Master's Thesis
on

MIB design of AXE Regional Processors

by
Per Holmgren
2001-03-15
at the
Department of Microelectronics and Information Technology
for
Ericsson Utvecklings AB (UAB)

Examiner
Rassul Ayani

Faculty advisor
Vladimir Vlassov

Ericsson advisors
Richard Tham and Lennart Malmberg

Abstract

The motivation for this thesis is that Ericsson wants to start using open standards and platforms for controlling their AXE telephony exchanges out on the field. Up until now Ericsson has based their AXE communication on what they call signals, and not any standard protocol. The idea for this thesis is first to investigate what informations regarding operations and maintenance are transferred back and forth between a central processor (CP) and the exchanges, which are made out by a controlling type of regional processors (RPs) and other interfaces, other types of RPs, I/O boards etc., that are controlled by these RPs. From the CP-RP signalling informations most of the necessary data for operations and maintenance can be extracted and put into quite simple but numerous objects to model the data variables and tables in an exchange. These objects will then be put in a Management Information Base (MIB). This thesis also covers case studies and general SNMP simulations which is done by the use of a Simple Network Management Protocol (SNMP) manager and a simulated SNMP agent, which can be remotely managed by talking SNMP.

Table of Contents

1. Introduction

The motivation for this thesis is that Ericsson wants to start using open standards and platforms, in which the IP-family is a part and also widely spread. With the complexity that the future holds for the telecommunication business Ericsson believes they can not hold on to their own standards for all eternity. A big reason for Ericsson's success in the exchange area is that the AXE exchanges are built on modularity, which makes it relatively easy to add or remove functionality in them. In the development of new hardware and software Ericsson also ensures that it is back compatible.

An Ericsson AXE exchange system is simplifiedly divided into two major parts: a CP (Central Processor) and RPs (Regional Processors). There are many types of RPs and fewer types of CPs. An exchange out on the field is built with a subset of the total number of RPs that are available, depending on what type of exchange it is (wired telephone, gsm, umts etc.), and the exchange is managed by a parallel running pair of CPs, which logically can be considered as one single CP, through signaling. Those signals (thousands in total) have been developed by Ericsson over the years and now the signals are collected in a database where they each are described byte for byte.

Each RP has software which is also divided into two major parts: the APT part (telephony applications) and the APZ part (maintenance and operations of the exchange). For this thesis the APZ is the part of interest. The APZ part is divided into 16 blocks, i.e. programs, that can deal with things such as file management, communication supervision, error handling etc. The operating system of an RP is also considered as a part of the APZ, i.e. the operating system is a block in the APZ. The operating system supervises all other blocks in the RP, both APZ and APT blocks.

This thesis will mainly be focused on the signalling between the CP and RPs. However, sometimes sidesteps are made to look at the communication between different RP blocks and also sometimes then look at raw program code to determine which block is and should be responsible for the data representation. This can sometimes be hard to determine from the outside just by looking at the signalling between the CP and RPs, because there can sometimes be interactions between different blocks which are not visible only by looking on the direct signalling between a CP and RP block.

Since the AXE system is based on CPs and RPs which interact as managers and agents it is natural to look at the SNMP protocol, which is the most widely spread protocol for network management and also a part of the open TCP/IP suite standard, although SNMP usually uses UDP as the underlying transport protocol. An SNMP manager uses a model of the agents it is controlling by using a predefined conceptual, or virtual, database from which the SNMP manager can fetch and set necessary variables in the controlled agents. The virtual database is called a MIB (Management Information

Base), which is known to the manager and the agent whom the MIB concerns. The MIB models the data and variables that are available in the agent(s).

A MIB module is built like a hierarchical tree which eventually branches out into leaves. Each leaf make out one object where each object can consist of one simple variable or reference sequences of other objects to make out tables. In the objects there are usually definitions of access rights, current status and also a textual description of what the variable in the object is supposed to represent. A MIB at normally least consists of the standard MIB [1] which is necessary for the underlying layers of the IP stack for communication, statistics etc. It is then possible for for example enterprises or organizations to extend the MIB tree and add their own objects in a dedicated enterprise branch to model their own vendor specific devices.

This thesis work includes a study of the ASN.1, MIB and SNMP languages and protocols. The real work is to investigate what type of data in an AXE exchange should and could be managed and supervised by using SNMP, and then implement the data into a MIB structure (see appendix A). There are also some small case studies in section 6.5 to verify and show how the MIB data could be managed and supervised.

The report is divided into sections where sections 2-5 is an introduction to the concepts of networking and sections 6-7 describes target system and the work that has been done to adapt SNMP to AXE.

Section 2 describes the background and reasons of this project. In section 3 there is a short introduction to network management. A description of the formal language ASN.1 is given in section 4. Section 5 describes the concepts of MIBs and how they are designed by using ASN.1 notation. In section 5 is also a description of the SNMP protocol and how it can benefit from MIBs. The thesis work and the target system is described in section 6, and then the results and conclusions are to be found in section 7.

2. Background

There was an investigation team at Ericsson UAB in Älvsjö in 1998 that looked on in what direction to aim for in the future when it comes to dealing with the management of stand alone platforms[4]. That report suggested that the TCP/IP suite should be looked at, such as SNMP, Telnet, TFTP etc. for managing and loading software into the exchanges. This thesis was initiated as an extension of that report.

At Ericsson Utvecklings AB, Ericsson UAB, no work has been done before on the RP blocks which this report will focus on. Some work on modeling other RP blocks using MIBs has been done before though [2]. That work implemented a few MIB objects into a regional processor with a Group switch interface (RPG), mainly to see how hardware and software would respond, such as if there was good support in the operating system for this and trying out connectors etc. No SNMP testing on a real target was made.

The reader of this report should have a general understanding of the TCP/IP suite. The report will introduce networking and explain the ASN.1/MIB/SNMP structures, protocols and languages, but the reader should have some prior knowledge of the underlaying TCP/IP suite to be able to assimilate from this thesis. Also some programming skills and computer hardware knowledge can be useful to understand some underlying structures and algorithms.

As the main source of theory for the first big part of this thesis (sections 4-6) a book by William Stallings [1] has been used and some RFCs [7][8][9] have worked as complements to the book. For the second big part of this thesis the information has been gathered from several people at UAB, whom have been invaluable in regards of getting information about the current system.

Since this area is covered with abbreviations, in both the AXE and SNMP worlds, there is a list of abbreviations in section 11.

3. An introduction to network management

A network consists of hosts (workstations) connected to the physical network where each host is connected through a network card and then out onto the twisted pair or coaxial cables that make out the network. The choice of wiring standard depends on the required throughput and cost. With the increasing number of networks and more hosts connected to them the importance of network management is becoming greater and greater. In this context there are two types of hosts: managers and agents. A manager keeps track of its agents, i.e SNMP agents, via the SNMP protocol.

Network management is needed for keeping statistics and states about hosts in a network. An agent is a process running on a host that among other things can collect statistics about the data traffic to and from it. The statistics regard things such data traffic to and from the hosts all the way down to separate protocol specifics such as the number of good and bad packets using the IP, UDP and SNMP protocols respectively. The states of a host could be things such as if a specific application on the host is running or not, or any type of other data regarding a specific application that could be of interest to know or possible to set remotely.

The agent and manager know how they should store and access the data respectively through their MIB structure, which is implemented in both the agent and manager. The manager should have a MIB structure that covers all the MIBs of all its managed agents, while the agents themselves only need to keep a MIB structure of the relevant data for that particular host.

Networks can often be divided (see Fig. 3.1) into Local Area Networks (LANs) and Wide Area Networks (WANs). LANs are often networks managed by an organization or a company and the network is often located in a building or a group of buildings. A WAN is usually built from of a number of LANs that are interconnected

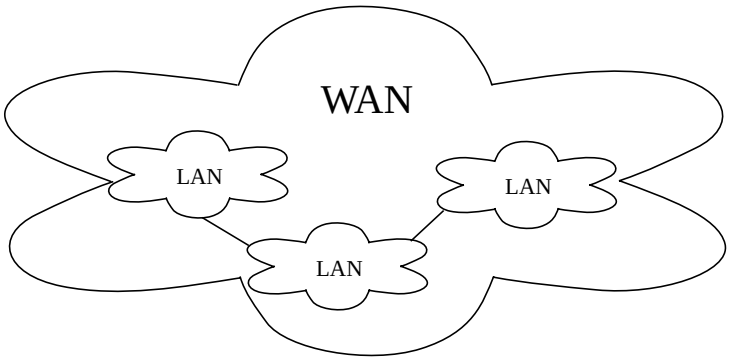

Figure 3.1 The relations between a LAN and a WAN

through bridges and routers, today usually routers. Of course the Internet is the biggest WAN in the world today.

A LAN must always be managed in some way. Typically a host on the network is appointed or dedicated to become the manager. The manager gets rights to get and set data on other hosts, e.g. agents, regarding networking statistics and communications etc. The standard today is that all hosts on a network, both managers and agents, support the Simple Network Management Protocol (SNMP) for network management. SNMP [1] is the protocol which for network management is the most widely spread and to which the tools of how to add manageable information are easily available.

A MIB [1][8] is a structure, i.e a tree, in which each leaf represent an object. Each MIB object represents a variable or a scalar. The MIB objects are defined in a formal language called ASN.1, which stands for Abstract Syntax Notation One ([1], in appendix). A MIB structure can be written from a subset of the whole ASN.1 language possibilities and it is those relevant parts of ASN.1 that are described in section 4. Then in section 5 there are rules of how a MIB should look like when using ASN.1, and that is called SMI, the Structure of Management Information [7]. This report only deals with the second version of the SMI, the SMIv2, which is the way of defining MIBs today. The second part of section 5 describes the second version of the SNMP protocol, SNMPv2 [1], which defines in which way the MIB data is transferred, e.g. requested and set, between managers and agents.

4. Abstract Syntax Notation One, the formal language

4.1 ASN.1 structure

ASN.1, Abstract Syntax Notation One [1], is a formal language that makes it possible to formally and uniformly define manageable application data. In a MIB only a subset of the ASN.1 language is allowed. An ASN.1 skeleton, which always has to be defined, is called a module. One module normally consists of information about one type of system and is collected in one file.

A MIB module [1][8] (see Fig. 4.1) always starts with the DEFINITIONS clause in which to name the module. For this project the modules's name is the RP-MIB module (see section 6.4 and appendix A). It is also optional to here give the module its place in a predefined structure, for instance in the enterprises branch of the MIB-II standard (see section 5.3).

```
<modulereference> DEFINITIONS ::=
BEGIN
   EXPORTS
   IMPORTS
   AssignmentList
END
```

Figure 4.1 The general definition of how an ASN.1 module is defined.

Inside the module there are the IMPORTS and EXPORTS statements to specify from which previously defined modules specific types are allowed for import respectively which types in the module are allowed for export to other modules. If no EXPORTS statement is defined it is assumed that the whole MIB module is allowed for export. The RP-MIB module in this thesis imports a few types from previously well known MIB structures (See appendix A). Finally the necessary objects that correspond to the application variables you want to model are defined in an `AssignmentList`.

For this project the module is the defined RP-MIB structure (see appendix A) which is added to the standard MIB-tree (see section 5.1).

4.2 Supported data types

The ASN.1 language supports data types of four different categories: simple, structured, tagged and other [1].

Simple: The most commonly used definition. The most common types under this category are the Integer, OctetString, BitString types. All other types are derived from the types of this simple type category.

Structured: ASN.1 totally contains four different structured types, SEQUENCE, SEQUENCE OF, SET and SET OF, but only the two first types are allowed to be used in a MIB structure, where they provide a possibility to build tables.

