Computer Science

Mattias Ahlberg and Fredrik Carlsson

Serial RP-Bus in workstation -

A bridge between simulated and real AXE-10 hardware

Bachelor's Project

2001:25

Serial RP-Bus in workstation -

A bridge between simulated and real AXE-10 hardware

Mattias Ahlberg and Fredrik Carlsson

This report is submitted in partial fulfillment of the requirements for the Bachelor's degree in Computer Science. All material in this report which is not our own work has been identified and no material is included for which a degree has previously been conferred.

Mattias Ahlberg

Fredrik Carlsson

Approved, 6 June 2001

Advisor: Stefan Alfredsson

Examiner: Stefan Lindskog

Abstract

When writing software for large computer systems it is often easier, more secure, cheaper and faster to test and debug software on a simulated platform before deploying it to the real hardware. Ericsson Infotech, department of Test Support and Simulated Platforms (TSP) develop such a simulator platform, called Simulator Environment Architecture (SEA). This platform is mainly used for simulating the AXE 10 digital switching system, which is also developed by Ericsson. The AXE 10 system is the heart of the telephone network in Sweden and many other countries.

SEA is a very easy and practical tool for testing software for the AXE 10 switch, but sometimes some of the software that is to be tested has to be run on real target hardware. To be able to achieve this a bridge between the simulated environment and the real hardware is needed, and this is the purpose of this project. The main goal is to investigate if it is possible to connect a real serial Regional Processor (RP) to a Central Processor (CP) simulated in SEA. Both the RP and the CP are parts of the hardware in a real AXE 10 switch. The secondary goal is, if possible, to implement the communication between the real RP hardware and the CP, simulated in SEA.

To accomplish these goals, the hardware for the two communication buses between the CP and RP were first investigated. This hardware investigation revealed four possible solutions, but only two of these solutions were considered suitable for this project. The first possible solution was to use a Regional Processor Handler Magazine Interface (RPHMI) card in the workstation running SEA, and then connect a Serial Regional Processor Bus Handler (RPBH-S) to this card. Then it should be possible to connect real RPs to the RPBH-S. This solution was however considered as being too complex and expensive. The second solution, which was finally chosen, was to use the SERPENT RPB-S emulator. The SERPENT is a piece of hardware that was made especially for connecting RPs to a workstation, and it was therefore ideal for this project. Thus the primary goal with the project was accomplished. The secondary goal was not entirely completed, but an early prototype was designed and partly implemented.

Acknowledgments

We would like to start with thanking our supervisors at Ericsson, Claes Giljegård and Per Nordh, without them this project would have been hard to carry out. Our University supervisor Stefan Alfredsson has also been of great help.

Second we would like to thank the Ericsson people at LMF in Finland who lent us the SERPENT RPB-S emulator and for showing interest in this project. There are also a few guys at UAB in Stockholm and EPA in Australia that has helped us with finding documents about the hardware in the AXE switch.

Table of Contents

List of Figures

List of Tables

1 Introduction

The Department of Test, Support and Simulated Platforms (TSP, also called EIN/T) at Ericsson Infotech AB in Karlstad develops, among other things, a simulator environment for telecom systems. The simulator makes it possible to test software without access to target hardware. It is also Ericsson Infotech that has initiated this project which purpose is to determine if it is possible to connect real hardware to their simulated environment.

1.1 Introduction to simulation

Simulation is the process of designing a model of a real or imagined system and then using the model to perform experiments. Models have been constructed for nearly every system imaginable, such as flight dynamics, integrated circuits and embedded computer systems. In each of these systems, a model has proven to be more cost efficient, less dangerous, faster and more practical than experimenting with the real system.

The general purpose of a simulation is to allow an analysis of a systems capabilities and behaviour without constructing of, or experimenting with the real system. It can also be used for educational purposes and demonstrations.

1.2 Report structure

In some chapters of this report there are a lot of abbreviations, which can be unfamiliar for most readers. The abbreviations are listed in appendix A .

Chapter two is an overview of the problem that this project is faced with and thereby it gives the purpose for this project. When reading **chapter three** the reader will be given an introduction to the AXE-10 digital switching system and some of its components. Among the described components are the Serial Regional Processor and its communication bus. In **chapter four** the reader will be given a brief description of the Simulator Environment Architecture software (SEA), developed by the EIN/T department. Furthermore, **chapter five**

describes the communication hardware in the Regional Processor, Regional Processor Handler and the Regional Processor Handler Bus interface in the Central Processor. It is also an investigation of different products that could be used to communicate on the RPB-S. **Chapter six** is a complete description of the system design. It gives a detailed description of how the system will be implemented. The last part of this report, **chapter seven**, is a conclusion of this work and it is also a personal reflection from the writers of this report.

Chapter three to six will end with a short review of the most important subjects described in the chapter.

2 Background

This chapter gives an overview of the problem that this project faces and thus, its purpose.

As mentioned in the introduction the department EIN/T develops a simulator for computer systems and it is called Simulator Environment Architecture – SEA. It is mainly used to simulate the AXE-10 system, which is the heart of the telephone network in Sweden and many other countries. In section 2.1 and 2.2, a few components in the AXE-10 system are mentioned. These components are described in chapter 3.

2.1 Problem

For testing purposes, SEA is a very easy and practical way of testing software but sometimes the software that is to be tested has to be run on real target hardware. The Regional Processor (RP) software is one of these but so far it has not been possible to do this in an uncomplicated way.

2.2 Purpose

To make it possible to mix simulators and real hardware, such as the Regional Processor, bridges between simulated and real hardware are needed.

The purpose with this project is to find out if it is possible to use a commercially available communication card or some other device for a PC or a Sparc workstation to implement a server for the Serial Regional Processor Bus (RPB-S). To do this it is necessary to investigate the hardware in the RP and see if it is possible to find some communication hardware that matches with its physical interface.

If it is possible to find a suitable card or device the purpose is also to implement the RPB-S protocol stack on the selected hardware.

2.3 Method

Since this project is dealing with simulated environments the following chapter explain how the real hardware works. After that it is explained how the simulator is constructed followed by an investigation of different hardware solutions that could be used.

3 Brief overview of the AXE-10 system

This chapter introduces the AXE-10 digital switching system and some of its components. Among the described components are the Serial Regional Processor and its communication bus.

To increase the understanding for this project and the Simulator Environment Architecture (SEA) platform it is important to have basic knowledge of the AXE-10 digital switching system and its components.

3.1 What is the AXE-10 digital switching system

The AXE-10 is a digital switching system for telecommunication, which is designed and developed by Ericsson. It handles among other things Public Switched Telephone Networks (PSTN) that is used for stationary telephones, common in every home. It also handles Integrated Services Digital Networks (ISDN) and Public Land Mobile Networks (PLMN)[5].

3.2 Components in the AXE-10 system

A part of the AXE-10 hardware platform can be seen in Figure 3.1. This figure gives a good overview of some of the components in the AXE-10 digital switching system. The components that increase the understanding of this report are also described below.

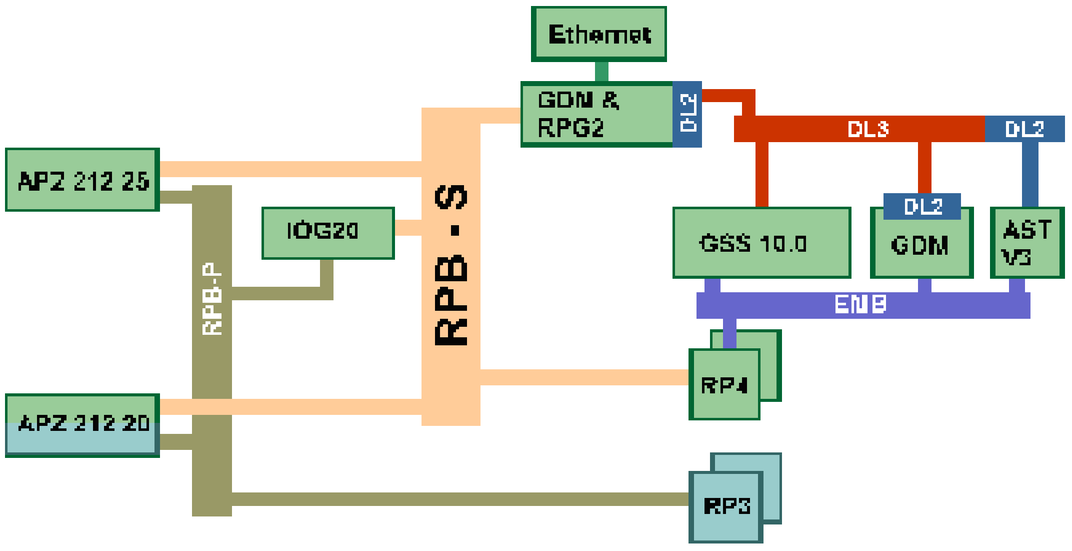


Figure 3.1 - A part of the AXE-10 hardware platform

As mentioned in chapter 2, this projects purpose is to communicate on RPB-S. In Figure 3.1 it can be seen that this bus is located between the APZ and the RP4. In the figure there are also some other devices connected to the RPB-S, but the function of these components are not relevant to this report.

3.2.1 APZ

The APZ, also called Central Processor (CP), is the control system of the AXE-10 platform. As shown in Figure 3.1 there are several models of the APZ but it is only APZ 212 20 and higher that supports the Serial Regional Processor Bus (RPB-S). For higher safety the CP is replicated in each APZ, one is called CP-A and the other CP-B, see Figure 3.2. Each of the CPs has its own RPB-S that can be seen in Figure 3.2.

Furthermore each CP is divided into two parts, the Instruction Processing Unit (IPU) and the Signal Processing Unit (SPU). It is the SPU that communicates with the Regional Processor Handler (RPH), which controls the RPs. This communication is described in detail below.

3.2.2 Regional Processor (RP)

The Regional Processors performs processing-intensive protocol handling and protocol conversions as well as routine repetitive processing tasks. Their main task is to relieve the CP of these simple real-time demands as well as handling low-level protocols.

The are several models of RPs but only some of them have been modified to incorporate the RPB-S interface [4]. Among these are the RP4, the RPV2 and the RPG2. Each one of these handles different protocol conversions and in Figure 3.2 these RPs are shown.

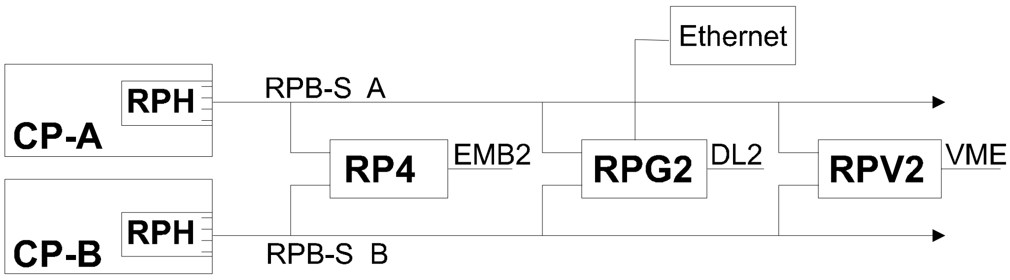


Figure 3.2 - Overview of RPB-S and compatible components

3.2.3 Serial Regional Processor Bus (RPB-S)

In the AXE-10 system there are several components and they all communicate through different Ericsson proprietary communication standards. One of the communication paths is the serial RP-Bus that is used for communication between a CP and the RPs. This bus is designed to replace the old parallel RP-Bus, but the two busses can also coexist. The RPs provides the AXE-10 applications with a processor platform and the communication between the CP, RPs and user applications in extension modules [4]. The extension modules and the parallel RP-Bus will be given no further explanation because they are not relevant to this report.

