by nearly 8%. A similar mechanism exists for the receive path, too. Another important implication is, that cells can be transfered without any action being required in between.

4.4 Identification of Incoming Cells

During the receive process, a cell header is pulled from the receive FIFO. Solely with the cell header available a decision must be made on whether the cells belong to an open connection or not. One can think of several approaches of how to determine that:

tree-based search: This way, the whole range of VPIs and VCIs can be covered. The tree contains pointers to dynamically allocated connection descriptors. The number of connections is limited by the available memory for connection descriptors. The time it takes to lookup a certain connection depends on the number of connections. By the use of self optimizing trees the seek time for "high-traffic" connections could be minimized while incrementing the time for other connections. But, this would induce a certain level of unfairness. Furthermore, it would be necessary to investigate if inserting a node (opening a connection is not a time-critical operation unlike the lookup on cell arrival) could break the hard time limits for searching the tree. Another negative point: figuring out that an entry is not present may require walking the tree down in its full depth.

hash-based search with overflow buckets: A common approach would be to calculate a hash value from VCI and VPI as an index into a hash table with overflow buckets. Although this method could reduce the mean lookup time, it can result in heavily different lookup times. Again the lookup costs for a non-present entry can be enormous.

VPI/VCI as table index: By reducing the significant bits in the VPI and VCI field to a total of, say 10, the value gained from merging the significant bits is used as an index into an array of descriptors. This reduces the number of VPIs and VCIs the firmware can cope with. The number of connections is limited by the table size (which in turn is limited by the available memory). Using an array of pointers to dynamically allocated connection descriptors, the table size (and thus the VPI/VCI space) could be slightly enlarged while incrementing the costs for a lookup operation by an additional level of indirection.

With respect to the criteria listed in Section 3.4.2, the method described last looks most suitable for the given problem. It takes minimum and constant lookup time — no matter whether an open connection exists or not — at the price of a reduced VPI/VCI space.

4.5 Cell Discard and Packet Discard

Incoming cells get into one of four receive FIFOs in the ESP, even if no connection has been opened for those cells or a connection's buffer is filled up. In both situations the firmware should remove the cells from the receive FIFO. Therefore, the firmware reads the cells from the receive FIFO without further processing. Even in an hand-optimized loop this takes at least twelve cycles per cell.

A solution that significantly reduces the overhead associated with discarding cells could be to adjust the memory pointers of the ESP's FIFO logic. That way, a short address calculation and a single write access to an ESP register would be enough to get rid of cells. This method is most efficient for large blocks of cells. Even though the pointers are accessible from the i960, no stable mechanism was found by the time of writing. However, independent of the method used for discarding, cells classified as to be discarded produce no extra load on the host CPU.

The firmware tries to reduce the work for the host CPU. Therefore, it implements a packet discard mechanism for AAL5 PDUs. Whenever a connection's receive buffer has insufficient free space for the current PDU available, the connection is marked as disabled and all future cells except for the End-of-PDU cell are discarded. After arrival of the last cell, the connection is enabled again. Since the length of a AAL5 PDU is encoded in the *last* cell, there is no way to determine at arrival of a PDU's first cell whether the entire PDU will fit into the available buffer space. Hence, for connections with variable PDU sizes this packet discarding scheme cannot completely avoid unnecessary PCI-Bus usage, whereas with a commited fixed PDU size the firmware can begin discarding of cells even at the start of a PDU. Nevertheless, it unburdens the handling of dropped packets from the host CPU in overload situations, which is a major advantage over the previous combination of the PCA-200E driver and L4ATM.