Tagged: Types derived from other defined types. Is useful when a user wants to give types associated names relevant for a specific device type or a project for instance.

Other: Two types are defined here, the ANY and CHOICE type. The any type is quite easy. Just define your variable as an ANY-type and go. This is useful when you don't know in advance the type of your variable. Of course you should re-define the variable as you go along and you find out what type of variable(s) it is. The CHOICE type is for the possibility to assign several possible types to one variable name. This is useful when the circumstances decide what type is assigned to a variable and the possible types are known in advance.

4.3 ASN.1 macro definitions

ASN.1 provides for macro notation where the macro notation is split up into three different levels [1]:

Macro notation: Used for defining macros. A macro notation is like a super macro which defines how macro definitions, the level two definitions, are possible to build.

Macro definition: The way in which related kinds of objects are defined. The definition specifies what types are mandatory and optional to an object type.

Macro instance: A manageable object derived from a macro definition.

There is also a sub level to the third level called the *macro instance value.* It can be considered as an extension of the macro instance. It is reached when the specific value is set to a macro instance.
A few macros will be described in more detail in section 5.4.

5. The Structure of Management Information

A Management Information Base (MIB) is a collection of objects. A single object can represent a scalar in an agent's instrumentation, e.g. an application variable such as an integer or a string, which is to be managed and supervised by a managing entity. Each MIB object also has its own position in the so-called MIB-tree, where each leaf in the tree represents one single scalar. The position of an object in the MIB-tree is decided by the object's OBJECT-IDENTIFIER, which is a sequence of integers separated by dots and can be considered as the object's MIB-tree address.

To be able to use SNMP on a host that participates in a TCP/IP network some parts of the MIB-tree should always be implemented on it. This implementation is called the standard MIB (see section 5.1). There are a couple of versions of additions to the standard MIB which are common. The most common addition now is called MIB-II (see section 5.2), which is really just an extension of the first version of the MIB, MIB-I. MIB-II just provides some more informations about statistics when using the standard internet protocols.

5.1 The standard MIB

The standard MIB is the skeleton of the ASN.1 tree [1]. The root of the tree is the object referring to the ASN.1 standard. The root has three sub-branches: `iso(1)`, `ccitt(2)` and `joint-iso-ccitt(3)`. The number inside the parenthesis is the actual branch number, but for readability each branch is usually also represented by a textual string. One of the sub-branches under the `iso` branch is the organizations (`org(3)`) branch, where the US Department of Defence (`dod(6)`) is one. Under the `dod` branch the IAB has allocated the `internet(1)` branch, which then has the address: `1.3.6.1` or in textual notation `iso.org.dod.internet`. Under the `internet` branch are four more sub-branches:

`directory(1)`: Reserved for future use with the OSI X.500, which is a collection of ways to gather information about individuals that participate in a system.

`mgmt(2)`: Objects approved by IAB. Those are the objects in the first MIB, MIB-I, and the second MIB, called MIB-II, which practically just added some objects to the MIB-I.

`experimental(3)`: Objects used during Internet experiments. This is the branch that has been used during this project. MIBs under development can use this branch and knowing that the OBJECT-IDENTIFIERS under this branch will not collide with

other objects. When the MIB is made public it is moved under a different branch, e.g. the `enterprises` branch.

`private(4)`: Objects defined unilaterally. The only public sub-branch is the `enterprises` branch.

Ericsson has the official `enterprises` sub-branch number `193` (see Fig 5.3) which can be looked up at IANA (ftp://ftp.isi.edu/in-notes/iana/assignments/enterprise-numbers). This project has applied for, and got, an `Ericsson` sub-branch of number 85.

Each MIB object has its own OBJECT-IDENTIFIER, but the value to a single object which has that OBJECT-IDENTIFIER has the addition `.0`. For example, the sysUpTime object in MIB-II has the OBJECT-IDENTIFIER `1.3.6.1.2.1.1.2`, but the value which it represents has the OBJECT-IDENTIFIER `1.3.6.1.2.1.1.2.0`.

Tables defined in MIBs are a bit special. In a MIB definition only the table header is defined. The table header contains the objects that represent each column of the table. When compiling the MIB to an agent you specify how many rows the table should contain.

5.2 What the MIB-II is

The MIB-II [9] is an extension of a MIB called MIB-I, which is an addition to the standard MIB (see fig 5.1). MIB-II is practically the same as MIB-I but with some added objects. MIB-II has sub-branches for monitoring a `system(1)`, `interfaces(2)` and things such as the `ip(4)`, `icmp(5)`, `tcp(6)`, `udp(7)` and `snmp(11)` protocol traffic on a host (see Fig. 5.2). Typical MIB objects are for monitoring of the traffic for each of the different protocols. These are informations regarding things such as counters for the number of incoming, outgoing and bad packets. States for different kinds of entities on a host, such as if they are up and running or not are also common. The SNMP branch of MIB-II for instance contains objects for the total number of incoming and outgoing SNMP packets, the number of incoming packets with the wrong SNMP version, the number of packets with the wrong community name and the number of incoming and outgoing GetRequests and SetRequests etc.

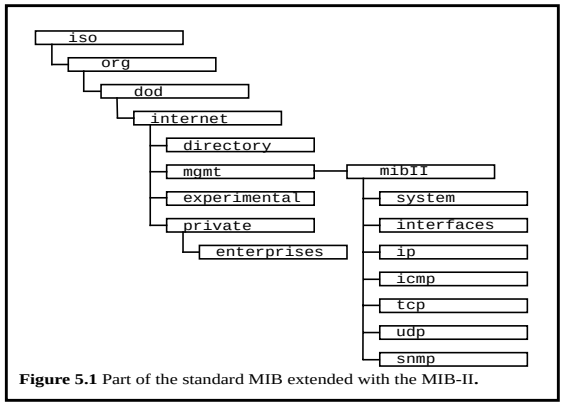

Figure 5.1 Part of the standard MIB extended with the MIB-II.

5.3 Making extensions to an existing MIB

A MIB-tree isn't just useful for network layered control but can also provide management and maintenance of vendor specific applications by adding new application or device specific objects to the tree, e.g. states of hardware and software such as temperatures or power supervision and test states or program versions respectively. The standard MIB has a branch that is called the `enterprises` branch. To this branch anyone can apply for a sub branch from IANA (see section 11). Each applicant then receives a sub address in the `enterprises` branch to which they can add their own developed MIB module with their application representation. Each SNMP agent with access to an application which is supposed to be monitored and supervised via SNMP should have the MIB that represents the application implemented/added in its MIB-tree.

Today there are about 8000 extensions to the enterprises branch and the official Ericsson branch has the `enterprises` sub-number `193` (see Fig 5.3). Then under `enterprises.193` there are about 85 sub addresses, where each address represents either an Ericsson project or a subsidiary to Ericsson.

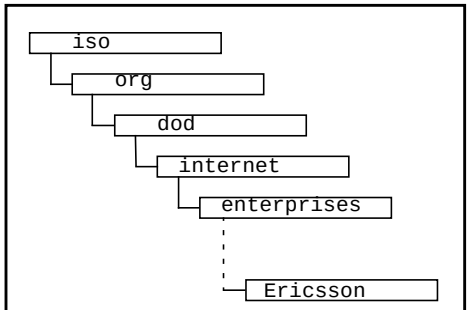

Figure 5.3 The Ericsson extension to the standard MIB.

The MIB-tree was developed to support simplicity and the possibility of extending the tree in an easy to do fashion. To reach these goals emphasis has been put on tight restrictions on implementation or else interoperability between vendor specific applications would eventually suffer. These rules are defined in the SMI [7].

5.4 Versions of the Structure of Management Information

There are two versions of the SMI [7] (Structure of Management information), SMIv1 and SMIv2 [1]. An entity using The SNMPv2 can often compile MIBs defined in both SMI versions, but the entities using SNMPv1 can normally only use SMIv1 defined types.

SMI and SNMP should not be mixed up. The SMI defines the rules in which MIB objects are defined, while the SNMP version decides how the actual information, the objects, should be requested, updated and collected between hosts. The versions described for SMI and SNMP in this report are both for version two, if not stated otherwise.

The SMI contains all the different types of objects,i.e. object macros, which are allowed to use in a MIB. The macros, see section 5.4.2 for an example, specify in detail what a type of object must and can contain and the macros are written in ASN.1. An SNMP entity to which a MIB is compiled must be aware of the SMI version that

the MIB is defined. There should not be mixes between objects written in SMIv1 and SMIv2 in the same MIB module.

5.4.1 The MIB objects

A ‘normal’ MIB object, i.e. defined as an OBJECT-TYPE (see section 5.4.2), is referred to a scalar or variable that can be monitored in an agent by a manager. It is this type of object that represent the leaf of a MIB structure. An ‘unnormal’ object is an object that is not by any means accessible from a manager. Such objects are only there to administrate a MIB module. A MIB module is normally the collection of objects that represent a hardware or maybe several related functions of a hardware.

Each MIB object is defined from the rules of the SMI (see section 5.4). A typical object consists of mandatory and optional clauses. Each object is initialized with a macro definition, where the macro defines what is mandatory and what is optional for that type of object. Examples of the most common macros are the MODULE-IDENTITY, OBJECT-TYPE, NOTIFICATION-TYPE, OBJECT-GROUP and NOTIFICATION-GROUP macros, where the OBJECT-TYPE macro is the most commonly implemented since this macro is the only one which describes a ‘real’ object, an application variable. The NOTIFICATION-TYPE macro is for defining which objects should be included in a SMIv2 notification, which is a kind of trap (see section 5.6.1.6) in SNMPv2. The other macros are inventions for administration of internal, e.g. grouping of objects that are related within a MIB module, and external MIB structures, i.e. the relations of the current MIB module to other MIB modules or structures.

```
OBJECT-TYPE MACRO ::=
BEGIN

TYPE NOTATION ::= “SYNTAX” Syntax
                  UnitsPart
                  “MAX-ACCESS” Access
                  “STATUS” Status
                  “DESCRIPTION” Text
                  ReferPart
                  IndexPart
                  DefValPArt
VALUE NOTATION ::= value (VALUE ObjectName)
Syntax ::= type(ObjectSyntax) | “BITS” “(“Kibbles”)”
Kibbles ::= Kibble | Kibbles | “,” Kibble
Kibble ::= identifier “(“ NonNegativeNumber “)”
UnitsPart ::= “UNITS” Text | empty
Access ::= “not-accesible”|“accesible-for-notify”|“read-only”|“read-write”|“read-create”
Status ::= “current” | “deprecated” | “obsolete”
ReferPart ::= “REFERENCE” Text | empty
IndexPart ::= “INDEX” “{“ Indextypes ”}” | AUGMENTS “{“ entry “}” | empty
IndexTypes ::= IndexType | IndexTypes “,” IndexType
IndexType ::= “IMPLIED” Index | Index
Index ::= value (indexobject ObjectName)
Entry ::= value (entryname Objectname)
DefValPart ::= “DEFVAL” “{“ value (Defval ObjctName) “}” | empty
Text ::= ””””” Text ””””

END
```

Figure 5.4 The OBJECT-TYPE macro that defines what is possible to include in an OBJECT-TYPE object. It also specifies what is mandatory and optional to include.

5.4.2 The OBJECT-TYPE macro

The OBJECT-TYPE macro [1] at least consists of a SYNTAX clause, a MAX-ACCESS clause, a STATUS clause, a DESCRIPTION clause and an OBJECT IDENTIFIER. These are referred to as the mandatory clauses of the OBJECT-TYPE macro. There are also some optional OBJECT-TYPE macro clauses allowed, such as the INDEX, the AUGMENTS, the DEFVAL, the UNITS and the REFERENCE clauses. The formal ASN.1 definition of the OBJECT-TYPE macro can be found in Figure 5.4.