The protocol that is used for communication over the RPB-S is organized in three layers, equivalent to the three lower layers in the OSI model [3] (see Figure 3.3). The two lowest layers are the physical and data link layers which both are implemented in hardware.

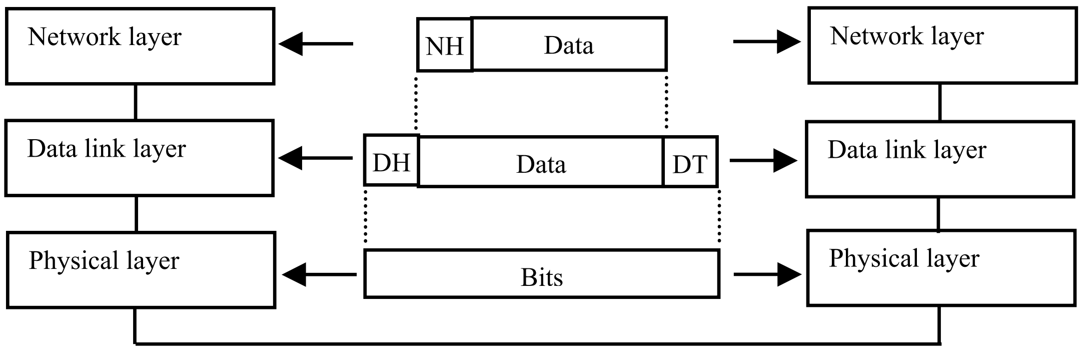


Figure 3.3 - The three lowest layers in the OSI model

The physical layer is based on Low Voltage Differential Signalling (LVDS, will be described in detail below) and uses Non Return to Zero Inverted (NRZI-L, where L means invert on zero) line coding. With this coding a logical one is coded as no change, and a zero is coded as a level shift [12], see Figure 3.4.

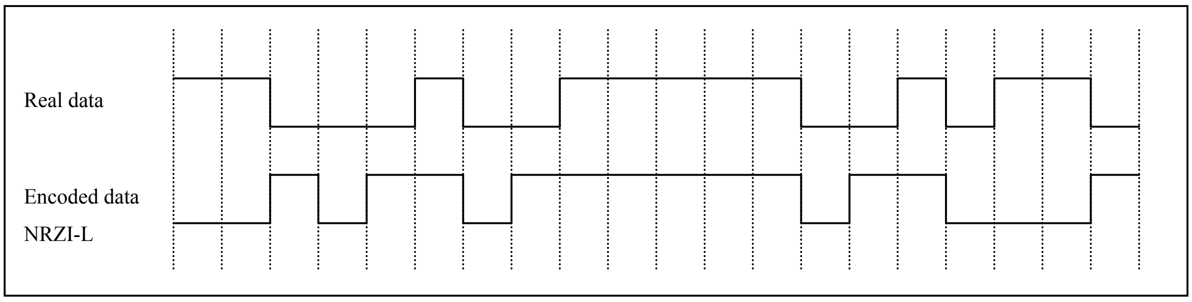


Figure 3.4 - Some sample NRZI-L data encoding

The data link layer is implemented according to the standard protocol High-level Data Link Control - Normal Response Mode (HDLC-NRM). The framing structure can be seen in Table 3.1 [3].

Data link header			Data	Data link trailer	
Opening flag	Address field	Control field	Information field	FCS	Closing flag
8 bits	8 bits	8 bits	n bytes	16 bits	8 bits

Table 3.1 - Framing structure of the data link layer

The purpose of the opening and closing flag is to delimit the frame, and for this to be possible they must be unique. To accomplish this, the entire frame (except the opening and closing flags) must be bit stuffed. This means that if a sequence of bits that consists of five or more bits are detected, i.e. looks like the opening or closing flag, a zero is inserted after the fifth one, this is to ensure that the starting and closing flags remain unique. An example of this can be seen in Figure 3.5.

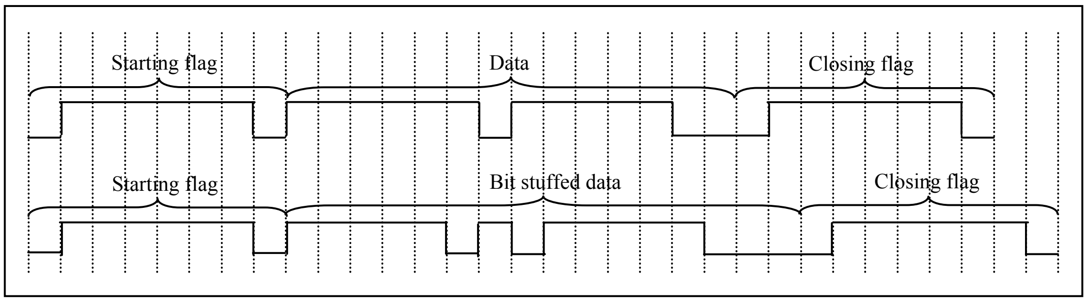


Figure 3.5 - Example of bit stuffing

The third and the highest layer is the network layer and its purpose is to send signals between central and regional program blocks [2]. The framing structure of the network layer can be viewed in Table 3.2.

Header	Information
Payload type	Data
8 bits	n bytes

Table 3.2 - Framing structure of the network layer

The contents of the information field is different depending on what payload type the frame has. At the moment the following payload types are defined [2], see Table 3.3.

Number	Type
0	Unpacked RP signals
1	Packed RP signals
2	Unpacked EMRP signals
3	Packed EMRP signals

Table 3.3 - Payload types in the network layer

3.2.4 Serial Regional Processor Handler (RPH)

Each CP has a serial RPH and its main function is to send and receive signals on the RPB-S. In other words it is an interface for the CP subsystem to the RP subsystem. Because there are two RPB-S, the two RPHs must decide which bus that should be active [4]. It must also make sure that both CPs get the signal from the active bus, see Figure 3.6. As with the APZ the reason for replicated components is to get a higher fault tolerance.

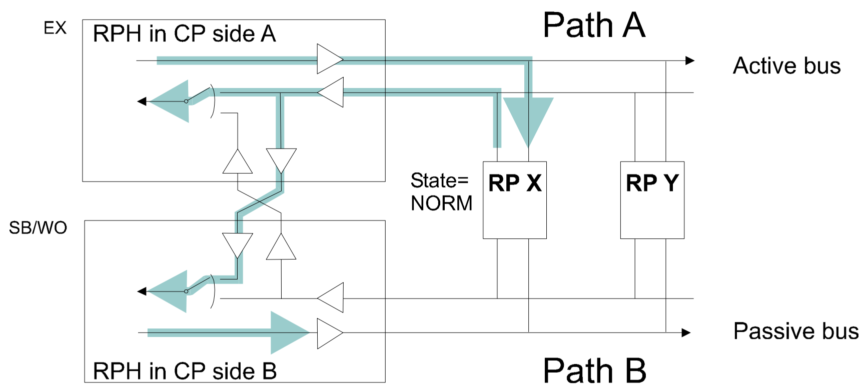


Figure 3.6 - Example of communication case

Furthermore the RPH consists of up to 64 Serial Regional Processor Bus Handlers (RPBH-S) but the CP software allows only 32 RPBH-S. The RPBH-S are placed in RPH Magazines (RPHM) in groups of eight. Therefore it can be up to four RPHMs connected to one SPU. Each of these RPBH-S handles up to 32 RPs that gives a maximum of 1024 RPs [1].

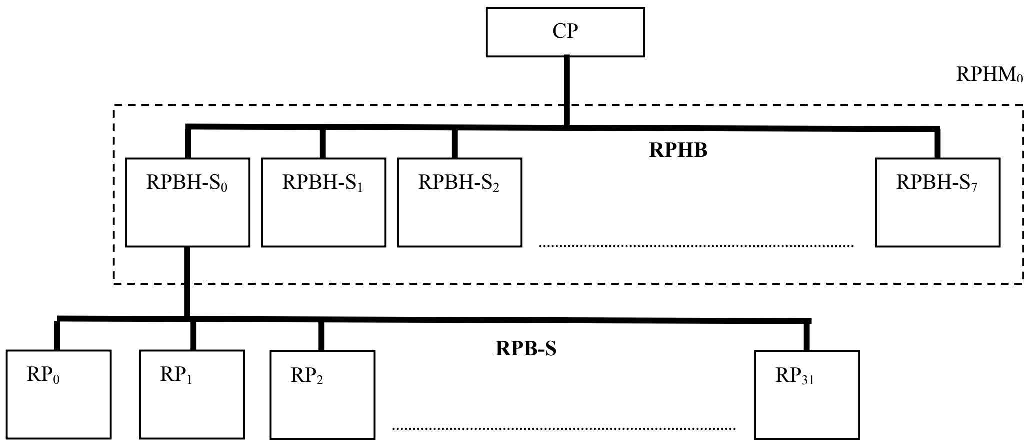


Figure 3.7 - Overview of the RPHB and RPB-S

3.2.5 Regional Processor Handler Bus (RPHB)

As shown in Figure 3.7 the RPHB is located between the CP and the RPBH-S. The purpose with the RPHB is to connect the RPBH-S with the CP. As in the case of RPs there exist both a serial and a parallel version of RPBH-S and both can be connected to the same RPHB. The only dissimilarity is that they use different versions of the RPHB protocol.

The RPHB-S protocol is half duplex [8]. That means that data only can be transferred in one direction at a time, and the RPBH-S is only allowed to drive the bus when the CP tells it to. Everything that the CP sends to the RPBH-S is an order, and each order consists of one ore more 16-bit words. An order can either contain a signal that should be sent to one or more RPs, or it can be a direct order to the RPBH-S. An order can also be directed to one, all or a group of RPBH-S. The orders are then called Local, Global and Group respectively. The general format of an order can be seen in Table 3.4.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Order code						RPBH-S group number							RPBH No.		
Number of bytes to follow (N)															
Data byte 2								Data byte 1							
Data byte 4								Data byte 3							
…								…							
Byte not valid if N is odd, or data byte N if N is even								Data byte N if N is odd or N-1 if N is even.							

Table 3.4 – General order format to RPBH-S

The order code specifies the type of order. The different order codes recognized by RPBH-S can be seen in Appendix B .

Every message that an RPBH-S sends towards the CP is either a search answer, an acknowledgement or a normal data message [8]. The general message format can be seen in Table 3.5.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
L	Message code					RPBH-S Address									
Number of bytes to follow (N)															
Data byte 2								Data byte 1							
Data byte 4								Data byte 3							
…								…							
Byte not valid if N is odd, or data byte N if N is even								Data byte N if N is odd or N-1 if N is even.							

Table 3.5 - General message format from RPBH-S

The L bit is used to indicate a long RP signal in message from RPBH-P and is always set to 0 in RPBH-S. The message code specifies the message type, and the different types allowed are listed in Appendix B .

3.3 Chapter review

- The AXE-10 is a digital switching system for telecommunication, which is designed and developed by Ericsson.
- The APZ, also called the Central Processor (CP) is the control system of the AXE-10 platform.
- Each CP has a Regional Processor Handler (RPH) that consists of up to 64 Serial Regional Processor Bus Handlers (RPBH-S). Their main task is to send and receive signal on the Serial Regional Processor Bus (RPB-S).
- Signals between the CP and the RPH are sent on the Regional Processor Handler Bus (RPHB).
- A Regional Processor (RP) relives the CP of simple real time repetitive processing tasks.
- Signals between the RPBH-S and the RP are sent on the RPB-S.