4.6 FIFO Selection Scheme

The transmit FIFO selection scheme described in Section 3.3.2 works fine for up to three concurrent real-time connections. Scenarios with more than three concurrent real-time connections can be handled similarly as long as they can be implemented by PDU interleaving (VBR connections). For the remaining cases, two FIFOs should be associated with the two connections with the highest rate, while a cell-level scheduler multiplexes the remaining real-time connections onto the third high-priority FIFO. That approach may be limited to quite low data rates because of the PCA-200E architectural constraints. In [SG98] a feasibility study of a software-based cell-level scheduler is presented:

based on a Pentium Pro 200MHz up to 1000 concurrent connections can be shaped simultaneously. But, in contrast to the i960 on the PCA-200E, that CPU is *not* responsible for actual data movement. It runs algorithms for the scheduling of DMA transfers and cell transmits. This comparison and the estimates from Section 4.7 indicate, that software-based cell-level scheduling at high rates is likely to fail on the PCA-200E hardware.

4.7 CPU Cycles

Estimates

The i960 CPU on the PCA-200E board is clocked at 25MHz. A saturated full-duplex 155Mbps link transfers 706414 cells/s. The PCA-200E hardware allows to transfer cells as twelve 32-bit words in a burst operation (taking one external cycle per word), leaving an average of 23 cycles per cell for overall processing overhead. Even with the i960 being able to execute about two instructions per cycle, this value seems insufficient.

Call Costs

The modular design presented in Section 4.1 is strongly supported by the i960 architecture. The combination of two features, a sophisticated call-and-return mechanism saving procedure-local registers and a saved register cache with a maximum depth of sixteen, makes function calls very cheap. In a typical program, procedure calls and returns cause procedure depth to oscillate a few levels around a median call depth. Unless oscillation is larger than the number of cacheable register sets, no cache flush is required. A **call** or **return** instruction involves transfer of sixteen 32-bit registers which consumes only four clock cycles. By the use of clever instruction scheduling, up to two instructions prior to the call/return can be executed in parallel with the **call** or **return** instruction.

Execution Timings

This section lists execution times for certain tasks. Due to the lack of a timestamp counter in the i960, timings were taken by generating an interrupt after a certain number of iterations of the task. The host then calculated the duration between two successive interrupts by use of the Pentium's timestamp counter. From that the cycles for a single iteration were derived. Obviously, the values gained reflect a minimum — the i960's instruction cache combined with its 16-word prefetch buffer eventually may have reduced execution time during measurements in a way that is not achievable under normal conditions. On the other hand, running the i960 with cache disabled would increase execution times more than likely.

task	cycles
test if cells are in the any of the receive FIFOs	5
parse empty tx fifo queues	28
enqueue an internal transmit descriptor to a FIFO's empty queue	21
enqueue an internal transmit descriptor to a FIFO's queue	38
dequeue an internal transmit descriptor from a FIFO's queue	34

Table 4.1: Execution Times

Related to the estimated 23 cycles per cell these values underline the motivation of header coalescing. Header coalescing enables transfer of multiple cells in a block operation, grouping data transfer cycles together. That way "long running" operations (taking more than 23 cycles) become feasible.

4.8 Maximum Bandwidth

When people talk about network adapters, the most often asked question probably is: "How fast is it?" Thinking about that twice, there are a lot of answers to this question. In the context of ATM network adapters, interesting aspects are throughput as well as latency. Lacking suitable ATM measurement tools latency was not measured. The numbers presented in this section were obtained either by the use of interrupts and the host's timestamp counter or from a network management/monitoring tool whose precision is more than questionable.