5.4.2.1 Mandatory clauses

These are clauses that always must be present in an object defined with the OBJECT-TYPE macro (see Fig. 5.4).

SYNTAX: The way and sometimes the size or range in which the data is to be stored. The allowed data types are all derived from the types of the ASN.1 language. The ASN.1 types allowed to be used in MIBs are Integer, OctetString, null, object identifier and sequence (or sequence-of). The types in a MIB that are derived from the allowed ASN.1 types are INTEGER, Integer32, Unsigned32, OCTET STRING, OBJECT IDENTIFIER, BITS, IpAddress. Counter32, Counter64, Gauge32 and TimeTicks. For some of the types there is a an optional possibility to set the size or range of the data. Some of the types are only altered by name and others also by size or range.

MAX-ACCESS: The maximum allowed access rights which a manager has to an object located in an agent. The available rights are: *read-only, read-write, read-create, not-accessible* and *accessible-for-notify.*

read-only: A manager can only read the data in the managed object.

read-write: A manager can both read and set the data in the managed object.

read-create: A manager has read-write access to the managed object data plus the right to create new objects of that instance, if the object is part of a table in which the rows are creatable and deletable. Tables can also be static, i.e. the objects of the table are always there but not necessarily set.

not-accessible: A manager cannot access this object at all. Is used for the objects that make out the skeleton or description of a table.

accessible-for-notify: A manager has no rights to access the object itself. The object is only readable when it is spontaneously sent as a trap.

STATUS: The status in which the object is. There are three possible status states: current, deprecated and obsolete.

current: The object is in use and is supposed to be used for applications of today.

deprecated: The object is about to become obsolete but can still be used on systems supporting this object.

obsolete: The object is old and not in use any more. Such an object should not be removed and certainly not replaced with another object with the same OBJECT-IDENTIFIER since that would become a problem when dealing with past releases of the MIB.

DESCRIPTION: A textual description of what the object is supposed to do and how the data is represented and supposed to be read and understood.

OBJECT IDENTIFIER: The number that describes where in the MIB-tree the object is located. Either the whole identifier from the root all the way down or from a predefined place in the existing tree that is already implemented. As an example the `enterprises` branch is located at the OBJECT IDENTIFIER address `1.3.6.1.4.1`. The textual description of the same branch is `iso.org.dod.internet.private.enterprises`.

5.4.2.2 Optional clauses

The optional clauses makes it for example possible for developers to make tables, extend existing tables or give objects default values when the agent starts up.

INDEX: For making tables this clause is mandatory. An object containing this clause is the conceptual (or abstract) object which describes a row in a table. In SMIv2 such an object should also have its MAX-ACCESS clause set to not-accessible. The INDEX clause contains the name(s) of the object(s) which distinguish one row in a table from another row. Often one object is enough for distinction, but it is possible to point out as many as needed.
Example: A table where vehicles should be able to be pointed out (not by using the number plates). The first distinction could be cars, buses and trucks, which are the types of vehicles. This isn't enough for pointing out one specific vehicle. The cars could then be separately indexed and also the buses and trucks, by using numbers. The vehicle type and the index is then enough for pointing out one specific vehicle.

AUGMENTS: An object containing this clause is used as an object containing the INDEX clause, to describe a row in a table. This clause is rather used to extend already existing tables, where extending means adding columns. The AUGMENTS clause contains the object which described the row in the already defined table, i.e. the object

that contains the INDEX clause in the table that is to be extended. An object can not contain both the INDEX and AUGMENTS clause at the same time.

DEFVAL: The default value that an object should have when the agent it resides on starts up.

UNITS: A textual description of which unit applies to the data, for example bytes, seconds etc.

REFERENCE: A textual description of a reference to another MIB object in another module.

5.5 Overview of the Simple Network Management Protocol version 2

5.5.1 An SNMP retrospect

The SNMP protocol was specified in the late 1980s. It has now become a standard TCP/IP-suite protocol for network management since it is vendor independent, as all

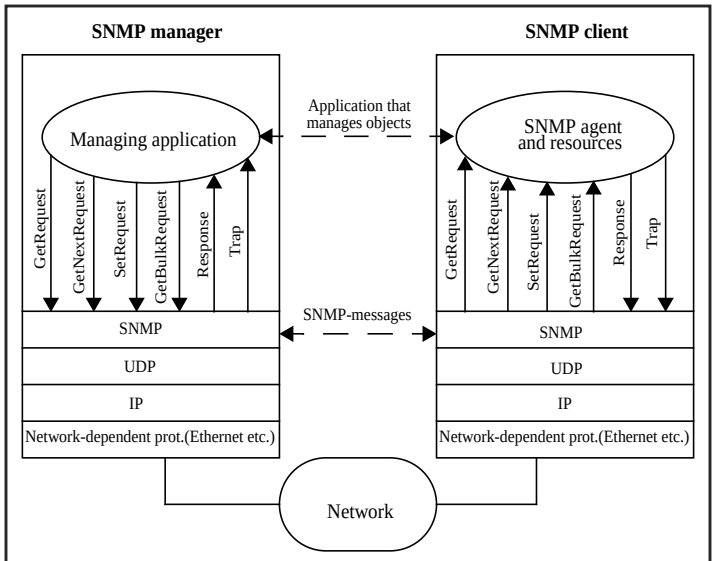

Figure 5.5 The layering and structure of normal SNMP exchanging between an SNMP agent and manager.

IP-protocols should be, and easy to use. SNMP doesn't necessarily need IP nor UDP as the underlying layers to function, but this is usually the case (see Fig. 5.5).

The first SNMP protocol is now referred to as SNMPv1 since there in 1993 was a new version called SNMPv2 issued. In 1998 SNMPv3 was issued but it has not become very widely spread, yet. A small comparison between SNMPv1 and SNMPv2 shows

that the second version is more efficient and that it has more functionality. The biggest reason for this is the introduction of the *GetBulkRequest* (see section 5.6.1.3). SNMPv3 was issued mostly because the other versions don't support any real security, although version 3 has the same functionality that version 2 provides [1]. SNMPv2 was supposed to have encryption built in, but agreement problems between the developers made them skip that part.

For this project SNMPv2 is the considered protocol. It has the functionality this target system could benefit from and also the step from SNMPv2 to SNMPv3 is not that great as from SNMPv1. SNMPv3 should be possible to use in the future with the possibilities that SNMPv2 provides today.

5.5.2 SNMP protocol functionality

The operations that are supported in SNMP [1] are inspection and changes of scalar variables, where the scalar variables are represented as objects in a MIB. SNMP is used to manage and monitor all kinds of equipment such as computers, routers, hubs, printers and toasters. For the equipment to be managed in such a way it needs a running SNMP agent on it together with an implemented and compiled MIB which describes what data in the applications that is available for reading and setting by a managing entity. Some data can also be considered of such importance that an alteration of it triggers the agent to transfer it spontaneously to a manager. This behavior is called that the agent sends a trap to the manager.

SNMP works with a limited set of messages that are transferred between a manager and the agent(s) which the manager has access to over a network. In some cases SNMP messages can also be transferred between managers.

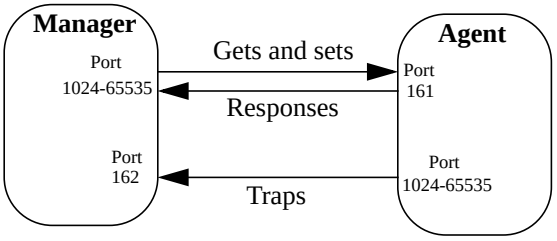

Figure 5.6 The relations of ports and SNMP messaging using UDP.

When sending SNMP messages over UDP the manager uses any available port [10] above 1023 (see Fig 5.6). An agent listens on port 161 for incoming UDP messages. Eventual replies from agent are sent to the same port as the incoming message came from. The agent does the same thing as the manager when initializing a spontaneous message, a trap, but instead sends that message towards the manager's port 162, which is also dedicated for UDP traffic. A trap is never responded to by anybody.

The messages within SNMPv2 are the *GetRequest*, *GetNextRequest*, *GetBulkRequest*, *SetRequest*, *InformRequest*, *Response* and *Trap* messages (see Fig 5.7 and section 5.6.1). All messages except *Responses* and *Traps* are issued by a managing entity. The

Response message is also issued by managing entity as an acknowledgment to an *InformRequest*, which is a message type passed between managers.

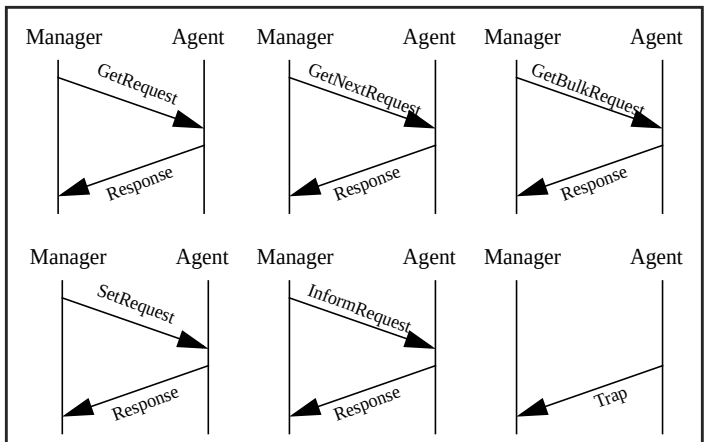

Figure 5.7 The different message relations in SNMP

5.6 Packet Structure

The version field of a SNMP packet tells which version of SNMP is used and the community field works as an insecure password of authentication. If no external encryption is used then SNMPv2 messages are sent as readable text (see Fig. 5.8).

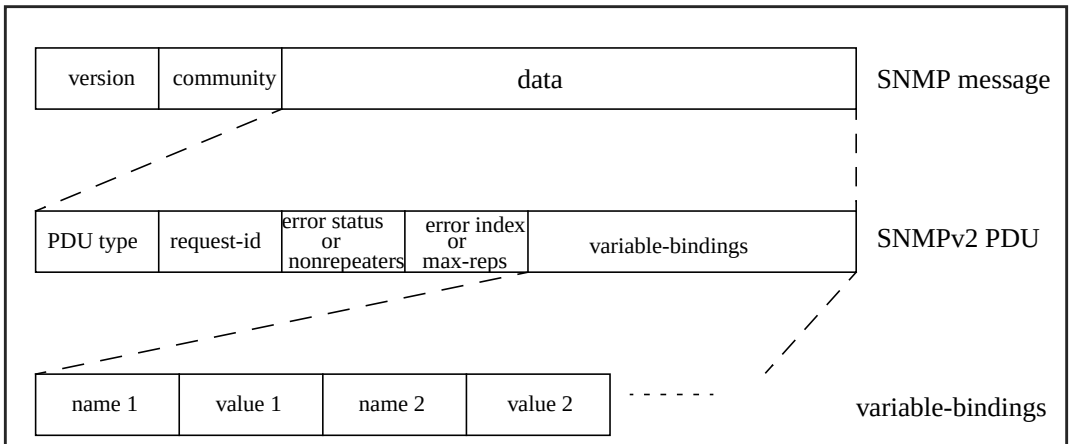

Figure 5.8 The general SNMP packet structure

The community field works as an authentication, where a manager and an agent interacting must support the same community string. Often readable data has one community and writable data another community, and both the manager and the agent must each set their community strings for reading and writing. In SNMP textbooks and other documents those communities are usually referred to as *public* and *private*, thus

system administrators often use those, and therefore their network management framework can not be considered as having very much security at all.
The data field of a packet is then the actual message, such as a *GetRequest* or a *SetRequest*.