4 Simulator Environment Architecture (SEA)

This chapter briefly describes the Simulator Environment Architecture software, developed by the EIN/T department.

As mentioned in chapter two, SEA is a simulator software that makes it possible to simulate the AXE-10 digital switching system. When simulating the AXE-10 system it is possible to run real AXE-10 software dumps (see section 4.3) in the simulated environment. This makes it easier and cheaper to test, debug and evaluate new or existing software.

Currently SEA is running on Sun workstations with Solaris but it is being ported to Linux.

4.1 SEA layers

SEA is built up using a layer model and each layer is constructed of components. There are three layers, each having its own component type. The defined layers are [16] (see Figure 4.1):

- **osCore** – This is the abstraction of the host operating system
- **simCore** – This layer contains generic simulation components of common interest to the system.
- **appCore** - In this layer system specific components can be found.

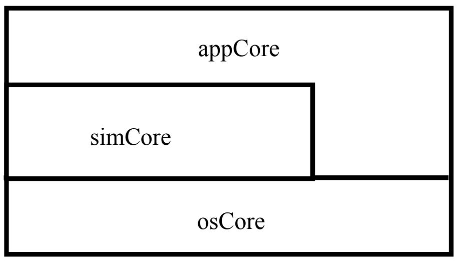


Figure 4.1 - The SEA layers

4.1.1 The osCore

The initial layer is the osCore. Its purpose is to isolate operating system specific services and thereby providing low-level virtual operating system functions. Some of the services provided are memory management, process handling, signal handling, SEA component manager services and socket handling.

4.1.2 The simCore

Above the osCore is the simCore (see Figure 4.1). It provides services needed to configure and control simulation components. Some of these services are functions for message handling, thread scheduler, event handling and real time management. This layer also provides user interface functions, such as HTTP server and Tcl interpreter. The Tcl interpreter evaluates commands sent to a specific component in SEA through the user interface, see section 4.3.

4.1.3 The appCore

The last of these three layers is the appCore. This is where components for the specific simulated computer system can be found. This makes it possible to simulate almost any computer system, but it is now mostly used for AXE-10 system simulations.

4.2 SEA components for the AXE-10 system

SEA is based on the Microsoft® Component Object Model (COM) and this makes it possible to incorporate future standard products implementing COM. With this model arbitrary computer simulation environments can be built up, using components [16]. Each of these components have different interfaces, which are used for communication with other components. When SEA is being initialised it asks every component what interfaces it has and uses this information when it connects the current simulation components to each other.

A few of the components simulated in SEA are the CP, RPBH-S and the RP. All these components are described in chapter 3. There are a lot more AXE-10 components for SEA, but these are outside the scope of this report and will therefore be given no further explanation.

The real hardware components communicate using two different buses, RPHB between CP and RPBH-S, and RPB-S between RPBH-S and RP. In SEA this communication is accomplished by a client/server model in the different components. For example the CP component implements a server for the RPHB (IRphbServer), and the RPHB implements the client part (IRphbClient) of the connection. The communication between the RPHB and the RP works in the same way, see Figure 4.2. As mentioned above, components as the CP and RPBH-S uses interfaces to connect and communicate with each other. IRphbServer and IRphbClient are two examples of such interfaces.

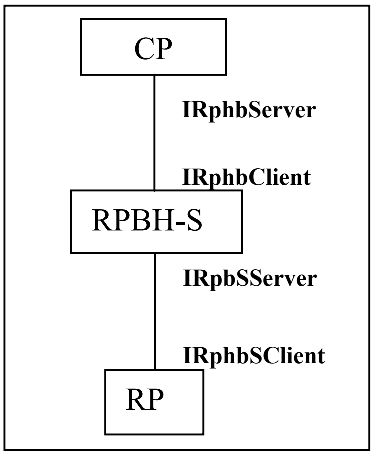
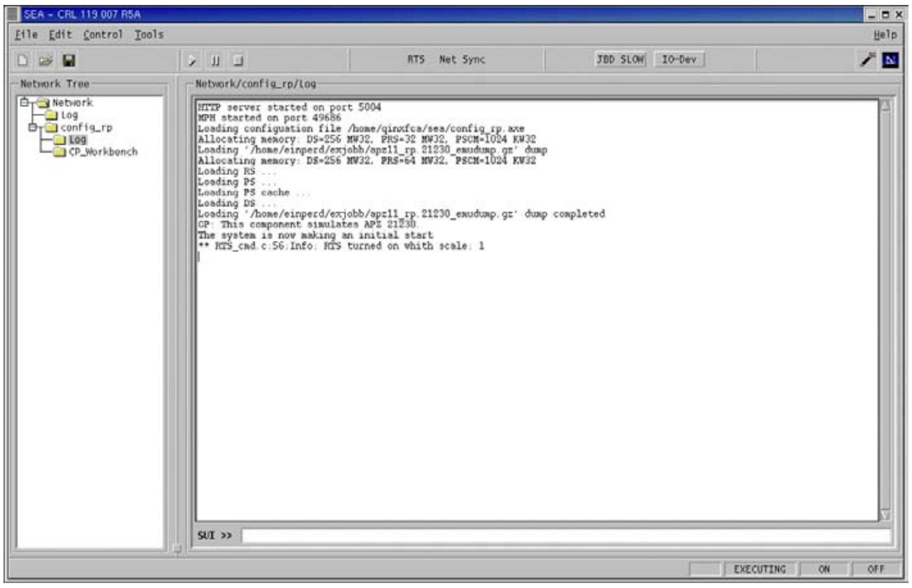


Figure 4.2 - Some AXE-10 interfaces in SEA

4.3 SEA configuration and user interface

SEA is configured using a special configuration file. This file specifies which hardware components that should be simulated, and which interfaces that should be used in the communication between them (see Figure 4.2 for some examples of components and interfaces). SEA must also know which software dump to use. The software dump is a file that contains all the software needed, and it is the same file that is used on a real AXE 10 switch. From this file SEA can auto generate a configuration file that will specify the same components as is in the real AXE 10 switch. An example of a configuration file could be seen in appendix C **.**

Both the software dump file and the configuration file names could be given to SEA as arguments. Further arguments that could be given to the program are –RTS to enable Real Time Simulation and –START to automatically start the system. If all these arguments are specified, the log window in the user interface for SEA will look similar to Figure 4.3. The configuration file used here is called config_rp.axe (can be seen in appendix C) and the software dump is called apz11_rp.21230_emudump.gz. These files specify that SEA should simulate an AXE 10 switch, which is equipped with a CP (model APZ 212 30), an RPBH-S and five RPs.



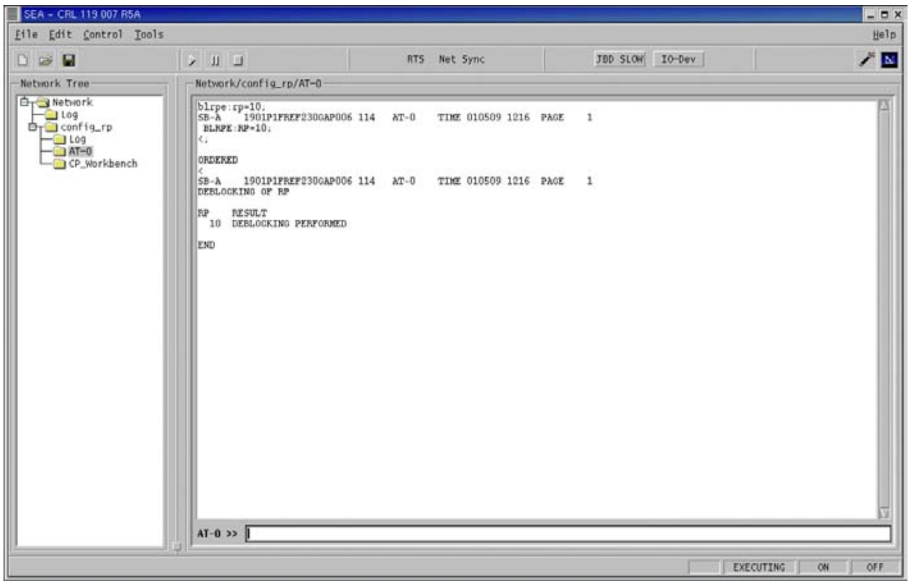
```
HTTP server started on port 5004
MPH started on port 49686
Loading configuation file /home/qinxfca/sea/config_rp.axe
Allocating memory: DS=256 MW32, PRS=32 MW32, PSCM=1024 KW32
Loading '/home/einperd/exjobb/apz11_rp.21230_emudump.gz' dump
Allocating memory: DS=256 MW32, PRS=64 MW32, PSCM=1024 KW32
Loading RS ...
Loading PS ...
Loading PS cache ...
Loading DS ...
Loading '/home/einperd/exjobb/apz11_rp.21230_emudump.gz' dump completed
CP: This component simulates APZ 21230.
The system is now making an initial start
** RTS_cmd.c:56:Info: RTS turned on whith scale: 1
```

Figure 4.3 - SEA log window

In the bottom of the log window there is also a command tool called SUI (Simple User Interface). This can be used e.g. for component configuration.

Another useful window in the SEA user interface is the I/O-device window. This window has the same function as the I/O terminal in the real AXE 10 switch, i.e. to send commands to the CP and to view output from it. The command language used is called MML (Man Machine Language). The I/O device window can be seen in Figure 4.4. This figure also shows the usage of the MML command BLRPE, which will order the CP to try to deblock the specified

RP (RP number 10 in this case). If the command succeeds the RP will be in deblocked state and therefore be able to send signals on the bus.

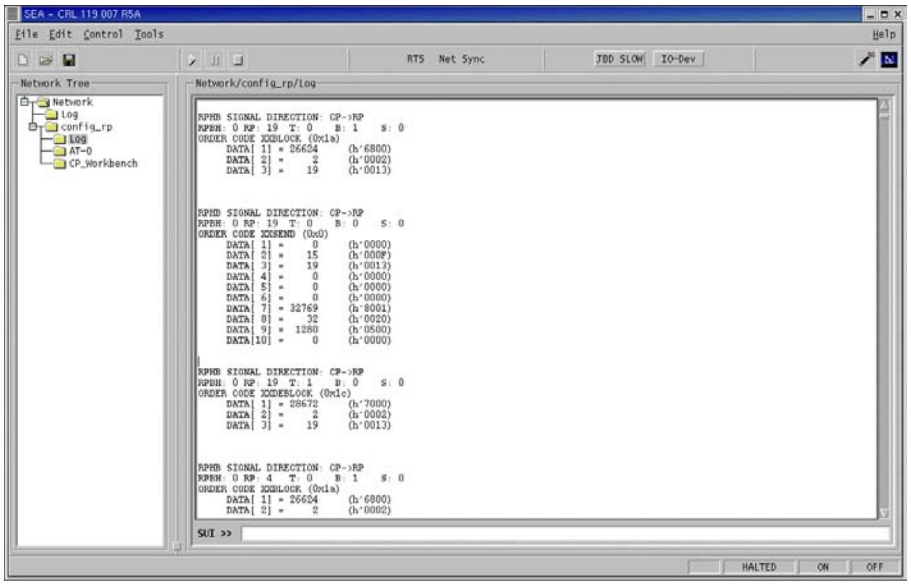


```
blrpe:rp=10;
SB-A    1901P1FREF230GAP006 114   AT-0    TIME 010509 1216  PAGE    1
 BLRPE:RP=10;
<;

ORDERED
<
SB-A    1901P1FREF230GAP006 114   AT-0    TIME 010509 1216  PAGE    1
DEBLOCKING OF RP

RP    RESULT
  10  DEBLOCKING PERFORMED

END
```

Figure 4.4 - The I/O device window in SEA

One of the most common tasks in SEA is to trace signals between different components. This can be done in many ways, and between several components. Here follows an example of how to trace the signals on the RPHB between the CP and the RPHB-S.