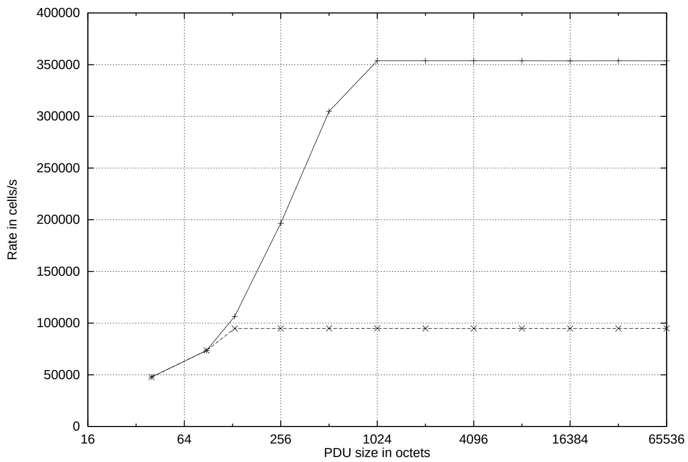

Figure 4.2: Maximum Transmit Rate

Figure 4.2 shows the maximum achievable transmit bandwidth with respect to the PDU size. The dashed line shows the values for a connection with reduced transmit rate. The degradation on small PDU sizes relates to the processing overhead in the firmware. The transmit FIFO drains off before the firmware is ready to refill the FIFO again. For a connection at link speed the overhead becomes significant for PDU sizes of less than 1024 octets. A connection with approximately 100000 cells/s can achieve its requested bandwidth even with a PDU size of three cells.

4.9 Concurrency

One of the problems addressed by DROPS is the isolation of concurrent activities. Regarding ATM connections, this means reducing mutual influences of concurrent connections. In this section the behaviour and effects of concurrent connections are investigated.

4.9.1 Concurrent Transmitters

With the ESP's transmit FIFOs the firmware is able to shape up to three concurrent outgoing real-time connections while offering the remaining bandwidth for best-effort connections. Given that there is a feasible cell schedule[1] for these connections, no mutual influences should occur. The graphs in Figure 4.3 show cell rates of four concurrent outgoing connections and their sum: C2 with 24000 cells/s, C3 with 48000 cells/s, C1 whose rate is changed and the best-effort connection C4.

During the experiment, the rate of C1 was changed. The rate of C2 and C3 remained stable as expected whereas the rate of C4, the best-effort connection, adapted to the remaining rate.

4.9.2 Concurrent Receivers

Two central ideas of this work are the isolation of inbound connections and the protection of the host system from overload situations. Connections exceeding their negotiated rate are to be policed by discarding their packets. The host's CPU utilization should not be influenced by those connections.

The setup depicted in Figure 4.4 was used for the measurements: Three machines (A, B and C) ran DROPS with an application to control the firmware. Additionally, an "idler" ran on C to measure the available CPU time. Host A generated two AAL5 streams of 8192 KByte PDUs, both with a rate of

[1] It is left to L4ATM to determine if a cell schedule exists for a set of real-time connections. This decision should be made during the admission control phase at connection setup.