5.6.1 Message descriptions

SNMP consists of a handful of messages or PDUs which are described in more detail below. All messages are practically constructed in the same way (see Fig. 5.8).

PDU-type: Indicates what type of message it is, for example if it is a *GetRequest, SetRequest* or a *GetNextRequest* etc.

Request-id: A unique identifier for each request, thus the manager can have several outgoing requests at the same time, also towards the same agent. The identifier can also be considered helpful when dealing with an unreliable transport service such as UDP. The issue that first comes to mind is duplication avoidance, but the real message check must be performed at the application layer.

Variable-bindings: The object instances that were requested. The variable-bindings list is divided into fields. Each field consists of a pair where the first part of a pair is a reference to an object name, and the second part of the pair contains the value for that object.

The PDU types that vary some from the majority are the *GetBulkRequest* and the *Response* messages. The majority of the messages have the *error status or non-repeaters* and e*rror index or max-reps* fields set to zero, 0, whereas the *GetBulkRequest* and the *Response* might have other values. For a description what those fields are and why see each of the message descriptions below.

5.6.1.1 GetRequest

A *GetRequest* is issued by a manager that wants to retrieve one or several values of objects from one of its managed agents. The manager specifies the OBJECT-IDENTIFIER of the object value he wants to retrieve as the first part of a pair in the variable bindings list. The second part of a pair in the variable-bindings list in any kind of request is always UnSpecified (NULL), since the manager doesn’t know any of the requested values yet.

If the requested object is missing or the corresponding value has not been set yet in the agent, then the *Response* will indicate that by a NoSuchObject or NoSuchInstance value instead. Other objects that may have been properly addressed in the same *GetResponse* will be correctly responded, though.

5.6.1.2 GetNextRequest

A *GetNextRequest* is issued by a manager who wants to retrieve the value of the next object in lexicographic order than the actual OBJECT-IDENTIFIER supplied in the *GetNextRequest*. The format is otherwise identical as for a *GetRequest*. The pair(s) in the response then contains the OBJECT-IDENTIFIER and the values of the objects that were in fact obtained.

If there is no lexicographic successor the identifier field has the same OBJECT-IDENTIFIER as in the request and the value field is set to endOfMibView. This only happens when the *GetNextRequest* is made on the very last object in the MIB module.

The reason for having *GetNextRequests* at all is for the manager to be able to use objects already known to him to find out if there are any more objects that the manager is not aware of yet, for instance to check if a table contains any more rows.

5.6.1.3 GetBulkRequest

A *GetBulkRequest* is issued by a manager wanting to obtain a large amount of information. The maximal number of variables to retrieve is syntactically defined to 2147483647, which is not realistic because the real limit to a packet is the limited packet size on the network, since a *Response* is never divided into several packets. The MTU is usually around 1500 bytes. A realistic amount of retrievable objects at a time is less than 100, of course depending on how big the individual objects are.

The *GetBulkRequest* makes use of the non-repeaters and max-repetitions fields of the message structure. The values in the fields can be set to zero or more and both values

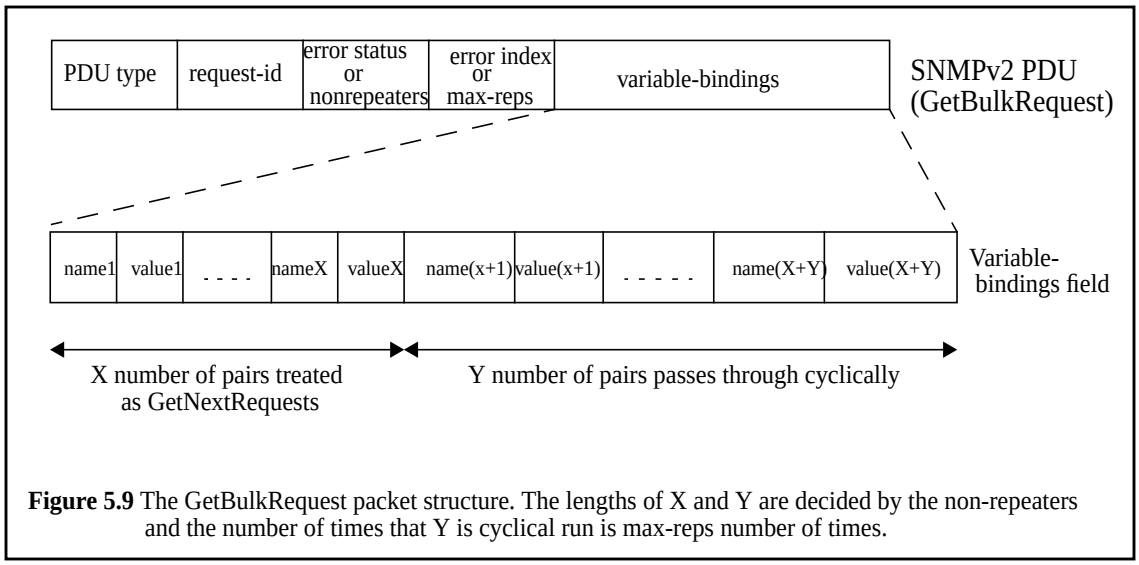

Figure 5.9 The GetBulkRequest packet structure. The lengths of X and Y are decided by the non-repeaters and the number of times that Y is cyclical run is max-reps number of times.

can be set at the same time. Each *GetBulkRequest* is responded by one *Response* message only. It is up to the manager to act if the responded information has been limited by the MTU.

The non-repeaters (X) field works as follows (see Fig 5.9): X indicates how many of the first of the pairs in the variable-bindings list the agent receiving the request should consider just as normal GetNextRequests. For each name of the requested pairs covered by X a GetNext action is taken by the agent.

The max-repetitions (Y) field works as follows (see Fig. 5.9): For the eventual last pair(s) (Z), that were not considered when using the non-repeaters field of the GetBulkRequest, each value for each pair one after another will be collected by the agent. When the last pair in the variable-bindings list of the request has been collected the agent will start with the pairs from the beginning (X+1) again, except that the first pair this time will be the lexicographic successor of the first pair of the past turn. The agent will do this for Y times and collect the pairs, names and values, in the normal *Response* message. This means that if the pairs in the request are the objects of headers in tables the response will consist of sequential rows of the table. Y will be the number of received rows, if the *Response* message doesn't become too big.
The number of pairs in a response to a GetBulkRequest will be Y + (Z*Y). The reason for the GetBulkRequest algorithm is that it should be possible to retrieve single object data and also table data in the same *Response*.
For any reason, except that the end of the MIB has been reached, the agent can not process a GetBulkRequest the *Response* will consist of an error status and an error index to indicate which object failed. In these cases either all requested objects are returned or none.

5.6.1.4 SetRequest

A *SetReques*t is issued by a managing entity that wants to set a new value to an object in a manageable agent. It is possible to set several objects of an agent in the same *SetRequest*, but if the agent fails to set any of the values then none of the values will be set at all. This behavior is referred to as an atomic operation, i.e. all or nothing. A response will then contain an error indication.

5.6.1.5 Response

A *Response* packet is sent to all the above messages. If an object name, i.e. OBJECT-IDENTIFIER, can not be found or there is something faulty with the object value, that is reported in the response. Example of fault types are *noError*, *tooBig*, *noAccess*, *wrongType*, *wrongLength* etc.
A *Response* message is only as big as the MTU of the network it is sent on, and not split up in pieces.

5.6.1.6 Traps

A *Trap* is issued spontaneously by an agent wanting to inform a managing entity of changes to data considered of such importance that a manger should be updated as soon as possible. In SNMP a *Trap* is never acknowledged, and since UDP is used as the underlying transportation protocol there is no way of knowing in higher layers if the message was received or not, even less if the message was OK or not.

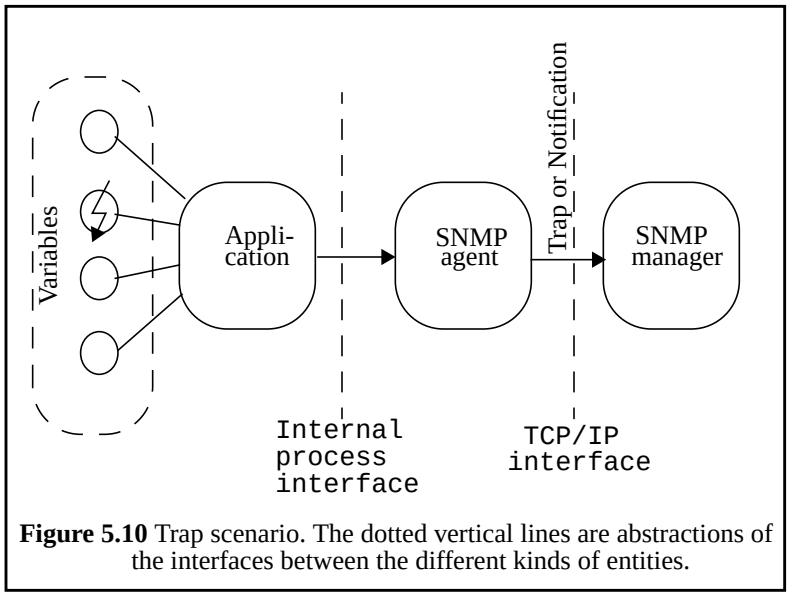

Figure 5.10 Trap scenario. The dotted vertical lines are abstractions of the interfaces between the different kinds of entities.

In SNMPv2 traps can be used, and should be used, in symbiosis with continues polling from the managing entity. If a change occurs the change is directly trapped to a manager for him to deal with it, and if the trap for some reason isn't received correctly the manager will be informed of the change eventually by polling anyway.

SNMPv2 in conjunction with SMIv2 makes it possible to define traps in two different ways. Either a defined MIB object is dedicated for being sent only as a trap, in this case the MAX-ACCESS clause is set to accessible-for-notify in SMIv2. This type of object is not accessible by a managing entity, only readable when an agent has sent it spontaneously. This is the way in which traps are issued in SNMPv1.

The other way of trapping, and only possible in SNMPv2, is by sending objects, which have their MAX-ACCESS clauses set to at least read-only. These objects are possible to send within a NOTIFICATION-TYPE object, which is another object macro definition. A notification trap is received by a manager with the following pairs of object names and values in the trap variable-bindings list: sysUpTime.0, snmpTrapOID.0 and the pairs of the objects' names and their values of the objects supposed to be included in the specified notification. When the manager receives a NOTIFICATION-TYPE

message he knows for how long the agent that trapped has been up and running, the OBJECT-IDENTIFIER of the trap sent and the object name and value pairs that were relevant for this trap. The benefits of this type of trap is that it makes it possible to also send other relevant information, stored in other objects, along with the intended trap data.

5.7 Weaknesses of SNMP

- SNMP was not developed for retrieving large volumes of data. In SNMPv2 this problem has been addressed by the introduction of the *GetBulkRequest* message. If the network is big the polling can result in a large volume of requests and responses. This could result in response times that are eventually unacceptable.
- The traps are not acknowledged. It is up to the application to keep track of those, and make continuous pollings of data.
- There is not very much security. SNMPv2 provides for no encryption and a poor way of authentication, only by the use of the community strings.
- There are limitations to how complex an object can be. They are not very sophisticated. This can of course also be seen as an advantage since the understandability doesn’t become too difficult.

6. The project objectives

The goal of this project was to define a MIB structure (see appendix A) for the software modules (blocks) on a Regional Processor (RP) that handle the O&M (see sections 6.1 and 6.3). After the scope of this project this new MIB structure, called RP-MIB, will be implemented in an SNMP agent software module which will be a process running on an RP accessing data of the other software modules, mentioned above, that are already running on the RP (see Fig. 9.1). The type of RP on which the SNMP agent will run is the SCB (Support and Connection Board).