First the tracing must be activated. This is done in the SUI (described above) by sending a command to the simulated RPBH-S component. The commands sent are *sigtrace on rp2cp* for activating the trace in the RP to CP direction, and *sigtrace on cp2rp* for activating the trace in the CP to RP direction. When this is done all signals going through that RPBH-S will also be printed in the log window. An example of this can be seen in Figure 4.5. This figure also shows some of the signals sent between the CP and the RPBH-S, when trying to deblock an RP after issuing the BLRPE command described above.



```
RPHB SIGNAL DIRECTION: CP->RP
RPBH: 0 RP: 19  T: 0    B: 1    S: 0
ORDER CODE XXBLOCK (0x1a)
     DATA[ 1] = 26624      (h'6800)
     DATA[ 2] =     2      (h'0002)
     DATA[ 3] =    19      (h'0013)


RPHB SIGNAL DIRECTION: CP->RP
RPBH: 0 RP: 19  T: 0    B: 0    S: 0
ORDER CODE XXSEND (0x0)
     DATA[ 1] =     0      (h'0000)
     DATA[ 2] =    15      (h'000F)
     DATA[ 3] =    19      (h'0013)
     DATA[ 4] =     0      (h'0000)
     DATA[ 5] =     0      (h'0000)
     DATA[ 6] =     0      (h'0000)
     DATA[ 7] = 32769      (h'8001)
     DATA[ 8] =    32      (h'0020)
     DATA[ 9] =  1280      (h'0500)
     DATA[10] =     0      (h'0000)

RPHB SIGNAL DIRECTION: CP->RP
RPBH: 0 RP: 19  T: 1    B: 0    S: 0
ORDER CODE XXDEBLOCK (0x1c)
     DATA[ 1] = 28672      (h'7000)
     DATA[ 2] =     2      (h'0002)
     DATA[ 3] =    19      (h'0013)


RPHB SIGNAL DIRECTION: CP->RP
RPBH: 0 RP: 4   T: 0    B: 1    S: 0
ORDER CODE XXBLOCK (0x1a)
     DATA[ 1] = 26624      (h'6800)
     DATA[ 2] =     2      (h'0002)
```

Figure 4.5 - Example of signal trace output in the log window

4.4 Chapter review

- Simulator Environment Architecture (SEA) is a simulator software that makes it possible to simulate the AXE-10 digital switching system.
- SEA is built up using a three-layered model. The layers are the osCore, simCore and appCore.
- SEA uses the Microsoft® Component Object Model (COM). With this model arbitrary computer simulation environments can be built up, using components.
- In SEA it possible to simulate almost every component in the AXE-10 system such as the RP and CP.

5 Hardware investigation

This chapter describes the communication hardware in the RP, RPH and the RPHB interface in the CP. It is also an investigation of different products that can be used to communicate on the RPB-S.

5.1 RPB-S Electrical Interface in the RP

The RPB-S has a balanced differential line interface, which makes it possible to have multipoint party line communication on two shielded four wire twisted-pair cables. This means that more than one RP can be connected to the same RPB-S.

In order to communicate on the RPB-S several components are needed. These components can be viewed in Figure 5.1.

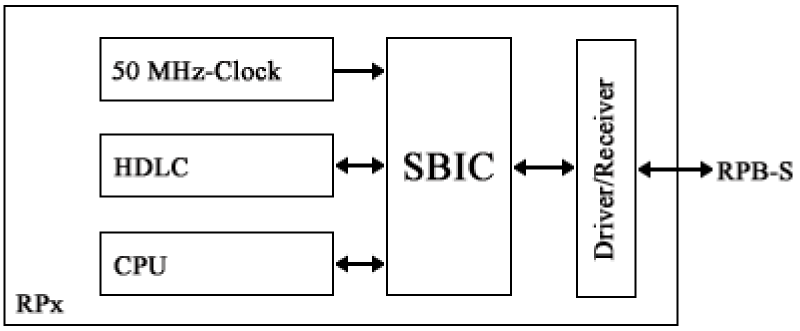


Figure 5.1 - Electrical Interface Overview

5.1.1 Driver/Receiver circuit

This circuit is designed for applications requiring high data rates and to accomplish this the circuit uses low voltage differential signalling (LVDS). Advantages with this design are that the circuit suffers very low power dissipation and has very low noise. With this design data rates up to 400 Mbps can be accomplished [17][18].

The driver circuit translates low voltage TTL/CMOS (typical 3,3 V) input levels into low voltage (typical 350mV) differential output levels. Analogous the receiver translates the low voltage differential input to low voltage TTL/CMOS output levels.

To be able to support multiple stations on the same bus the driver/receiver circuits also supports the TRI-STATE function, which disables the output.

The circuit is manufactured by Ericsson, but a comparable circuit is also commercially available from National Semiconductors and it is called DS90LV031A [15].

5.1.2 Serial Bus Interface Circuit (SBIC)

As shown in Figure 5.1, the SBIC is connected to the serial RP through the driver/receiver circuit. The main functions of the SBIC are [11]:

- Clock recovery (clock regeneration), i.e. to extract the clock from the incoming data. This is needed because the incoming data might have a slightly different bit rate than the normal 10 Mbit/s [12].
- Data encoding/decoding (NRZI-L / NRZ).
- Poll recognition, that detects if a poll frame has been received. In that case the RP is allowed to transmit on the bus.
- RP status recognition. There are two RP status types, normal and separated. The status is represented in a status bit in each frame that is compared with the status bit stored in an internal RP register.
- Individual, group and global address recognition. Compares the address of the incoming frame with the one stored in a local register in the RP.
- Enabling/disabling of RPB path A and path B.
- Frame preamble used for clock synchronization. Prior to a response frame(s) the SBIC adds four level shifts so that the receiver can recover the clock.
- RTS and CTS function (Ready To Send and Clear To Send). Signals used between the SBIC and the HDLC controller to synchronize data transfer.

5.1.3 The HDLC controller

Between the physical and network layers is the data link layer that is implemented in a single circuit from SGS-Thompson microelectronics. The circuit has a 10 Mbit/s full duplex HDLC channel. Of course two of the main tasks are to delimit frames with the 8 bit flag (01111110) and to make sure that the flag is unique, accomplished by bit stuffing.

5.1.4 Regional Processor Bus Serial Interface Circuit (RPBSIC)

This circuit is not a part of Figure 5.1 because it is a newer design of the SBIC. The big difference is that this circuit implements both the SBIC and the HDLC controller. Otherwise the main functions of the RPBSIC are identical with the functions described above.

5.2 RPH

In section 3.2.4 a brief description of the RPH is given. In that part it is described that the RPH consist of up to 4 RPHMs and that each RPHM can handle eight RPBH-S. To add some confusion the RPBH-S is implemented on a board that is called Serial Regional Processor Bus Interface (SRPBI). Each SRPBI can handle one RPB-S and it also has a cross connection to the other CP side, this can be seen in Figure 3.6 and Figure 5.2.

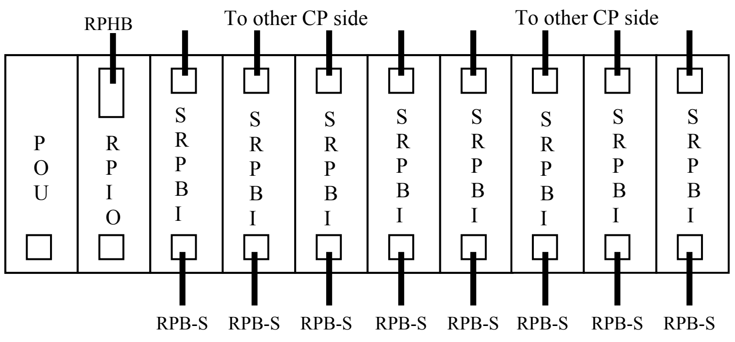


Figure 5.2 - RPH Magazine and connections

5.2.1 Power Unit (POU)

The POU is a part of the RPHM and it consists of a DC/DC converter which supplies the backplane (in the RPHM) with +5 V [7], see Figure 5.2.

5.2.2 Regional Processor I/O board (RPIO)

The RPIO board connects the RPHM backplane with the RPHMI (located in the CP) through the RPHB (see Figure 5.2). To connect to the RPHB, the RPIO board contains differential driver and receiver circuits [7].

5.2.3 Serial Regional Processor Bus Interface board (SRPBI)

The SRPBI is the board that physically connects the RPHM with RPB-S. The main functions of the SRPBI board is buffering of signals towards RPs, frame generation, polling of RPs and clock recovery. Furthermore it also handles buffering of signals from RPs and cross-connection of the RPB-S to both CP sides. The cross-connection can be seen in Figure 3.6. All of the functions above are implemented in a RAM based programmable logic device (PLD) and it is loaded from EPROM (Electrically Programmable Read Only Memory) at power-up [7].

5.3 A closer look at different hardware boards

The main part of the hardware investigation has been concentrated to find a PCI card or other device that could communicate with the RPB-S. Below, three different solutions for communicating with the RPB-S and one that communicate with RPHB are discussed. The reason for the RPHB solution is only as a backup if the investigation would show that it is not possible to use the solutions for RPB-S.

5.3.1 SIO PMC board

As a part of a project called SIO, at Ericsson Utvecklings AB (UAB), a PCI Mezzanine Card (PMC, also called Compact PCI) was developed which incorporated two RPBSIC. The PMC module is equipped with a compact PCI buss (this is actually what PMC means) that can be connected to an ordinary PCI buss in a PC or Sparc workstation through an adapter [14][13]. A simple overview of this solution can be viewed in Figure 5.3.

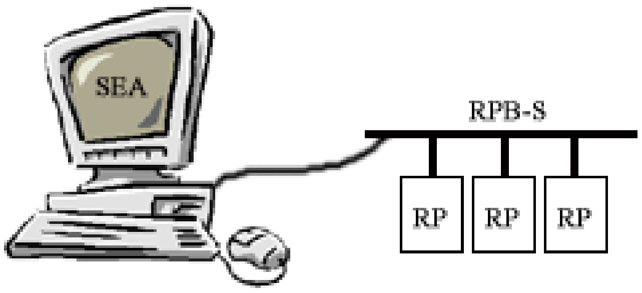


Figure 5.3 - RPB-S solution

After a thorough investigation of the card, and especially the RPBSIC, it was obvious that this card was not suitable. The reason is that the circuit has too many RP specific functions implemented. Among these there are especially one function which makes it impossible to use the card, the RPBSIC cannot send until it has received a poll frame (see poll recognition in section 5.1.2). This makes it impossible to send a poll frame from that card to other RPs, and therefore it cannot be used as an RPBH-S. The investigation has also shown that it is not possible to alter this function in any way, to resolve the problem.

5.3.2 DSP board

Another solution to the problem would be to use a Digital Signal Processor board that would be programmed to handle all the functionality for communicating on the RPB-S, i.e. NRZI-L coding, clock recovery, framing etc. This solution would be very much alike the one with the SIO PMC board, see Figure 5.3.