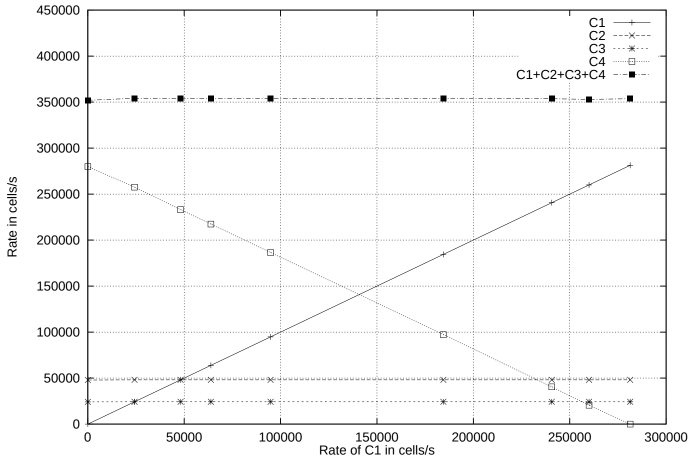

Figure 4.3: Concurrency of Outbound Connections

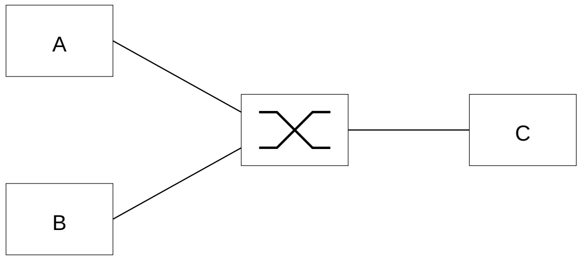

Figure 4.4: Measurement Setup

20 Mbit/s. Host B generated an AAL5 stream with varying PDU size and rate. The switch was configured with three PVCs leading to the port C was connected to. The "idler" on C repeatedly measures the number of iterations in a tiny loop in a certain number of CPU cycles. Taking the value for an unloaded CPU as reference, this allows estimation of remaining CPU time. Furthermore, the firmware was configured to request an interrupt for every PDU that was received successfully.

As expected, without any open connection the host CPU does not even notice the incoming data stream. Figure 4.5 shows utilization of the host's CPU time for a connection at link speed and for a 40 MBits/s connection, both with varying PDU size. Here, the interrupt handler on the host immediately acknowledges the PDU by advancing the tail pointer of both the receive buffer and the receive queue. Values for smaller PDU sizes are not available because of massive CRC errors, probably induced by the firmware's overhead.

The influences of misbehaving inbound connections on the host CPU are shown in Figure 4.6. Here, two connections of 20 MBit/s each and a third connection of 40 MBit/s were opened. Host A sends on the two connec-

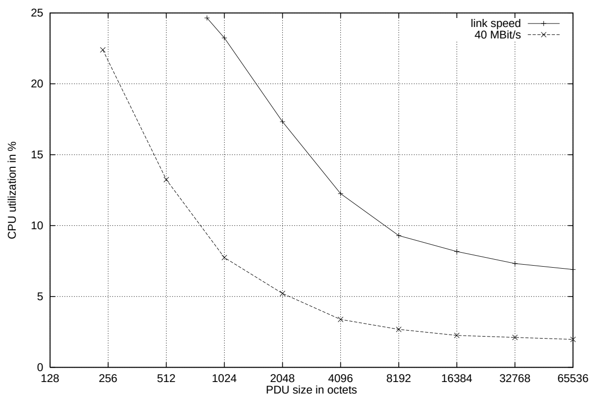

Figure 4.5: Host CPU Utilization

tions at their negotiated rate, whereas the rate of the other connection, the “hazard”, originating at B is changed.

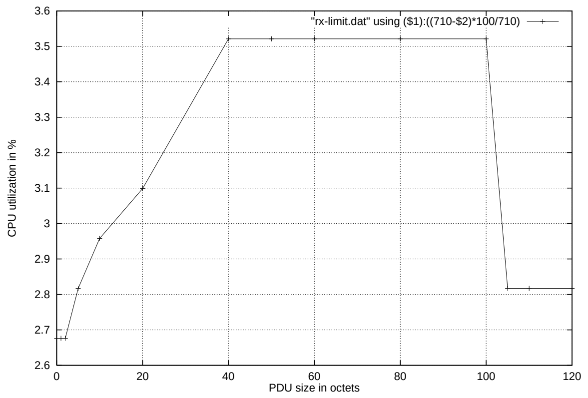

Figure 4.6: Host CPU Utilization with “hazard”

Up to a rate of 2 MBit/s no significant load can be detected. Starting with 5 MBit/s, an almost linear increase in CPU utilization can be monitored, up to the point where the actual rate exceeds the negotiated rate. Since the receive buffer is drained with a rate of max. 40 MBit/s, additional PDUs are discarded by the firmware — the host does not get an interrupt for that. Hence the CPU utilization remains constant. When the rate of the connection exceeds 100 MBit/s, a remarkable decrease in CPU utilization can be seen. This is obvious: all three data streams are joined on C’s switch port, which gets saturated at about (20 + 20 + 100) MBit/s. Due to cell discard strategies in the switch, cells of the hazardous connection are dropped. This leads to damaged PDUs, which in turn pose no load to the C’s CPU.