To achieve the scope of the project the writer of this report has studied the signals transferred between RPs and the managing entity of today, the CP, to get a picture of the interface structure and what data is worth keeping in a MIB. Software has also been obtained to compile and run an SNMP manager and an SNMP simulation agent to verify that the RP-MIB is correct from an SMI (see section 5) and SNMP point of view (see sections 5.4 and 5.5).

6.1 An outline of the target system

The target system is an RP in an Ericsson exchange. An RP is a board contained in an exchange cabinet. The RP is a logical device that can monitor its surroundings and/or divide work, transfer calls etc., among other RPs or I/O boards.

An exchange is typically made up of several cabinets, which are divided into magazines. Each magazine can logically contain up to 32 slots. In each slot one board can be placed. A board can be an RP, such as an SCB board, an I/O board etc. An SCB is a controlling type of RP that supervises the other boards in the magazine. The magazines will then be interconnected via the SCBs through an RP bus, which is placed in the front of the cabinet (see Fig. 6.1).

The slots of a magazine are interconnected through the back plane. The APZ (see Abbreviations) part of the back plane is the so called I2C- bus or M-bus (see Fig.6.2).

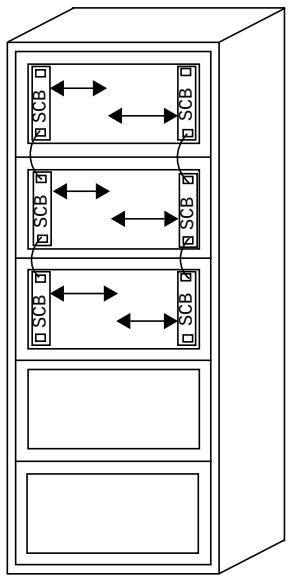

Figure 6.1 An AXE cabinet. Each SCB of an SCB pair is responsible for about half of the boards between them.

The RP that this thesis mainly is intended for is the so called SCB board, which currently is being designed. The RP-MIB module is also adaptable for use on other RPs that use any of the blocks (see section 6.3) studied in this thesis too.

An SCB is an RP type that supervises other RPs in a magazine. The SCBs are meant to be placed on both ends of a magazine to monitor and supervise other types of RPs and I/O boards in a magazine. The SCB can then report to a managing device, today a CP. The programs that will run on a SCB will be written in C.

An SCB is an RP type that has control of a magazine, at least when it comes to dealing with the O&M part of it. As said earlier one magazine typically contains two SCBs, where each SCB controls and collects data for one half of the magazine.

The tendency is to move more and more complex tasks from the CP onto the RPs, since their hardware is continuously updated in new versions and becoming faster and can have more funtionality. The main O&M task of a controlling RP today is mainly to divide work on behalf of the CP and collect and store data about other RPs and I/Os of a magazine.

The future perspective is to stop using a CP solution and instead run the exchanges in a stand alone kind of fashion. The logic will be kept in the RPs, and external devices will work as servers instead, such as storing the programs that will be loaded onto the RPs. However, a stand alone product also needs monitoring and supervision and that is what this project is a part of.

In traditional AXE the CP and the RPs which it controls are normally physically located within 10 meters of each other. The task of a CP is to handle important decision making and more complex problems. A CP also contains the programs that are loaded into the RPs when they start up. Normally two CPs are running in parallel and are doing exactly the same job, which is a result of keeping redundancy. If one of the CPs fails the other CP can take over exactly where the first one left off.

The CP and the RPs are connected to each other via a HDLC link which has approximately a <100 Mbit/sec capability. The back plane of a magazine has approximately a 10 Mbit/sec capability.

The aim for the future is to use some type of switched gigabit Ethernet protocol standard between the managing entity and the RP magazines and around 100 Mbit/sec in the back plane equivalence. The idea is then to interconnect the different parts of the exchange, both the APT and APZ parts.

6.2 Signal descriptions

Signals in AXE are divided into four categories: CP->RP, CP->CP, RP->CP and RP->RP (see Fig 6.2). The definition for CP->CP signalling is transfer of data between CP blocks on the same physical CP unit. The RP->RP signalling is mainly for distribution of data between or through different RP-blocks, where these blocks can reside on the same fiscal RP or another RP. The focus of this report is the signalling between the CP and RPs, the CP-RP and RP-CP signalling.

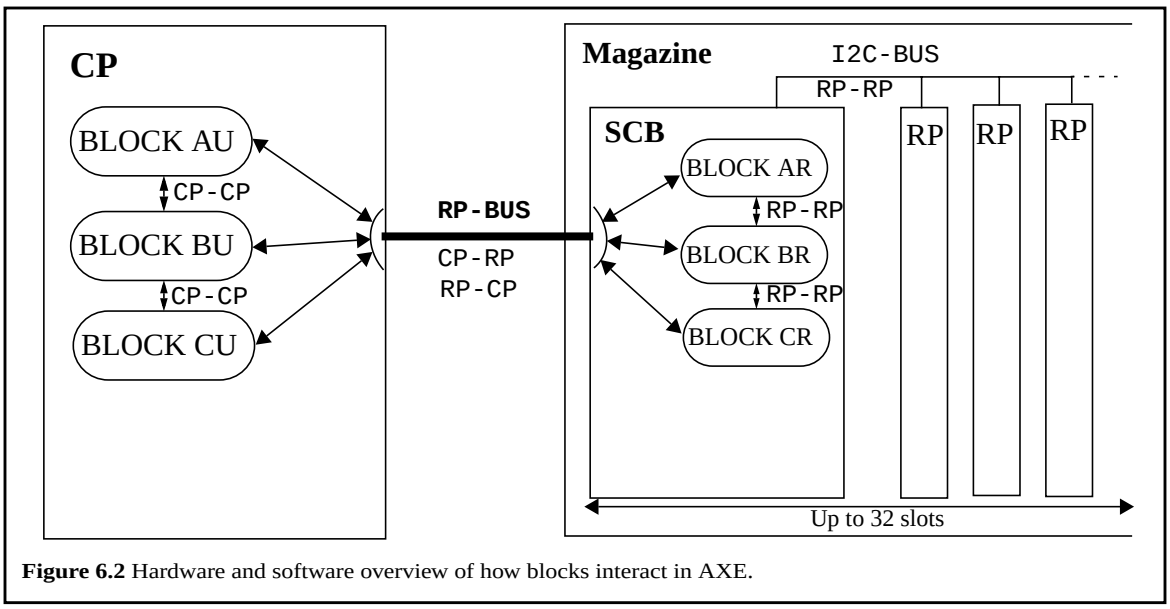

Figure 6.2 Hardware and software overview of how blocks interact in AXE.

Each AXE signal description can be found in a database at Ericsson, where they are stored according to a template. Each description tells who the possible sender and receiver is, what function the signal has, the possible return signals and the signal priority. A signal has a priority level between A and D where D is the lowest priority. The signals in APZ are normally at priority level C. Typically a CP->RP signal contains a reply signal pointer so that an eventual reply signal can be distinguished by the receiving CP. This scheme can from an O&M point of view be considered as a distributed client-server model, where the CP acts as client and the RP (SCB) as server.

Each data byte or word, a word is two bytes, of a signal is normally named in the signal description and then there is often a more detailed description of each name. For this project most of the CP-RP signals of the concerned blocks have been gone through to find out if the data is worth and possible to store in the RP-MIB and what data could be useful in an SNMP application. The criterias for finding data relevant to save in the RP-MIB have been a bit unclear but can be categorized in:

- Real stored data in the RPs, which is different kinds of PROM data such as HWI (HardWare Inventory see 6.3.1), processor serial numbers and clock rates. This type of data is normally read-only.
- Physical states of boards such as temperatures, power states etc. This data is normally read-only.
- Hardware devices that can be monitored and set such as different kinds of LEDs.
- States of different programs such as different test states, polling principles etc. This type of data can be both readable and writable from the managing side.

6.3 The considered RP-blocks

An RP-block is a software module that has defined funtionality and is run on an RP. There is also a corresponding CP-block running on the CP. The RP-blocks that have been of interest for this thesis are the blocks that handle APZ, Operations and Maintenance, of an exchange. The blocks in question will all be run on a new RP board, the SCB, for which the software currently is being designed. Other RPs, even though they might not utilize all blocks, can also benefit from this thesis.

In reality a block is divided into two parts, one part that runs on a CP and one corresponding part that runs on the RPs. The part running on a CP has a U extended to the block name and the part run on an RP has an R extended respectively. The names are then for example *RPMBHU* versus *RPMBHR*. The part of interest that this thesis regards are the R parts, if nothing else is stated.

The considered RP-blocks are named *RPMBH, RPMM, RPFD* and the *OS* (OS is not really an accepted name, but for general understanding OS will do).

A CP block part is normally programmed in the PLEX language and the RPs are normally programmed in C, which is the case for the SCB board.

The tasks of each of the considered RP-blocks are described below.

6.3.1 The RPMBH block

The main task of this block is to monitor the other boards in regards of HWI PROM and to set the states of the MIA (Manual Intervention Allowed) LED lights on each board.

The block is told by the CP how often the RP on which this block resides on should poll the slots for other RPs and boards for HWI data, how they are polled (not at all, only once or continuously), and what state the MIA-LED lights are in (see Fig. 6.3). The MIA-LED light states and HWI data of all boards in a magazine are stored in the

RAM memory of the SCB that the RPMBH block resides on. Logically the SCB can handle up to 32 boards (see Fig. 6.2), although a magazine normally holds around 20 boards.

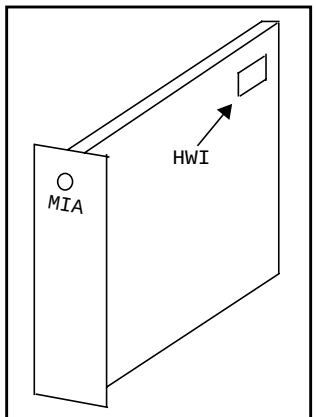

Figure 6.3 A general board architecture

MIA is a LED on each board that when lit indicates that it is OK to manually remove the board from its slot position. The HWI PROM contains information about the particular board, such as the board's serial number and date of manufacturing. One HWI PROM can contain of up to 255 bytes of data, but normally contains less than 100 bytes. The HWI data and MIA-LED states are collected and set by a controlling RP and then possible to distribute to the manager for reading and for the MIA case also setting.

6.3.2 The RPMM block

The block's task is mainly to monitor the power and fan status of the fans connected to the RP that this block resides on. One or several fans can be put in a cabinet to keep the temperature at an acceptable level.

The block keeps track of the power status. Normally the RPs are fed by two power branches (A and B). If one fails, permanently or temporarily, it is detected buy this block.

The supervised fan status concerns things such air temperature, motor current, LED states (on, off or blinking) and the status in which the SCB-Fan data communication is in, such as if there are framing or internal protocol errors. The fans are supervised by an SCB and the protocol used in that communication is the Denib protocol, which is an internal Ericsson protocol. An SCB has the possibility to monitor up to eight fan units at a time, but it is usually only one fan per controlling RP, though.

6.3.3 The RPFD block

This block deals mostly with testing of the RPs, not only SCBs, and the so-called EM-buses (Extension Modules buses). The block keeps track of things such as which test is running on the RP and if the test is in active or passive mode. This block is not only intended for use on the SCB type, but can be used on any RP type that needs the type of testing this block provides. The block is also responsible for testing Extension Modules bus (EM-bus). An EM is a board to which there is a logical bus, the EM-bus,

which an RP can have control over. Testing of the EM-bus is done by echoing data on it and looking if the returned data has been corrupted.