After the investigation of the driver/receiver circuit in part 3.1.1 it was found that the circuit was commercially available. This makes it possible to use a DSP board if it uses this circuit. Furthermore the board must fulfil some additional requirements, like being able to sample the incoming bit stream with a rate of 50 M samples per second. This is required to ensure that the clock is safely recovered.

After a thorough investigation of different DSP boards only one was found that used the proper LVDS driver/receiver circuits [15], manufactured by Innovative Integration. This board however could only sample the incoming bit stream with 28,5 M samples per second which is not enough for clock recovery. Another disadvantage of this board is that it is very expensive (112 000 SEK, including software development and debugging tools) [6]. In addition to the very high price for the hardware this solution would also require a lot of man hours for software development.

5.3.3 SERPENT RPB-S Emulator

This solution is slightly different from the two solutions above. It is a "box" that is connected to a SCSI card placed in a PC or Sparc Workstation. This makes it possible to connect the RPB-S to a portable computer if so is desired. An overview of this solution can be viewed in Figure 5.4. The device is developed and manufactured by a section called LMF at Ericsson in Finland.

Its main purpose is to act as a CP (SPU) from the RPs point of view and it is used mainly as a testing and maintenance tool for different AXE-10 devices [17].

LMF has also developed software for both Linux and DOS and the Linux version also has a graphical interface in which signals can be send and received. In their software package all source code are included which is good because it can be reused when developing the bridge between simulated and real environments.

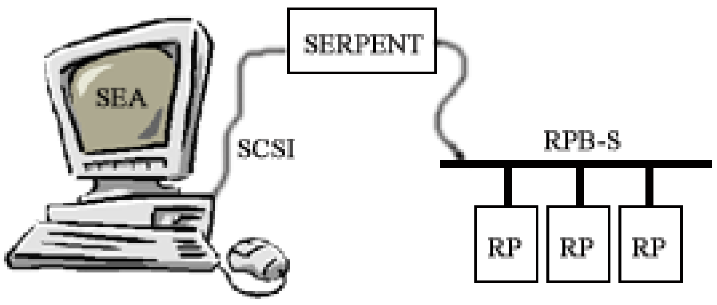


Figure 5.4 - SERPENT solution

5.3.4 RPHM Interface board

The Regional Processor Handler Magazine Interface (RPHMI) board is normally used to give an interface for the CP to the RPHB. Like the SIO PMC board this card also have a compact PCI bus and can therefore be plugged into a PC or workstation using an adapter [9].

An advantage with this board is that UAB has used it in their development and therefore has some software for it written for Solaris.

This solution requires more hardware because it cannot connect to the RPs directly, see Figure 5.5. It requires an RPH that handles the RPB-S and this RPH is then connected to the PC or Sparc workstation through the RPHMI board. This is the biggest disadvantage with this solution because it leads to higher costs.

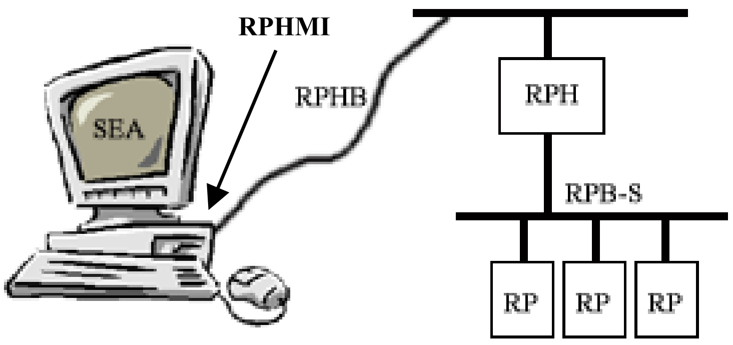


Figure 5.5 - RPHMI solution

5.4 Investigation results and Chapter review

This investigation has shown that the solutions with the RPHMI board and SERPENT RPB-S emulator are the only possible ones that do not require any hardware construction or modification (which is outside the scope of this project).

The outcome of this investigation is that the SERPENT solution is the one that fits best within the project goals. Advantages with the SERPENT solution, compared with the other three solutions are:

- It requires less hardware.
- It is developed for the purpose of being connected to a PC. The RPHMI board is not because it is intended to be a part of the new APZ 212 40.
- The RPHMI board is still a prototype, while SERPENT is a released product.
- The SERPENT can be connected to a portable computer with SCSI, which broadens the range of application.
- It includes a software package with source code written for Linux.

As mentioned in chapter two, the purpose with this project is to find out if it is possible to connect a RP to SEA. The purpose is also, if suitable hardware are found, to implement the RPB-S protocol stack and handle RP signals. Because SERPENT incorporates an RPH circuit (i.e. RPBH-S circuit) the communication with SERPENT must be with the RPHB protocol, over SCSI, instead of the RPB-S. This solution is rather an advantage than a drawback because the RPH circuit in SERPENT handles all polling of RPs.

6 System design

This chapter is a description of the system design. It gives a detailed description of how the system will be implemented.

6.1 Overall design

In the previous chapter it was decided that the SERPENT solution should be used. One of the reasons for this choice was that it had software written for it, which could be reused. As mentioned in chapter five, this software is written for the Linux platform, but SEA runs mainly on the Solaris platform at the moment. Due to time limitation and the high cost for a SCSI card for a Sparc Station, it was decided that the server should be implemented for the Linux platform. This makes it necessary to have some kind of link between the computer running SEA, and the computer that has the SERPENT connected to it. This link could easily be accomplished with a SEA component called the Message Protocol Handler (MPH). This solution also result in several advantages, like being able to connect any computer on the local network to the RP Server with the real RP hardware, and that the RP Server software, that is hardware dependent, never has to be rewritten if SEA is ported to another OS. A simplified overview of the total design can be seen in Figure 6.1.

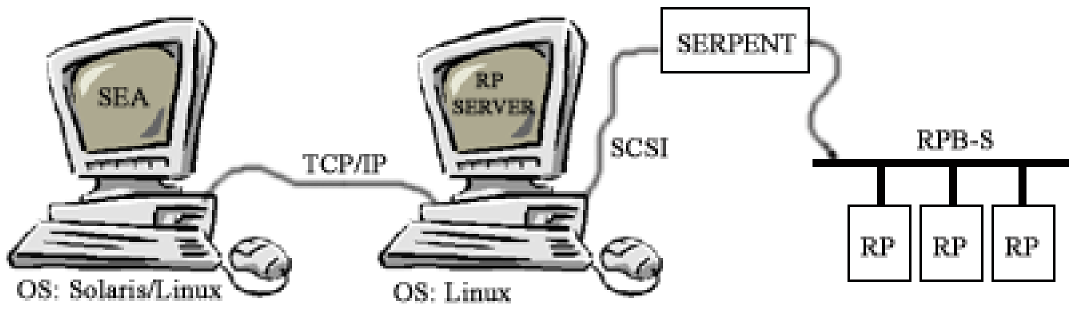


Figure 6.1 – Simplified overview of the system design

This solution will require two software components. The first component is the RP Server software. This software should handle the hardware initialisation and the connection/disconnection with the client. It also must handle polling of hardware and forwarding of signals, both from the client to the hardware and from hardware to client. The RP Server software could also include debugging tools, like logging and tracing of incoming and outgoing signals.

The second software component is the software running in SEA. This software should be implemented as a standard SEA component according to the COM model. The component should act as an RPBH-S proxy, i.e. handle forwarding of signals both from the CP in SEA to the RP Server, and from the RP Server to the CP. It should also handle buffering of signals, debugging tools like signal trace and the connection/disconnection to/from the RP Server. A complete overview of the system design can be seen in Figure 6.2

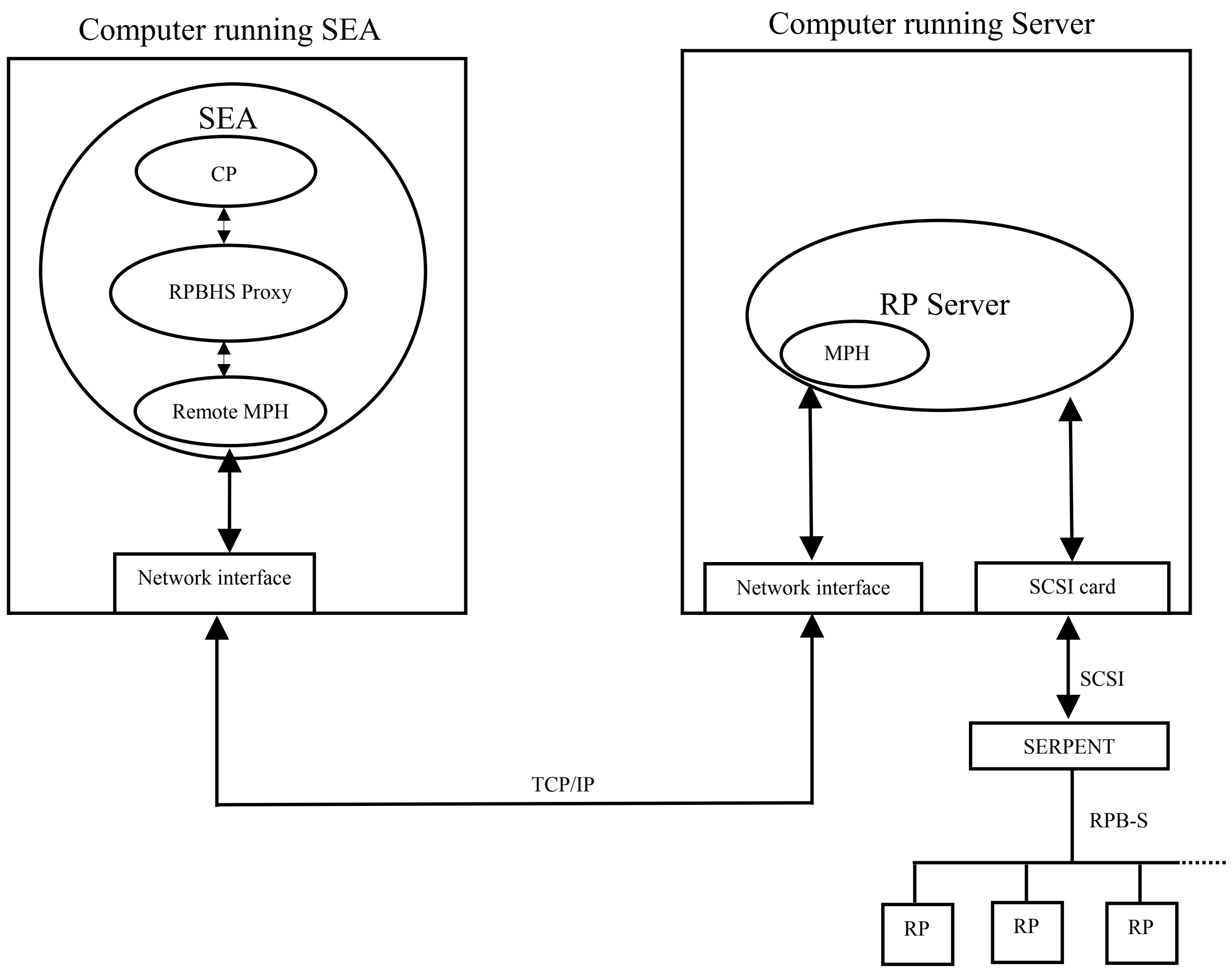


Figure 6.2 - Overview of the system design

6.2 Requirements

This section will specify the requirements for each of the two software packages (the RP Server and the RPBHS proxy). It will specify the different interfaces and functions that must be implemented to make the software work properly.