Chapter 5

Summary

This work aimed at design and implementation of a firmware for the PCA-200E ATM network adapter. With respect to the demands of real-time systems, and that of DROPS in special, a solution was presented.

The firmware supports traffic shaping on the transmit path, differentiates between real-time and best-effort connections, and allows implementation of zero-copy transmit and receive paths. It uses a simple but powerful policing mechanism to protect the host from misbehaving inbound connections. As shown in Chapter 4, it provides effective methods to minimize the work for the host CPU. Paired with L4ATM this firmware offers resource-saving real-time communication via ATM.

Although the design was aligned to L4ATM's requirements, the general structure allows easy adaption to different host systems. That covers changes to the buffer mechanism as well as different strategies for host notification via interrupts.

5.1 Future Work

There is still room for optimization. Although `egcs`, the cross-compiler used for this project, does a quite good job on optimizing, sometimes better code can be generated with some hints from the programmer (explicit register variables, statement reordering, etc.). To make the firmware's buffer management more flexible and to allow application of cache-partitioning schemes, scatter-gather buffers should be implemented for both receive and transmit buffers. Implementation of other AALs or even higher layer protocols in the firmware may be desirable, too.

5.2 Acknowledgements

I would like to thank the members of the Operating Systems chair at the Dresden University of Technology who helped bringing this work to a successful end, in particular Prof. Hermann Härtig, Jork Löser, Volkmar Uhlig and Lars Reuther for helpful discussions, Sebastian Schönberg and Jean Wolter for repeatedly reading and correcting this paper, and last but not least my parents for their understanding and their support during this busy time.

Chapter 6

Appendix

Configuration Descriptor

The Configuration Descriptor is used during initialization to set up queue lengths and the number of valid bits for VPI and VCI. Its structure is given in Figure

31 0	
command queue length	+0
transmit queue length	+1
event queue base	+2
event queue length	+3
number of VPI bits	+4
number of VCI bits	+5
address of result buffer	+6

Figure 6.1: Configuration Descriptor

Open Command Descriptor

The Open Command is used to enable reassembly of incoming data for a VPI/VCI pair into the buffer. Figure 6.2 presents the structure of the Open Command Descriptor.

31 … 0	
VPI	+0
VCI	+1
AAL	+2
receive buffer base	+3
receive buffer size	+4
receive queue base	+5
receive queue length	+6
handle	
address of result buffer	

Figure 6.2: Open Command Descriptor

Close Command Descriptor

The Close Command is used to disable reassembly of incoming data for a VPI/VCI. Figure 6.3 presents the structure of the Close Command Descriptor.

31 … 0	
VPI	+0
VCI	+1
handle	+2
address of result buffer	+3

Figure 6.3: Close Command Descriptor

Transmit PDU Descriptor

The driver sets up transmit PDUdescriptors in host memory, according to the structure given in Figure 6.4.

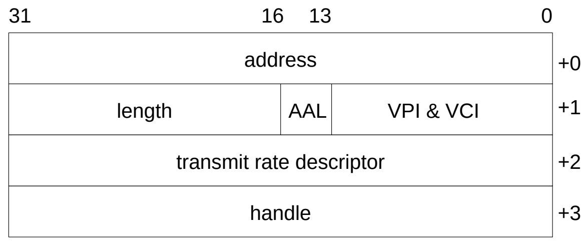

Figure 6.4: transmit PDU descriptor

Receive PDU Descriptor

The receive queue contains entries with a structure as shown in Figure 6.5. Since there is a private receive queue for connection, no information describing the VC is required.

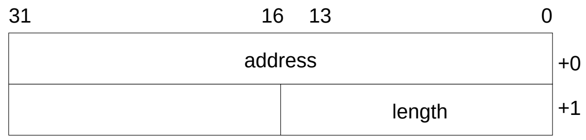

Figure 6.5: receive PDU descriptor

Bibliography

[Alm96a] Werner Almesberger. *Linux ATM API, Draft, version 0.4*. Laboratoire de Réseaux de Communication (LRC), Ecole polytechnique fédérale de Lausanne (EPFL), CH-1015 Lausanne, Switzerland, July 1996.