The possible RP-tests are:

- ***PS-test, Program Store test***. This is a checksum test where bytes are read from the PS area and computed into a checksum and compared with a previously known checksum. If the checksums are equal nothing happens otherwise the RP is halted.
- ***DS-test, Data Store test***: Data is written to the DS area and then read back to see if it matches the previously written data. This is done cautiously not to write over currently used and valid variables.
- ***CPU-test***: Testing of instructions, registers, the ALU, logical operations such as shifting etc., addressing and data memory bank testing.
- ***EM-test***: Extension Module test. The RPs that have EMs send echoing signals towards the EMs and make sure that the returned answer is the same. The SCB board does not have any EMs to control.

6.3.4 The OS block

From a maintenance point of view the OS takes care of two things.

- Specific board informations such as processor serial number, memory size, free memory size, processor clock rate, permanent and cached software etc.

- This block is in charge of maintaining the addressing of the board that this block resides on. A board in a cabinet is pointed out by two variables, the magazine address (subrack address) together with its slot number, which a board can obtain from reading its slot position from the magazine back plane. The boards are also appointed logical addresses which are stored by the RP together with their real addresses in an internal table. There is also a possibility to appoint a group address to an RP. This means that one address can be valid for several boards. This type of addressing is normally not used but the possibility exists.

6.4 RP-MIB specifics

In the RP-MIB module (See Appendix A) the data has been collected block wise, four blocks in total, where each block also can have sub functions (see Fig. 6.4). The reason to divide it this way is that some RPs use all blocks and some only a few. This way makes the RP-MIB relatively easy to adapt for the intended RP type, even though it is mainly intended for a controlling type of RP, such as the SCB.

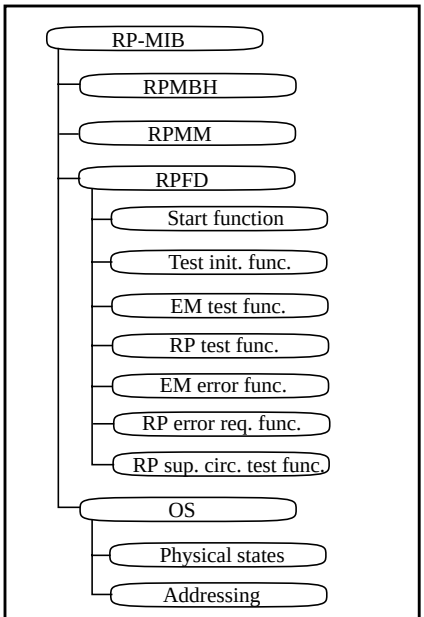

Figure 6.4 The RP-MIB module with the sub-blocks and their eventual sub functions.

Related data or data which is of the same type from several boards is normally collected in tables. The tables are normally indexed by an index number, such as an RP number or a fan unit number. Global data, data relevant for all boards, or single data, only relevant for the RP on which the SNMP-agent resides, is normally stored as single objects in the RP-MIB.

The memory usage that the actual data of this RP-MIB representation will use if fully implemented is in worst case about 5 KB. This estimation was reached by maximizing the table lengths and multiplying them with the maximum data lengths of each row and then also add all the single objects of the RP-MIB. The MIB compiled in the agent simulator was about 50KB with all tables lengths set to 32 rows, but this compilation also then contains the whole MIB tree structure.

The RP-MIB consists of a total of four tables and about 60-70 single MIB objects which from examining other MIB modules for other devices, for instance at Cisco Systems who have made their MIB modules public, seems like a reasonable amount of data.

6.4.1 RPMBH in the RP-MIB

The MIB objects of this block are single objects, relevant for all boards in a magazine, and board specific informations which are for this particular block stored in a table, the *RPBoardTable*.

The type of polling and the polling interval in number of 10 second intervals that this block does towards the HWI data is also saved in the agent as two separate objects.

This makes it easy to know for the agent and managing entity how often it is relevant to check the HWI data from this block, if so desired.

RPBoardTable (see Table 6.1): Consists of data for boards in all physical slot positions. The index of the table indicates which slot number it is (up to 32 slots). The relevant slot data to put in the MIB is the MIA LED state, HWI PROM data and details of possible changes in a slot. The changes can be that there is no board, that there is a new board a fault such as a missing EndOfText, an invalid checksum to the read HWI data or some other type of fault.

Slot number	MIA-LED state	HWI-data	Slot change
0			

Table 6.1

To prevent extensive SNMP polling of all slots in the table there is also a single MIB object linked to this information. This data is four bytes long (32 bits) where each bit represents one slot and tells if there has been any changes to a slot. If there has been any changes an SNMP manager then can read the more detailed information for that particular slot from the table and take appropriate actions.

6.4.2 RPMM in the RP-MIB

This block consists of power status information of the RPs in the magazine and of fan unit information.

A fan is not an RP, but is instead controlled by an SCB that exchanges information with the fan using an internal Ericsson protocol called Denib. The fan protocol communication is supervised from the RP side by looking for framing errors, checksum errors and other protocol errors.

FanDataTable in RPMM (see Table 6.2): Fan units are monitored an SCB. This table is indexed by the fan unit number, up to eight of them. The relevant data for a fan is air temperature, MIA LED state, HWI PROM data, whether the fan is accessible at all or not, if the HWI PROM data is accessible or not, and information about other physical things such as if the power feed, fan motor, communication etc. are OK.

Fan unit	Read result	Comm ErrorDet	Alarm info.	Air temp.	MIA-LED	DataStatus	HWIstatus	HWIReadRes	HWI data
1									

Table 6.2

For more detailed information see appendix A.

6.4.3 RPFD in the RP-MIB

RPFD handles tests of both the RP as well as the EM (Extension Module)-bus, if any. The RP-MIB contains two tables for this block, the *RPTestTable* and *RPEMTestTable*.

RPTestTable (see Table 6.3): There are four different tests that are cyclically run on each RP (see 6.3.3). For each possible test there is a corresponding bit that tells if the test is set active or passive. Trying to start a test with the test flag set passive will fail.

Type of test	Test state (act../pass..)
PS-test	⋮
DS-test	⋮
CPU-test	⋮
EM-test	⋮

Table 6.3

RPEMTestTable (see Table 6.4): The table is indexed by the EM numbers, up to 16 or 64 of them, depending on which type of RP it is. This table is represented in the RPs that have EMs to control. The SCBs do not normally have any EMs, although it can be in control of the RPs that have. The relevant test information for each EM address is which test state the EM is in, which CM controls the EM and in which order the tests are done. A CM is a software slot in the RP in which a block can reside in. There are 32 CM numbers (0-31). The OS always has CM number 16 and APZ related blocks have numbers 17-31. The remaining CMs, 0-15, are used by APT blocks.

EM address	EM test state	CM number	Em test order
0 ⋮	⋮	⋮	⋮

Table 6.4

6.4.4 OS in the RP-MIB

The objects in this block are stable physical information about the RP and objects for addressing the RP. The physical information regards things such as the product identity of the bootprogram, the size of the memory the bootprogram resides in, RP type number, board number, production name, processor serial number, processor clock rate etc.

There are also seven single objects for addressing the RP that this block resides on.

- **Subrack address**: An address read from the magazine that this board is located in.
- **Board address**: A slot address read from the back plane.

- **Stable physical address**: The conjugated magazine address and board address.
- **Logical individual address hardware**: The logic address that other interfaces use to address this board, this address lies in the back plane communication interface.
- **Logical individual address memory**: Same as above but this address is instead saved in the RP's memory.
- **Logical group address hardware**: The logic group address which lies in the back plane communication interface.
- **Logical group address memory**: Same as above but is instead saved in the RP's memory.

The reason for keeping two address pairs where each pair show the same address is for the RP to find out that an internal address fault has occurred. If the address pair is not the same then there is a fault somewhere, and logical conclusions can hopefully be made to find out where the real problem is.

6.5 Case studies for project verification

Some case studies have been made to verify that the project has been done according to the thesis specification and that the job has been correctly done. The cases have been prepared by the writer of this report together with the people who know what the cases should result in. The cases have then been implemented by the writer and approved by the people that outlined them. The cases are illustrated by sequence and flow diagrams. Three cases have been looked at. The first deals with what happens when a board is removed or replaced, i.e the HWI data changes. The second case looks at what happens when a LED is switched on by a managing entity and the third case shows what should happen when an unexpected address update is detected by the RP.

In the cases that include a trap being sent by the agent to the manager the traps are thought as being lost, i.e not received by a manager. The reason for this is that traps sent by UDP not can be relied on. In the RP-MIB there are always backup objects to the traps objects. The backup objects are normally quite small and can be continuously polled by a manager to find out if anything has happened.

See **section 9** for explanation of the OMF, which is referred to in the cases.

6.5.1 Case 1, HWI update

The first case studied uses information in the RPMBH block. The sequence diagram (see Fig. 6.5) shows how HWI data is transferred through an RP, from the application block out to the SNMP agent and eventually to an SNMP manager. The flow diagram (see appendix B) gives an explanation on how the RPMBH block works internally and uses the information that can be found in the RPMBH part of the RP-MIB.

To prevent extensive polling of whole tables there is a ‘master’ MIB object, rPGlobal-Change, containing a four byte string that is updated by the agent that this block resides on. rPGlobalChange is supposed to be polled continuously. It indicates if anything has happened in a slot. If something has happened that is indicated in the rPGlobalChange object, by a bit being set to 1, then polling has to be made of all objects linked to it (See appendix A which objects rPGlobalChange represents).

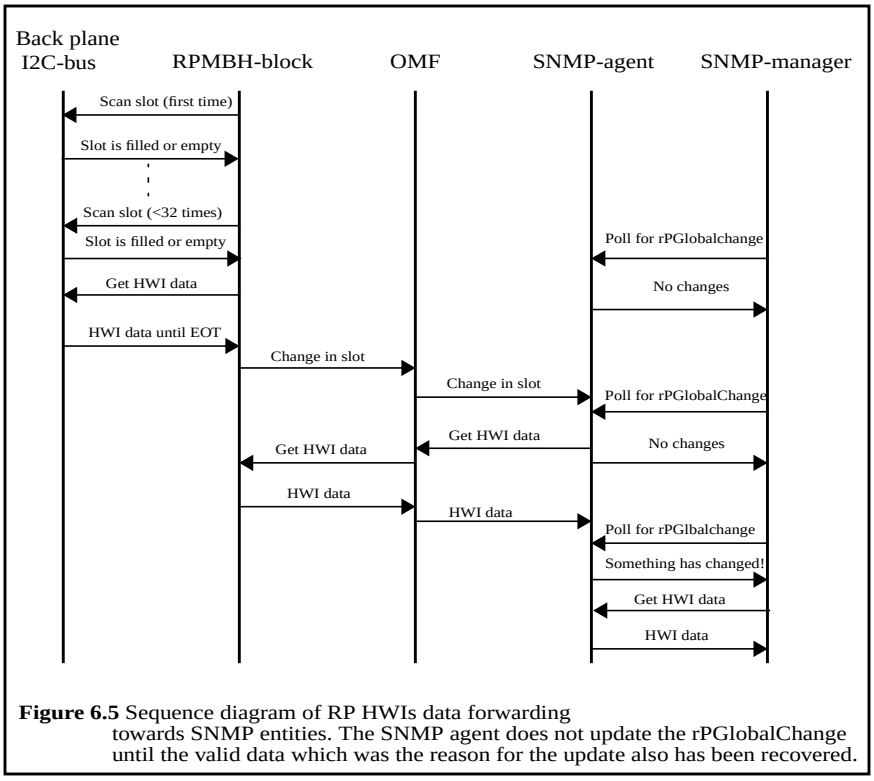

Figure 6.5 Sequence diagram of RP HWIs data forwarding towards SNMP entities. The SNMP agent does not update the rPGlobalChange until the valid data which was the reason for the update also has been recovered.