6.2.1 Requirements for the RPBHS proxy software

For the component to comply with the COM model it must first of all implement three basic functions. These are:

- **QueryInterface()**, which will tell other components which interfaces that this specific component supports.
- **AddRef()**, called every time a new reference to the component is created.
- **Release()**, called when a reference to the component is released. When all references are released the component could be removed from memory.

For the CP to be able to connect to the RPBHS proxy component, this component must also implement the IRphbClient interface (see chapter 4).

Furthermore the component must implement the *IMphRemoteResult* and *IMessageReception* interfaces. The *IMphRemoteResult* interface shall be used to handle the connection with the MPH server and the connection errors. The *IMessageReception* interface should handle incoming messages from RP Server.

Another very important interface that the RPBHS proxy must implement is the *ISimulationThread*. This is necessary because the component needs to be executed and scheduled by the scheduler in the SimCore (see chapter 4).

The two last interfaces that the component must implement are the *IAxeMgrConnection* and *ICommandEval*. The *IAxeMgrConnection* should among other things handle the connection

and disconnection of the component to other components in the system, e.g. the connection between the CP and the RPBHS proxy. The *ICommandEval* interface is used for handling of commands sent to the component through the Simple User Interface (SUI) in SEA.
A graphical user interface for the component should also be implemented, where the user can specify connection parameters.

Another requirement for the RPBHS proxy is that the Signals from the RP Server to the CP should be buffered. This is because the signals should not be sent to the CP until the component is scheduled. Signals from the CP to the RP Server do not need to be buffered, as they are taken care of in the CP.

6.2.2 Requirements for the RP Server software

Because the RP Server software is a stand-alone Linux application, it does not need to implement any interfaces, like the RPBHS proxy. However it must use the MPH functions in the MPH library to make it possible for the RPBHS proxy to connect to it. Therefore the RP Server must implement some callback functions used by the MPH library. These functions are used e.g. for connection, disconnection, error handling and reception of messages.

The RP Server software also must initialise the SCSI device and make sure that the device is not used by any other applications. The code for this functionality can be taken from the original SERPENT software with some modification. Also the RPH circuit in SERPENT must be initialised, and this code can also be taken from the original SERPENT software. However care must be taken about what part of the initialisation the RP Server should perform, and what part of the initialisation the CP does automatically when performing a global system reset of the AXE-10 switch.

The RP Server software should use the existing device driver for the SCSI card that came with the original SERPENT software. In this driver there also exist a TRANSMIT_DATA function that should be used. This function sends RPH signals towards the hardware.

There should also be a logging functionality implemented in the RP Server. This function should print the incoming and outgoing messages to screen or file for debugging purposes.

6.3 Design description

In the previous section there were a few requirements specified for the two components. Described below, more exactly, is how the two components will be implemented.

6.3.1 Design description for the RPBHS proxy software

As mentioned, the RPBHS proxy component implements a few interfaces. As with the three functions that are needed for the COM model, every interface might require that one or more functions are being implemented in the component that uses it. In section 6.3.1.2 to 6.3.1.9, there is a description of all interfaces that the RPBHS proxy component inherits and all the functions that needs to be implemented in the component.

6.3.1.1 Component structure

The RPBHS proxy component consists of three classes:

- **RpbhsProxy**, this class inherits all classes with interfaces described in section 6.3.1.2 to 6.3.1.8.
- **MsgQueue** is used to buffer all messages, delivered by the MPH component, received from the RP Server.
- **RpbhsProxyFactory** is used to create the RpbhsProxy instance. It inherits the IClassFactory class with the interface described in 6.3.1.9.

6.3.1.2 IRphbClient interface

This interface requires three functions to be implemented:

- **CpStateChange(CPSTATE)**, it is called by the CP component to notify the RPBHS proxy that it has changed its state.

- **SpuCloseDown(IUnknown *),** it is called by the CP when it is closed down .The RPBHS releases its pointer to the CP.
- **PendingCpRpSignal( )**, it is called by the CP to increase a counter named pendingCpRpSignal, by one, every time the CP has a signal to send towards a RP.

6.3.1.3 ISimulationThread interface

This interface requires only one function to be implemented, it is the **ThreadEntry (ULONG , LONG *)** function. This function is called by the scheduler in the simCore (discussed in chapter 4.1.2) every time the RPBHS component is allowed to execute. In this function the RPBHS proxy does two things. First it checks if the CP has any signals to send to the RP. This is done by checking the variable, *pendingCpRpSignal*, set by the function *PendingCpRpSignal()*. If it is greater than zero the RPBHS component fetches the signals one by one and sends them to the RP Server. Second, it checks if there are any messages in the queue from the RP that are to be sent up to the CP. If there are, it empties them and returns control to the scheduler. This function can be viewed in appendix E .

6.3.1.4 IAxeMgrConnection interface

As discussed in part 6.2.1, the *IAxeMgrConnection* interface is responsible for connection and disconnection of components in SEA. It requires that four functions are being implemented to work properly.

- **GetNoOfConnArgs(const struck GUID &, int *)**, this function is called by the AXE manager before it starts the component to see how many connection arguments the RPBHS Proxy components can receive.
- **GetConnArgs(const struct GUID &, long int, char **)**, returns the connection arguments that the RPBHS Proxy component can receive.
- **ConnectComponent(const struct GUID &, struct IUnknown *, long int, char **)**, is called by the AXE manager when the RPBHS proxy connects with the CP component. In this function the RPBHS proxy gets a pointer to the CP component. This pointer is used to call the *ConnectRpbh()* function in the CP. The *ConnectComponent* function also sets up a connection to the MPH component so that it can connect to the RP Server, and thereby send/receive messages to/from it.

- **DisconnectComponent(const struct GUID &, struct IUnknown *)**, this function is called by the AXE manager when the RPBHS proxy no longer needs to be connected to the CP component.

6.3.1.5 ICommandEval interface

This interface requires only one function to be implemented, it is the **CommandEval(char *, IUnknown *)** function. In part 6.2.1 it is mentioned how a user can send commands to the RPBHS proxy component. When a command is sent it is this function that is called. The command is delivered in the *cmd* and it can be used to change parameters in the component.

6.3.1.6 IMessageReception interface

This interface handles the connection to the MPH component. It is used to send and receive messages from/to the RP Server It is also used to trace signals, i.e. send the signals to a window that prints the signals on the screen. To accomplish this it needs three functions in the RPBHS proxy and they are:

- **NewChannel(char *, MPH_REFIID , MPH_PeerHandle peer, MPH_ClientData, MPH_ClientData *)**, this function is called by the MPH when a new channel has been created. A handle to the MPH channel that was created is passed on to this function in the *peer* argument.
- **ChannelClosed(MPH_PeerHandle, MPH_ClientData)**, this function is called by the MPH when the remote side has closed its connection.
- **HandleMessage(MPH_PeerHandle , MPH_ClientData , int length, unsigned char *message)**, when the MPH receives a message from the RP Server it (the MPH component) calls this function. The message is passed on to the Proxy component in the *message* variable and the message length in the *length* variable.

6.3.1.7 IMphRemoteResult interface

This interface is only used to connect to the MPH server in the RP Server, it is not used for sending or receiving any messages.

- **ConnectionClosed(unsigned int peerHandle, unsigned int clientData)**, is called by the MPH when the RP Server has closed its connection.
- **ConnectResult(unsigned int peerHandle, unsigned int clientData, MPH_RemoteResult resultCode)**, the variable peerHandle contains a socket and it is past on as an argument to the function *SendMessage* when sending messages to the RP Server.

6.3.1.8 Iconfig interface

To be able to configure the RPBHS Proxy component, so it connects to the RP server, it needs some way of letting the user alter the component settings. This is easily done with the Iconfig interface. It is mentioned in chapter 4.3 that SEA uses a configuration file when it sets up the simulation. In this file it is possible to add information that is received and evaluated by the Iconfig component and then sent to the correct component, in this case the RPBHS Proxy component. The configuration as of today for the Proxy component can be viewed below and the whole configuration file can be viewed in appendix D For an explanation of the configuration below, see table Table 6.1.

```
1. config RPBH_0 {
2.   connect-serpent -host viking -port 1331
3.   sigtrace on rp2cp
4.   sigtrace on cp2rp
5. }
```

Line no.	Meaning
1	Tells the IConfig component to send the information to the RPHB_0 component. This is in fact the RPBHS Proxy and in the beginning of appendix D it is shown how the RPBH_0 is created and that it is of type RPBHSPROXY.
2	Tells the Proxy to set the connection variables so that it can connect to the RP Server. The –host argument is followed by the hostname to the computer running the RP Server, in this case the computer is named *viking*. The last argument tells the Proxy which port the RP Server is listening on.
3	This command enables tracing of signal coming from the RP and passed on to the CP. In Figure 4.3 it is shown how the tracing of signals can be viewed in a window in SEA.
4	This command does the same thing as the one on line three with one exception. It enables logging of signals from the CP to the RP instead of vice versa.
5	End of configuration of the RPBH_0 component.

Table 6.1- Explanation of SEA configuration for the RPBHS Proxy

6.3.1.9 IClassFactory interface

The RpbhsProxyFactory class uses the IClassFactory interface when it creates an instance of the RpbhsProxy component when the proxy is created.

6.3.1.10 Internal functions in the RpbhsProxy class

These functions, which are described below, are internal functions in the RPBHS Proxy class.

- **ConnectToRemoteServer(),** is called when the Proxy component establishes a connection to the RP Server.
- **InitTcl(),** is called when the Proxy component is created. In this function it is possible to add more commands to the Tcl interpreter. This is done by adding the following code, where *ticInterp* is a pointer to the interpreter

  ```
  ticInterp->CreateCommand("sigtrace",
                           (TIC_CmdProc*)SigTraceCmd,
                           (TIC_ClientData)this,
                           (TIC_CmdDeleteProc*)0);
  ```

  If the command *sigtrace* is written into the Simple User Interface (SUI, mentioned in chapter 4.3) in SEA the Tcl interpreter will call the *SigTraceCmd* function.
- **SendRphbSignal(RPHBSIGNAL *),** is used to send RPHB signals to the CP when they are received from the RP Server.
- **HandleRphbSignalTrace(RPHBSIGNAL *, int),** this function is used to print the RPHB signal, in the SEA log window, in a way that is easy to view and understand.
- **HandleRpSignalTrace(int , CPRPSIGNAL *, int),** is used to print CPRP signals in the SEA log window. A CPRP signal is a signal that is used on the RPB-S, it is encapsulated in the RPHB signal.
- **HandleRpSignalTrace_MPH(int , CPRPSIGNAL *, int),** in SEA there is also a special window, not shown in chapter 4.3, that is used to look at signals from/to one or more RPs. This function sends a CPRPSIGNAL to that window via the MPH.
- **RPHBSIGNAL* ConvertCpRpSignalToRphbSignal(CPRPSIGNAL *, int),** this function adds the appropriate RPBH header to the CPRP signal.
- **CPRPSIGNAL* ConvertRphbSignalToCpRpSignal(RPHBSIGNAL *),** as mentioned above, the CPRP signal is encapsulated in the RPHB signal. This function is used to remove the RPHB header.

All functions below are Tcl functions and they are used to evaluate commands that are sent from SEA through the SUI. All these functions take the same arguments and they are**: (ClientData , Tcl_Interp *interp, int argc, char *argv[]).** The variable *interp* is a pointer to the Tcl interpreter that is used to evaluate the command. Depending on what command is sent the proper function is called. In variable *argc* the function gets the number of arguments passed to it in the *argv[]* variable.