[Alm96b] Werner Almesberger. *Linux ATM device driver interface, Draft, version 0.1*. Laboratoire de Réseaux de Communication (LRC), Ecole polytechnique fédérale de Lausanne (EPFL), CH-1015 Lausanne, Switzerland, February 1996.

[Alm99] Werner Almesberger. ATM on Linux. `http://icawww1.epfl.ch/linux-atm/`, 1997 – 1999.

[ATM94] The ATM Forum. *ATM User-Network Interface Specification, Version 3.1*, September 1994.

[ATM96] The ATM Forum Technical Commitee. *Traffic Management Specification Version 4.0*, April 1996.

[Ben95a] Michael Benson. *ESP: An Enhanced SAR Processor*. FORE Systems, Inc., 1000 FORE Drive, Warrendale, PA 15086-7502, March 1995.

[Ben95b] Michael Benson. *PBI — PCI Bus Interface*. FORE Systems, Inc., 1000 FORE Drive, Warrendale, PA 15086-7502, November 1995.

[Bor99] Martin Borriss. *Operating Systems Support for Predictable High-Speed Communication*. PhD thesis, Dresden University of Technology, 1999.

[Cor96] Cornell University. U-Net — A User-Level Network Interface Architecture. `http://www2.cs.cornell.edu/U-Net/Default.html`, 1996.

[Dan98] Uwe Dannowski. An ATM Driver for DROPS. Großer Beleg, Dresden University of Technology, June 1998.

[Dre99a] Dresden University of Technology. DROPS — Dresden Realtime Operating System. `http://os.inf.tu-dresden.de/drops/`, 1996 – 1999.

[Dre99b] Dresden University of Technology. PCA-200E Linux Driver. `http://os.inf.tu-dresden.de/pca200e/`, 1997 – 1999.

[FOR97] FORE Systems, Inc., 1000 FORE Drive, Warrendale, PA 15086-7502. *Programmer's Reference Manual for AALI Interface*, May 1997.

[Int94] Intel Corp., Santa Clara. *i960 CA/CF Microprocessor User's Manual*, March 1994.

[KSL99] Fred Kuhns, Douglas C. Schmidt, and David L. Levine. The Design and Performance of a Real-time I/O Subsystem. In *Fifth Real-time Technology and Applications Symposium*, Vancouver, British Columbia, Canada, June 1999.

[Lie96] J. Liedtke. L4 reference manual (486, Pentium, PPro). Arbeitspapiere der GMD No. 1021, GMD — German National Research Center for Information Technology, Sankt Augustin, September 1996. Also Research Report RC 20549, IBM T. J. Watson Research Center, Yorktown Heights, NY, Sep 1996; available from URL: `ftp://borneo.gmd.de/pub/rs/L4/l4refx86.ps`.

[PCI95] PCI Special Interest Group, Portland, OR 97214. *PCI Local Bus Specification*, 2.1 edition, June 1995.

[PMC96] PMC-Sierra, Inc. *PM5346 S/UNI-155-LITE, SATURN User Network Interface 155.52 & 51.84 Mbit/s, issue 6*, March 1996.

[Sch98] Sebastian Schönberg. PCI Bus Performance and Real-Time Influences. In *First Workshop on PC Performance*. ACM, Oct 1998.

[SG98] J. Schiller and P. Gunningberg. Feasibility of a Software-based ATM cell-level scheduler with advanced shaping. In *Broadband Communications '98*, Stuttgart, Germany, April 1998.

[Tan90] A.S. Tanenbaum. *Betriebssysteme - Entwurf und Realisierung*. Carl Hanser Verlag, München, 1990.