Figure 6.5 shows what happens when a new board has been put in a magazine and the RPMBH block on the SCB is made aware of it.

An SCB continuously polls for slot changes in the magazine (see appendix B). When the SCB finds a new board the SCB polls it for the HWI data. When this is done it reports towards a managing entity, e.g. an SNMP agent, that there has been a change. The SNMP agent then polls the RBMBH for, in this case, the HWI data. The SNMP manager then has to collect all objects that are relevant for rPGlobalChange to be updated, and one of those things is the HWI data. It is then up to the manager to take appropriate actions.

6.5.2 Case 2, Setting of fan MIALED

The second case studied is what happens when wanting to change the state of the MIA LED of a fan unit, which is handled by the RPMM block. This process is illustrated by a sequence diagram (see Fig. 6.6). An SNMP manager wants to set the LED of a fan unit. He then set the object, fanMIALEDState, for that particular fan unit. The order is transferred towards the SCB board, which responds by setting another objects, fanMIALEDSetRes. This does not guarantee that the LED will be set, just that the SCB will make its best effort to change it, since there is no acknowledgment from the fan unit back to the SCB. Since there is no guarantee that the LED will be changed the final trap can only inform the manager of the attempt.

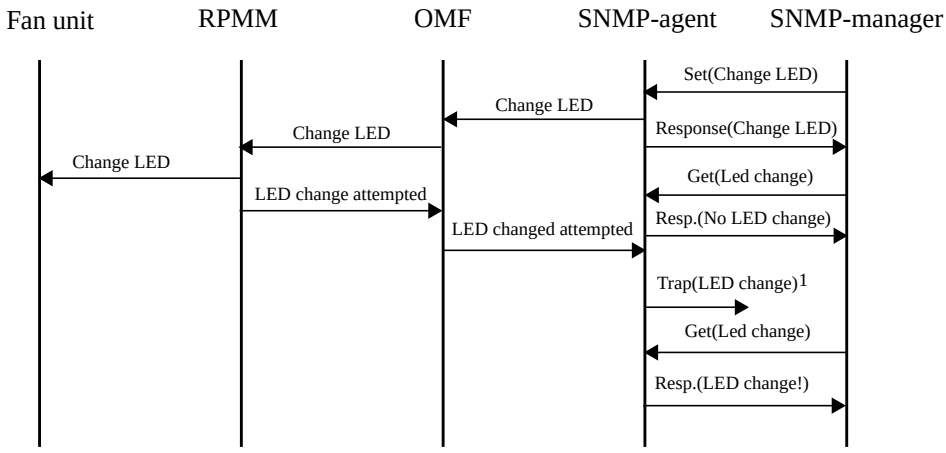

Figure 6.6 Sequence diagram for changing the MIA LED on a fan unit. 1) The trap is considered as being lost.

6.5.3 Case 3, Address update

A third case is what happens when a running RP makes an address update. An address update means that the RP sends all its address data spontaneously such as the subrack address, board address, stable physical address and all logical addresses to a manager. This spontaneous transmission is normally an indication that something is wrong and it is up to the manager to take appropriate action. See the sequence diagram (see Fig. 6.7).

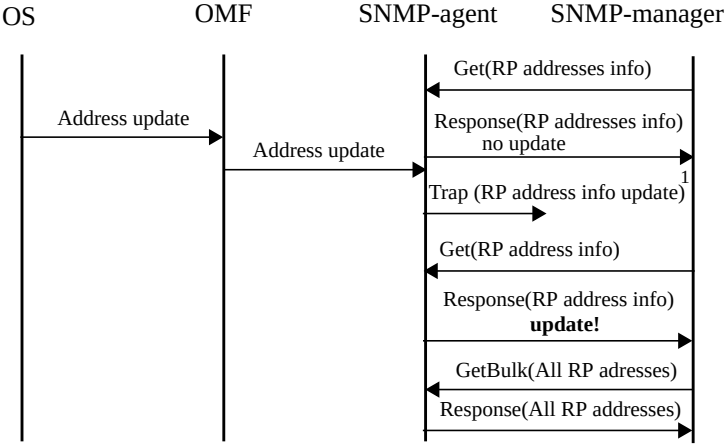

Figure 6.7 Sequence diagram of an RP address update. The manager is constantly polling the RP for changes. 1) The trap generated by the agent is considered to be lost.

6.6 Tools used

Since the time has been limited only small simulations have been made. Simulations for making verifications of the actual job and implementations on a real RP have not been made though. The software tools have been used to compile MIBs and to use SNMP on the compiled data, and to see and present how SNMP works in practice. The specification of this master thesis did not include a real implementation either.

The applications necessary for any kind of simulation are an SNMP manager and an SNMP agent. The manager can be any kind of SNMP manager since it only retrieves and sets SNMP data of an agent. A MIB is loaded to a manager and when the OBJECT-IDENTIFIERS are known the SNMP data can be correctly collected, see section 6.6.1 for more information about MIB compilation.

The requirements on the agent was that it should provide the ability to load and compile a privately defined MIB and make a database from it and also be able to use SNMPv2 and behave accordingly. There doesn't seem to be very many agents supporting those requirement that can be run on a UNIX station. Therefore the programs were obtained for a PC running Windows NT 4.0. The manager came from a company in Slovenia called MG-SOFT and the agent simulator, *SimpleAgent*, from a company in USA called SimpleSoft Inc.

The manger has the capability of sending and retrieving SNMP packets using SNMPv1, v2 or v3. There is also the possibility to receive both types of SNMPv2 traps

from an agent. In the package software for compilation of a MIB is also included, to see that it is syntactically correctly implemented. A MIB data structure is compiled into a readable file which the agent uses to build its data base when it is started.

The manager and agent have then been running on the same machine, but in two different windows, talking SNMPv2 with each other.

6.6.1 General MIB development and compilation

There are a number of programs to check and compile a MIB into an SNMP entity, i.e. manager or agent. See section above for the program chosen for this project. In the development of a real SNMP agent there are a number of steps to take:

- Find out what you want to manage.
- Select predefined MIBs, e.g. MIB-II, or define your own MIB module to be supported by the agent.
- Check (a first step of compilation) the MIB module by running it through a checking program, which usually comes as a part of a MIB development software package. This is just to see that the rules of the SMI are being followed.
- Obtain an SNMP agent that allows to be extended with the MIB of choice.
- Compile the agent and it creates function headers related to the implemented MIB structure for accessing the application data to be managed, the instrumentation.
- Extend the header functions with body code to access the instrumentation.
- Compile the code once again to the system of choice and the agent will be ready to run.

7. Summary and conclusions

In this report, I have presented design and specification of a Management Information Base (MIB), called RP-MIB, for Regional Processors that are used in an AXE exchange. The RP-MIB represents objects that reflect data (variables, tables and states) necessary for operations and maintenance of RPs via the standard protocols and platforms used for network management.

In order to design the RP-MIB, I have investigated what O&M informations transferred between a central processor (CP), which is used as managing entity today, and the RPs. This analysis allowed definition of numerous MIB objects that model data related to O&M in an exchange.

To eventually get a real SNMP agent running in an exchange, which is a real-time system, preliminary steps first had to be taken. These steps included an investigation if an SNMP agent at all could be implemented, and also what information the agent would monitor and manage.

When I started this work I didn't know anything about the target system nor very much about SNMP and nothing about all the concepts related with SNMP. The first thing I had to learn was how the AXE system works today with RPs, CPs, blocks, signalling etc. In parallel with this work I studied the concepts of SNMP, such as how it is used in systems today, what messages it contains, what data is sent in them and whether they are acknowledged or not.

I then studied the concepts of ASN.1, MIBs and the different versions of the SMI. I then transferred the relevant data of the AXE signals in question into a MIB module, the RP-MIB, which can be extended further in the future.

I then got some software to be able to compile the RP-MIB and then run a simulated SNMP agent and manager against each other. This because I wanted to try out response times and how much data that can be fetched in one datagram etc. in practice.

One conclusion to draw is that the concepts of SNMP works almost like the concepts of the AXE system, when it comes to the message distribution. The signals that are sent in AXE are de facto messages sent between processes which can reside on the same hardware or on separated hardware[1].

[1] To be able to send AXE messages between different hardwares a link handler must be used. Link handlers are included in the RPs' operating system, OSE Delta. This makes message passing between processes transparent whether the processes are running on the same hardware, i.e. RP, or not.

The AXE messages contain data for getting and setting data from a managing point of view. There are also signals that are spontaneously sent by RP blocks towards a managing entity, today the CP. SNMP works in a similar way although on a higher level, and normally also over greater distances. But the main advantage with SNMP is that it is a widely spread protocol which makes the systems that use SNMP quite easy to handle.

This thesis has resulted in a MIB implementation, the RP-MIB (see appendix A), of the considered RP-blocks, *RPMBH*, *RPMM*, *RPFD* and *OS*.

This project combined with other aspects have resulted in that Ericsson UAB will have a continuation of this project. First by purchasing SNMP development software, or use older software already available.

We have started development of a real SNMP agent, by using SNMP software adapted for the real-time operating system, OSE Delta, which is used on the SCB-RP platform.

SNMP as a protocol is quite simple to understand and will rely on the TCP/IP stack (called the INET stack by ENEA) that is already a part of OSE Delta in the considered RP types.

I have compiled a real SNMP agent, which does not contain any RP-MIB data yet, and loaded it onto an SCB. The next step is to connect an SNMP manager to it and se if it responds to anything that is already supposed to be supported by it, which are some of the objects that are included in MIB-II. After this is done I will start the real implementation to be able to use the data in the RP-MIB.

Things seem quite promising as they look right now.

8. Problems during the project

- A couple of weeks went by while waiting for the software and hardware to be able to run any kind of simulation.
- Starting up the SNMP agent simulator was at first impossible to do. There was another SNMP entity, a network monitoring agent, using the UDP port, 161. This problem took some time to find and sort out.
- It turned out that the agent simulator was not very sophisticated. 1) It did not support default values. 2) There was no possibility to set any object values by not using SNMP, to simulate instrumentation updates. 3) Table lengths were static, i.e. only one table length per MIB compilation was allowed. These problems did not have a serious impact on the project but were never the less of some nuisance.

9. Further work

The RPs which this thesis considers mainly run under an OS called OSE Delta developed by ENEA. OSE Delta has been the most commonly used OS on Ericsson's RPs for some years now.

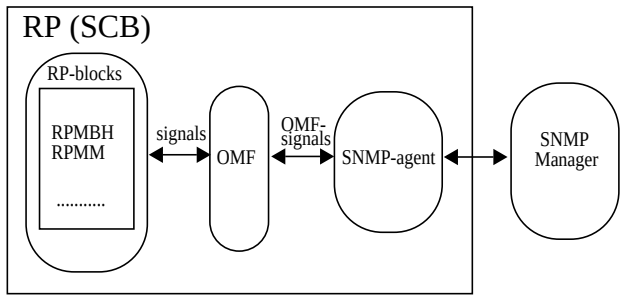

Figure 9.1 The future way in which message handling is to be done between the RP-applications and external entities.

According to ENEA they can provide an SNMP agent that supports the MIB-II, which means it supports their built in TCP/IP stack (INET) including SNMP. The agent can then be 'glued' on an existing RP and be run as an independent application. The writer of this report has been in contact with ENEA and has come to the conclusion that adding a MIB-module to the glued SNMP agent is something that can be done and is also supported in the software.