- **friend int ListCmd,** is used to list all commands supported by the RPBHS Proxy component.
- **friend int SigTraceCmdchar,** is used to enable/disable the tracing of RPHB signals.
- **friend int RpSigTraceCmd,** this function enables/disables the tracing of CPRP signals.
- **friend int ConnectSerpent,** receives the connection parameters specified in the configuration file and then it calls the *ConnectRemote* function in the Proxy class.

6.3.1.11 Functions in the MSGQueue class

The *MSGQueue* class is used for buffering of signals received from the RP Server. The following functions are implemented in this class.

- **BOOL insertElement(RPHBSIGNAL *),** is used to insert a signal to the queue. It is important to delete the signal after insertion to avoid memory leakage.
- **RPHBSIGNAL *getElement(),** returns the first signal in the queue. It is not necessary to allocate any memory before calling this function. It simply removes the signal from the queue and returns a pointer to it.
- **int numberOfElements(),** this function is used to see how many signals that the queue containes.
- **BOOL killEmAll(),** is used to remove all signals from the queue.

6.3.2 Design description for the RP Server software

This section will in detail describe the functions needed in the RP server. The device driver functions that followes the SERPENT software will also be given a brief description. The server software is split into two files, RPBHSD.c and RPBHSD.h. The device driver is also split into two files, dd.c and dd.h. The server software is then compiled into two .o files, dd.o and RPHSD.o, and these files are then linked together to the final program. Here follows the description of the most important functions in the server and device driver parts.

6.3.2.1 Functions in the RP Server

- **int RPHInit(),** this function initialises the RPH circuit in the SERPENT hardware. It will reset some registers in the circuit, and set some other registers to normal start up values. The function will also check the RPH circuit version. It must be four or higher, otherwise the software will not support it. The return value will be zero on success or error code on failure.
- **void PollRPH(),** this function will send a groupsearch order (see Appendix B) to check if there are any messages available from any RP. If it is, the function will fetch the messages and forward them on the MPH connection.
- **int Init_scsi (unsigned char *device),** the Init_scsi function is used for initialising the SCSI device, and make sure no other program is using it. This is done through the use of a lock file. The device argument is the device that should be used, e.g. /dev/sga for the first generic SCSI device. The function will return zero on success and –1 on failure.
- **int DeleteLock(unsigned char *device),** if the lock file already exists, the program will try to delete it. This function is then used to delete the lock file. Returns zero on success, negative value on failure.
- **void Exit_ih(int sig),** the purpose for this function is to catch the system signals SIGINT, SIGTERM, SIGQUIT and SIGPIPE. If any of these signals are caught, the function will be called and clean up before the program terminates.
- **void openCallback(CONNECTION *conn, int ch, void *clientData)**, this is a callback function for the MPH. It will be called when a new MPH channel has been opened. The connection information that comes in the argument should then be saved for later use e.g. when sending messages.
- **void messageCallback(CONNECTION *conn, int ch, int length, unsigned char *message, void *clientData)**, called each time a message arrives from the MPH channel. This

function should then forward the message on RPB-S, and maybe also log the signal to file or screen depending on if logging is activated or not.

- **void closeCallback(CONNECTION *conn, int ch, void *clientData)**, called when the channel has been closed.
- **void newConnectionCallback(int fd)**, this function is called each time there is a connection on the server socket. However it does not mean that a communication channel has been set up. Messages cannot be sent until a channel between the server and the client has been set up.
- **void errorCallback(CONNECTION *conn, char *name, char *errorMessage, void *clientData)**, called when an error occur with the connection. This function must then clean up the mess and make sure that a connection can be re-established.
- **void printSignal(RPHBSIGNAL rphbSignal,int length)**, prints an RPBHS signal in a formatted way on the screen and to file. Used for logging of incoming and outgoing signals.
- **int setupMph()**, this function sets up the MPH connection. It will first create a listening server socket, and then register a listener in the MPH. This is needed to only allow a specific component to be able to connect to the server. It will also register the callback functions in the MPH. Returns –1 on error, zero otherwise.
- **int setupSerpent()**, will only call Init_Scsi and RPHInit functions described above and make sure everything went well. Returns –1 on error, zero otherwise.
- **int main(int argc,char **argv)**, main function in the program. Calls all the initialising functions and parses the program arguments. Thereafter the server waits for incoming connections.

6.3.2.2 Functions in the Device driver

This functions was included in the SERPENT software package and they are not altered in any way.

- **unsigned char *INIT_SCSI_DD(const unsigned char *scsi_device),** the purpose of this function is to initialise the SCSI device. Used by the Init_scsi function described above. Takes the SCSI device as argument and returns a string with a status message, e.g. “In use!” if the SCSI device is used by another application.

- **unsigned char *Inquiry (void),** this function is used to request vendor, brand and model from the SCSI device. If the SERPENT hardware is set up correctly this function should return:
  Vendor: LMF
  Brand: Serpent
  Model: 1
- **int TRANSMIT_DATA(unsigned short *data, char nu_bytes),** transmits an RPBH-S signal towards the hardware. This function is used to forward the signals from the MPH channel to the SERPENT hardware, so that they eventually reach the RPH or RP (depending on order type). Returns zero on success.
- **int RECEIVE_DATA(unsigned char *data, char *nu_bytes)**, this function retrieves data from the SERPENT. It is used to fetch signals from the hardware and forward them on the MPH channel. Returns Zero on success and the pointers in the argument points to the data buffer (char *data), and number of bytes in data buffer (char *nu_bytes).

6.4 Chapter review

- Due to time limitation and the high cost for at SCSI card for a Sparc Station, it was decided that the server should be implemented on the Linux platform.
- The MPH component in SEA will be used for the communication between the RP server and SEA
- Besides the RP server software, an RPBH-S proxy component for SEA was also designed.
- The RPBH-S proxy component should forward signals between the CP and the RP server. It also handles buffering of signals.
- The RP server handles forwarding of signals between the RPBH-S proxy and the serpent hardware.

7 Conclusions

The purpose of this project was to find out if it was possible to use a commercially available communication card or some other device for a PC or a Sparc workstation to implement a server for the Serial Regional Processor Bus (RPB-S).

Our conclusion is that there exists at least two ways of doing this. One is the SERPENT RPB-S emulator developed by Ericsson LMF in Finland, and this solution was considered as the most suitable for this application. The second possible solution is the RPHMI PMC but we regarded this alternative to be more complex than the SERPENT solution. It would also, most likely, been more expensive.

The purpose of our project was also, if suitable hardware was found, to implement the software required for communicating between a simulated CP and a real RP. This is not finalized but the design for the software is. We do not consider this to be a failure as the main goal was to find suitable hardware and this was accomplished.

7.1 Experiences

When writing this report we discovered that the pre study is very important and it took a large part of the total project time. Since the hardware investigation took such a long time we therefore did not have time to finish the implementation.

7.2 Problems

In our search for information about hardware, we experienced that it was hard to find documents and get in contact with the right persons. The main reason for this is that Ericsson is such a huge company and many hardware components were therefore designed in other countries. For example the SERPENT is designed and manufactured by Ericsson in Finland, but we heard about it, by chance, from a person at Ericsson in Spain.

7.3 Recommendations

We recommend that this solution should be fully implemented, because we belive it would broaden the range of application for SEA.

8 References

[1] Bengt Ossfeldt, RPB-S Conceptual description, 35/0062-15/FCP1050001 Uen, 1997

[2] Christer Gillen, Network layer protocol on the serial RP-bus, 2/155 19-APZ 211 11 Uen, 1999

[3] Christer Gillen, The physical and data link layers on the serial RP-bus, 1/155 19-APZ 211 11 Uen, 1998

[4] Elizabeth Mossberg, RP-Bus Serial Interface with connected terminals, UAB/B/X-97:146 Uen, 1997

[5] Ericsson Telecom AB, AXE-10 APT, EN/LZT 101 1274 R1A, 1992

[6] Innovative Integration, Product catalog 2000, http://www.innovative-dsp.com, 2000

[7] Jeff Triplet, Regional Processor Handler, 10262-CNZ 211 264, 1996

[8] Jeff Triplet, RPH bus for RPB-S in APZ 212 2X, 2/15519-CRZ 211 03 Uen, 1998

[9] Jeff Triplet, RPHM Inteface in APZ 212 40, 1/15941-3/FCP 101 0508/F Uen, 1999

[10] Kåre Särs, RPB-S Emulator, 19817-LPAH 101 102 Uen, 2000

[11] Lars Forsberg, Description of CAR 101 02 SBIC, 1551-CAR 101 02 Uen, 1997

[12] Laurie Duffy, Bit synchronization block in RPBHS, 13/102 62-CDA 101 01 Uen, 1997

[13] Leif Carlsson, Design specification for SIO PMC board, 102 62-ROY 119 2044/1 Uen, 1999

[14] Leif Carlsson, Design specification: SIO PMC GLUE CIRCUIT, 102 62-RON 109 625 Uen, 1999

[15] National semiconductors Corp., 3V LVDS Quad CMOS DLD, http://www.national.com, 1999

[16] Peter R. Torpman, SEA system description, 1551-CRL 119 007, 2000

[17] Sven-Ola Komstadius, 3V LVDS differential line driver, 1301-RYT 109 126/4C Uen, 1997

[18] Sven-Ola Komstadius, 3V LVDS quad differential line receiver, 1301-RYT 109 127/4C Uen, 1997

A Appendix. Abbreviations

APZ	Not an abbreviation, internal Ericsson code.
AXE	Not an abbreviation, internal Ericsson code.
CP	Central Processor
DSP	Digital Signal Processor
EIN	Ericsson Infotech
EMRP	Extension Module RP
EPROM	Electrically Programmable Read Only Memory
FCS	Frame Check Sequence
HDLC	High-level Data Link Control
IP	Internet Protocol
IPU	Instruction Processing Unit
ISDN	Integrated Services Digital Networks
LVDS	Low Voltage Differential Signalling
MML	Man Machine Language
MPH	Message Protocol Handler
NRM	Normal Response Mode
NRZI-L	Non Return to Zero Inverted (L means invert on zero)
PC	Personal Computer
PCI	Peripherical Component Interconnect
PLD	Programmable Logical Device
PLMN	Public Land Mobile Networks
POU	Power Unit
PSTN	Public Switched Telephone Networks
RAM	Random Access Memory
RP	Regional Processor
RP4	Serial version of RP (called RP in the report)
RPB-S	Serial RP Bus
RPBH	RP Bus Handler

RPBSIC	RP Bus Serial Interface Circuit
RPH	RP Handler
RPHB	RP Handler Bus
RPHM	RPH Magazine
RPHMI	RPHM Interface
RPIO	RP Input Output board
SBIC	Serial Bus Interface Circuit
SEA	Simulator Environment Architecture
SPU	Signal Processing Unit
SRPBI	Serial RP Bus Interface
SUI	Simple User Interface
TCP	Transmission Control Protocol
UAB	Ericsson Utvecklings AB

B Appendix. Signals and Orders

Signals from RPBH-S to the CP

Message Code (binary)	Message Header RPBH=0	From RPBH-S	
		Message name	Description
00 cccc	0000 to 3C00	Data	Signal from RP cccc=Code from RP
01 0001	4400	No_Answer	No answer received to single poll of one RP
01 0010	4800	Poll_Answer	Signal from RP after single poll
01 0100	5000	Register	Contents of one register in RPBH-S
01 0101	5400	Block Response	
01 0110	5800	Idle_Buffer	1 or 2 output buffers in RPBH-S have become idle
01 0111	5C00	Internal_Error	Errror in RPBH-S
01 1000	6000	External_Error	Error on RPB-S
01 1001	6400	Discarded_Signal	
01 1010	6800	RP_Ack	Signal to an RP have been acknowledged
01 1011	6C00	RP_Ack Retransmitt=1	
01 1100	7000	Deblock_Response	
01 111x			Reserved for internal use in SPU

Orders to RPBH-S from the CP					
Order Code (bin)	**Header (RPBH=0)**	**Functions for RPBH-S**	**L**	**Gr**	**Gl**
00 0tbs	0000 to 1C00	Send Signal s=Simulate SB/SE state b=Broadcast to all RPs t=Transparent	X		
00 100x	2000 or 2400	Send signal with resend x=Retransmission Ack.	X		
00 101x	2800 or 2C00	Echo Test x=Transparent	X		
00 110e	3000 or 3400	Set Bus side e=Executive	X		
00 1110	3800	Reserved	X		
01 0000	4000	Reset Link Protocol	X		
01 0001	4400	Start polling	X		
01 0010	4800	Stop polling	X		
01 0011	4C00	Stop fetching	X		
01 0100	5000	Single poll of one RP	X		
01 1001	6400	Clear outbuffers	X		
01 101s	6800 or 6C00	Block an RP s=simulate SB/SE state	X		
01 1100	7000	Deblock an RP	X		
01 1101	7400	Read register	X		
01 1110	7800	Write register	X		
01 1111	7C00	RPH test	X		
10 1100	B000	Reset RPBH		X	
10 1101	B400	Reload configuration		X	
10 1110	B800	Write register		X	
11 0000	C000	Reset Link Protocol			X
11 0001	C400	Start polling			X
11 0010	C800	Stop polling			X
11 0011	CC00	Stop fetching			X
11 1001	E400	Clear outbuffers			X
11 1100	F000	Reset RPBH			X
11 1101	F400	Reload configuration			X
11 1110	F800	Group search		X	
11 1111	FC00	Signal fetch	X		

C Appendix. SEA configuration file 1

```
# /proj/emudumps/21230/apz11.0/apz11.0/config_rp.axe
# ------------------------------------------------------------------------
# This is a configuration file for SEA.
# It describes which components that should be created and how they should
# be connected inbetween themselves to simulate an AXE-10
# To run SEA on this configuration file give the following command:
# sea -config /proj/emudumps/21230/apz11.0/apz11.0/config_rp.axe
# ------------------------------------------------------------------------
# This file has been automatically generated by SEA Configuration Wizard R2E
# Executed by einperd@ksba8e   Fri May  4 10:06:55 MET DST 2001
# ------------------------------------------------------------------------

# Creating the CP...
#=========================================================
create CP21230.CP21230.1            CP

# Creating RPBHs...
#=========================================================
create RPBH.RPBHS.1                 RPBH_0

# Creating RPs...
#=========================================================
create RPSIM.RPV2.1                 RP_1
create RPSIM.RPV2.1                 RP_4
create RPSIM.RPG2A.1                RP_7
create RPSIM.RP4S1A.1               RP_10
create RPSIM.RP4S1A.1               RP_19

# Creating RP software components...
#=========================================================
create RPCMSIM.RPFDR.1              RP_7_31
create RPCMSIM.RPMBHR.1             RP_10_28
create RPCMSIM.RPMMR.1              RP_10_29
create RPCMSIM.RPFDR.1              RP_10_31
create RPCMSIM.RPMBHR.1             RP_19_28
create RPCMSIM.RPMMR.1              RP_19_29
create RPCMSIM.RPFDR.1              RP_19_31

# Connecting RPBHs to the CP...
#=========================================================
connect RPBH_0             CP            IRphbServer    0

# Connecting RPs to their RPBHs...
#=========================================================
connect RP_1               RPBH_0        IRpbSServer    1
connect RP_4               RPBH_0        IRpbSServer    4
connect RP_7               RPBH_0        IRpbSServer    7
connect RP_10              RPBH_0        IRpbSServer    10
connect RP_19              RPBH_0        IRpbSServer    19
```

```
# Connecting RP software components to their RP...
#=====================================================
connect RP_7_31            RP_7         IRpCmServer   31
connect RP_10_28           RP_10        IRpCmServer   28
connect RP_10_29           RP_10        IRpCmServer   29
connect RP_10_31           RP_10        IRpCmServer   31
connect RP_19_28           RP_19        IRpCmServer   28
connect RP_19_29           RP_19        IRpCmServer   29
connect RP_19_31           RP_19        IRpCmServer   31

# Configuring the CP...
#=====================================================
config CP {
   set-processor-configuration -model 21230 -frequency 1
   set-memory-configuration -dsdevice 3 -dsboards 8 -prsrange 1
   load-dump /home/einperd/exjobb/apz11_rp.21230_emudump.gz
}
```

D Appendix. SEA configuration file 2

```
# /home/qinxfca/sea/rpbhsproxyconfig.axe
# -----------------------------------------------------------------------
# This is a configuration file for SEA.
# It describes which components that should be created and how they should
# be connected inbetween themselves to simulate an AXE-10
# To run SEA on this configuration file give the following command:
# sea -config /home/qinxfca/sea/config.axe
# -----------------------------------------------------------------------
# This file has been automatically generated by SEA Configuration Wizard R3A
# Executed by qinxfca@vanguard   Thu Apr 19 10:26:28 MET DST 2001
# -----------------------------------------------------------------------

# Creating the CP...
#=========================================================
create CP21230.CP21230.1                    CP

# Creating RPBHs...
#=========================================================
create  RPBHSPROXY.RPBHSPROXY.1    RPBH_0

# Creating RP software components...
#=========================================================
create RPCMSIM.RPFDR.1                      RP_7_31
create RPCMSIM.RPMBHR.1                     RP_10_28
create RPCMSIM.RPMMR.1                      RP_10_29
create RPCMSIM.RPFDR.1                      RP_10_31
create RPCMSIM.RPMBHR.1                     RP_19_28
create RPCMSIM.RPMMR.1                      RP_19_29
create RPCMSIM.RPFDR.1                      RP_19_31

# Connecting RPBHs to the CP...
#=========================================================
connect   RPBH_0                  CP          IRphbServer     0

# Configuring the RPH...
#=========================================================
config RPBH_0 {
   connect-serpent -host viking -port 1331
   sigtrace on rp2cp
   sigtrace on cp2rp
}

# Configuring the CP...
#=========================================================
config CP {
   set-processor-configuration -model 21230 -frequency 1
   set-memory-configuration -dsdevice 3 -dsboards 8 -prsrange 1
   load-dump /home/einperd/exjobb/apz11_rp.21230_emudump.gz
}
```

E Appendix. The ThreadEntry function

```
/* Logging and sending of signals is only allowed from/to RPs 7, 10 and 19 due to testing */

STDMETHODIMP RpbhsProxy::ThreadEntry (ULONG exeTime, LONG *resTime)
{
  CPRPSIGNAL *cprpSignal;
  RPHBSIGNAL *cp2rpSignal;
  RPHBSIGNAL *rp2cpSignal;
  int rpNumber = 0;
  HRESULT hr;
  int bitField;
  unsigned char *tmpString;

  (void) exeTime;              /* Avoid warnings */

  /* First check if there are any signal from the CP component */

  while ((pendingCpRpSignal > 0) && (waitForIdle == FALSE))
  {
    hr = cpPointer->FetchCpRpSignal(rpbhNumber,&cp2rpSignal);

    if(cp2rpSignal == NULL || FAILED(hr))
    {
      REP_TRACE(TRACE_AXE_MISC,("RPBHS_Proxy: Fetched a NULL pointer as a signal from
CP\n"));
        pendingCpRpSignal = 0;
        break;
    }

    rpNumber = GET_INTERNAL_RPNUMBER(cp2rpSignal);
    bitField = GET_ORDER_CODE(cp2rpSignal);
    if(rpNumber == 10)
    {
      if((bitField == XXBLOCK) || (bitField == XXDEBLOCK))
      {
        waitForIdle = TRUE;       /*This is done because when the CP has sent an XXBLOCK or
                                    XXDEBLOCK it is not allowed to send any signals until it has
                                   received a blockresponse or deblockresponse from the RP */
      }
    }

    /*Logging of signals*/
    if (cprpTrace && ((rpNumber == 7) | (rpNumber == 10) | (rpNumber == 19)))
      HandleRphbSignalTrace(cp2rpSignal, 1);


    cprpSignal = ConvertRphbSignalToCpRpSignal(cp2rpSignal);
    /*
      if(tracePeer) {
      HandleRpSignalTrace_MPH( rpNumber, cprpSignal, 1 );
      }
```

```
     if( (rpSignalTrace[rpNumber] || cprpTrace) && ((rpNumber == 7) | (rpNumber == 10) | (rpNumber
== 19))) {
     HandleRpSignalTrace( rpNumber, cprpSignal, 1 );
     }
    */
    MEM_free(cprpSignal);
    /*End logging of signals*/

    if(serverConnection) /*Check if there is a connection to the RP server*/
    {
      tmpString = (unsigned char*)cp2rpSignal->data;

      if((rpNumber == 7) | (rpNumber == 10) | (rpNumber == 19))
      {
        mphPointer->SendMessage(remotePeer, cp2rpSignal->length * 2, tmpString);
      }
    }

    /* Free allocated memory */
    MEM_free(cp2rpSignal);
    pendingCpRpSignal--;
  }


  /* Second check if any RP-CP signals is ready */

  while(inBuffer.numberOfElements() > 0) /*Check if there is any signals in the queue*/
  {
    rp2cpSignal = inBuffer.getElement(); /*Get first signal in queue*/

    /*Logging of signals*/
    cprpSignal =  ConvertRphbSignalToCpRpSignal(rp2cpSignal); /*Convert RPHB signal to CPRP
                                                                 for logging purpose*/
    rpNumber = GET_INTERNAL_RPNUMBER(rp2cpSignal);

/*
  if(tracePeer) {
  HandleRpSignalTrace_MPH( rpNumber, cprpSignal, 0 );
  }

  if(( rpSignalTrace[rpNumber] || rpcpTrace) &&((rpNumber == 7) | (rpNumber == 10) | (rpNumber ==
19))) {
  HandleRpSignalTrace( rpNumber, cprpSignal, 0 );
  }

  if (rpcpTrace) {
  HandleRphbSignalTrace(rp2cpSignal, 0); /*This function prints the tracing of signals in the SEA
                                           log window */
}
*/
    MEM_free(cprpSignal);
    /*End logging of signals*/

    if(rp2cpSignal != NULL)
    {
      if(GET_ANSWER_CODE(rp2cpSignal) == RphAdmin_IdleBuffer_Serial ||
        GET_ANSWER_CODE(rp2cpSignal) == RphAdmin_BlockResponse     ||
        GET_ANSWER_CODE(rp2cpSignal) == RphAdmin_RpbhState_DeblockResponse)
      {
       waitForIdle = FALSE;  /* If the Proxy has received an idlebuffer, blockresponse or deblock-
                                signal from the RP it is ok for the CP to send signals towards
                                the RPs again. */
      }

      SendRphbSignal(rp2cpSignal);  /* Send signal to CP */
    }
    else
    {
      REP_PRINTF(("RPBHS Proxy: Fetched a NULL pointer as signal from inBuffer\n"));
    }
  }

  *resTime = 0;
  return S_OK;
}
```