There seem to be two alternative software choices that can be made. The first alternative originates from SNMP research which is big player on SNMP development tools. ENEA is also working together with them now. The second alternative will be to use SNMP tools from WindRiver, formerly known as Epilogue, which used to cooperate with ENEA.

The next step will then be to connect the MIB data objects to the blocks' variables, which are sometimes referred to as 'the instrumentation', that run on an RP. This will probably be one of the most tricky parts of a whole implementation. However, there is a parallel project at UAB in which development of a translation software module, called the OMF, is in progress and almost finished.

An external application can send signals to the OMF (see Fig. 9.1) and the OMF takes care of the 'unwrapping' of the raw response data from the RP-block and then forwards the requested response to the application that requested it, for example an SNMP agent.

Another problem that might arise is the well known problem of inconsistency. Will several applications have access to the same instrumentation? If there are continuous updates of the instrumentation data, will every change then be forwarded to the agent? These are probable problems that will need to be discussed in the future.

10. Own comments

When I started this work I basically only knew the theory behind SNMP, what it was used for and loosely about the messages within the protocol. I didn't know what a MIB was and even less about ASN.1. Now I think that I have a good knowledge about it and steps that are necessary to take, and avoid, to eventually get a functional SNMP agent on a system. I didn't really know very much about the SNMP surroundings when I started and even less about the AXE exchange system that Ericsson uses today.
This thesis hasn't really been focused on the SNMP protocol itself, but rather how to extract data from an existing system and then port it into a MIB for later use by SNMP. I think that the specified goals for this thesis have been reached and that it is now possible to proceed further.

11. Abbreviations

ALU	Arithmetic and Logic Unit
APT	The application part of an AXE system
APZ	The controlling part of an AXE system
ASN.1	Abstract Syntax Notation One
CCITT	Commité Consultatif International de Telegraphique et Telephonique
CM	Control Module
CP	Central Processor
EM	Extension Module
FTP	File Transfer Protocol
HDLC	High-level Data Link Control
IAB	Internet Architecture Board
IANA	Internet Assigned Numbers Authority
ICMP	Internet Control Message Protocol
IETF	Internet Engineering Task Force
ISO	International Organization for Standardization
LAN	Local Area Network
LED	Light-Emitting Diode
MIA	Manual Intervention Allowed
MIB	Management Information Base
MTU	Maximum Transmission Unit
OMF	Operation and Maintenance Framework
OS	Operating System
OSI	Open Systems Interconnection
O&M	Operations and Maintenance
PDU	Protocol Data Unit
RFC	Request For Comment
RP	Regional Processor
RPC	Remote Procedure Call
SCB	Support and Connection Board. Can maintain other boards and fan units.
SNMP	Simple Network Management Protocol
SMI	Structure of Management Information
TFTP	Trivial File Transfer Protocol
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
WAN	Wide Area Network

12. References

[1] **SNMP, SNMPv2, SNMPv3 and RMON 1 and 2, the third edition by William Stallings. 1999. ISBN: 0-201-48534-6.**

[2] **MIB for an SNMP based maintenance of an RPG in an AXE10, by Pernilla Jansson and Chaowana Kusonkhum (bachelor's thesis). Haninge College of Engineering, Royal Institute of Technology. June 1998.**

[3] **Getting to know AXE. An Ericsson training document. EN/LZT 101 548 R2A. 1987.**

[4] **Operation and maintenance of stand alone platforms. An Ericsson internal report. UAB/B/R-97:399 Uen. February 1998.**

[5] **Network Management by Ponthus Nyrelli. First part of a diploma thesis. October 1997.**

[6] **Slides from the Network Management Course 2g5552 at KTH by Volker Lausch. http://www.it.kth.se/edu/Ph.D/NM99/**

[7] **RFC 1155, Structure and Identification of Management Information for TCP/IP based Internets, by M.Rose and K.McCloghrie. http://www.ietf.org/rfc.html. May 1990.**

[8] **RFC 1212, Concise MIB Definitions, by M. Rose and K. McCloghrie. http://www.ietf.org/rfc.html. April 1991.**

[9] **RFC 1213, Structure and Identification of Management Information for TCP/IP-based Internets MIB-II, by M.Rose and K.McCloghrie. http://www.ietf.org/rfc.html. March 1991.**

[10] **FOLDOC, Free- On Line Dictionary Of Computing. http://ftp.sunet.se/foldoc/index.html**

13. Acknowledgments

I would like to thank the following people for their contributions to this thesis.

Lennart Malmberg, for explaining the AXE system both in general and especially when it comes to RPMBH and RPMM, and also for being a good mentor.

Richard Tham, for initiating the project and for getting tools needed for the project.

Håkan Magnusson, for explaining the features of the RPFD block.

Kjell Persson and Lars Jönsson for contributing input about the OS.

Stefan Wallin at DataDuctus AB, for suggesting SNMP testing tools.

Håkan Trygg at ENEA, for explaining some of the OMF and OSE features.

I would also like to thank the people at UAB/Y/I in general for making me feel welcome.

14. Appendices

APPENDIX A

```
RP-MIB DEFINITIONS ::= BEGIN

IMPORTS
    MODULE-IDENTITY, OBJECT-TYPE, NOTIFICATION-TYPE,
    experimental             FROM SNMPv2-SMI
--  RowStatus is used for monitoring what access level a row in a table
--  with createble rows has from the manager's point of view. (The agent
--  always has full access to own data). Here only fixed tables are used.
--  RowStatus                                    FROM SNMPv2-TC
    OBJECT-GROUP, NOTIFICATION-GROUP  FROM SNMPv2-CONF;


rPMIB MODULE-IDENTITY
    LAST-UPDATED "200102090000Z"
    ORGANIZATION "Ericsson UAB"
    CONTACT-INFO
             "Per Holmgren
              Ericsson Utvecklings AB.
                      Götalandsvägen 230
                      12525 Älvsjö
                      Sweden
                      Phone: +46 (0)8 727 1469
              E-Mail: per.holmgren@uab.ericsson.se"
    DESCRIPTION
            "The RP-MIB module for implementation in an RP supporting the RP
             blocks RPMBH, RPMM, RPFD and RP-OS. This module is intended for
             the SCB-board mainly, but can be extended/reduced to fit other
             RPs too. The SCB-board can be in charge for the O&M part of up to
             32 slots and up to 8 fan units.
             NOTE: Only informations known through the CP-RP
             'interface' are included"
        ::={experimental 1} --Will probably use {enterprises 193 ~80}

-- The data groups of RP-MIB

          rPMBH     OBJECT IDENTIFIER ::={rPMIB 3}

          rPMM      OBJECT IDENTIFIER ::={rPMIB 4}

          rPFD      OBJECT IDENTIFIER ::={rPMIB 5}

          rPOS      OBJECT IDENTIFIER ::={rPMIB 6}


rPGlobalChange OBJECT-TYPE
       SYNTAX       OCTET STRING(SIZE(4))
```

```
        MAX-ACCESS   read-write
        STATUS       current
        DESCRIPTION
            "rpGlobalChange is a global object for finding any type of unusual
             events (changes/alarms etc) that has happened in the RP. The
manager
             should poll this object continously. Each bit of the string
             (4 bytes = 32 bits) represents one RP. When manager has seen this
             event he is responible for setting the value to normal (Bit#=0)
             again.
             This object (rPGlobalChange)represents the following objects of
             this MIB module:
             rPSlotDataChange
             rPPowerStatus
             rPTemporaryPowFaultsNum
             rPPowerPathChange
             fanMIALEDSetRes
             fanDataorHWIStatus
             rPEMErrorType
             rPEMAlarmWord
             rPErrorCase
             rPErrorCode
             rPErrorData
             rPAddrInfo
             Most of the above objects also have trap equivalences (except
             rPSlotDataChange). Neither of the above objects represent a state
             which is very common to changes.
             Bit#:
                  0 = No change (normal)
                  1 = Unusual event"
        ::={rPMIB 1}


rPGlobalChangeTrap NOTIFICATION-TYPE
        OBJECTS     {rPGlobalChange}
        STATUS      current
        DESCRIPTION
           "rPSlotDataOneTrap is sent to the manager informing about that a
            sudden change/update has happened in up to 32 slots. The included
            data is a 4 byte string.
            See rPGlobalChange (which is the poll equivalent) for mor info."
        ::={rPMIB 2}


rPBoardTable OBJECT-TYPE
        SYNTAX      SEQUENCE OF RPBoardEntry
        MAX-ACCESS  not-accessible
        STATUS      current
        DESCRIPTION
            "The conceptual (not real) table listing the LED-state and PROM-
             data of a board in the RP, by using the slot number as the index"
        ::={rPMBH 1}
```

```
rPBoardEntry OBJECT-TYPE
        SYNTAX      RPBoardEntry
        MAX-ACCESS  not-accessible
        STATUS      current
        DESCRIPTION
             "A conceptual (not real row) entry in the RPBoardTable. The
              table is indexed by the slot number"
        INDEX       { rPSlotNumber }
        ::={rPBoardTable 1}


RPBoardEntry ::= SEQUENCE{
                          rPSlotNumber     INTEGER(1..32),--INTEGER(0..31),
but does not like index 0
                          rPBoardLEDState  INTEGER(0..1),
                          rPHWIPromData    OCTET STRING(SIZE(1..255)),
                          rPSlotDataChange OCTET STRING(SIZE(1))}


rPSlotNumber OBJECT-TYPE
        SYNTAX      INTEGER(1..32) --Does not like an index 0 in compilation
        MAX-ACCESS  read-only
        STATUS      current
        DESCRIPTION
            "Slot number is valid for numbers
             0-31"
        ::={rPBoardEntry 1}


rPBoardLEDState OBJECT-TYPE
        SYNTAX      INTEGER(0..1)
        MAX-ACCESS  read-write
        STATUS      current
        DESCRIPTION
            "MIA LED-state on a board
             0 = off
             1 = on"
        ::={rPBoardEntry 2}


rPHWIPromData OBJECT-TYPE
        SYNTAX      OCTET STRING(SIZE(1..255))
        MAX-ACCESS  read-only
        STATUS      current
        DESCRIPTION
            "Up to 255 bytes HWI PROM data for a hardware board. Usually
            less than 100 bytes(100 octets). The maximum number of bytes
            possible to retreive is rPPromMaxLen number of bytes"
        ::={rPBoardEntry 3}


rPSlotDataChange OBJECT-TYPE
        SYNTAX      OCTET STRING(SIZE(1))
        MAX-ACCESS  read-only
```

```
        STATUS      current
        DESCRIPTION
            "rPSlotDataChange is a 1 byte string informing the manager about
             the details of the changes in the slot.
             B0-B1: Board code
                    0 = Not used    --Could be OK instead
                    1 = No board
                    2 = New board
                    3 = Fault
             B2-B3: Not used
             B4-B7: Fault code, valid if board code is 3
                    0 = Missing EOT
                    1 = Invalid checksum
                    2 = Other fault
                3-15 = Not used"
        ::={rPBoardEntry 4}


rPPollPrinciple OBJECT-TYPE
        SYNTAX       INTEGER(0..2)
        MAX-ACCESS   read-write
        STATUS       current
        DESCRIPTION
            "Polling principle of an RP is an integer. Is useful
             for a manager to keep track of if polling is at
             all necessary for this RP.
             0 = Polling off
             1 = Polling performed once
             2 = Polling autonomous (normal)"
        ::={rPMBH 2}


rPPollInterval OBJECT-TYPE
        SYNTAX      INTEGER(0..31)
        UNITS       "Every 10 seconds"
        MAX-ACCESS  read-write
        STATUS      current
        DESCRIPTION
            "Polling interval is a 1 byte integer. Interval in units of 10
             seconds. This can be usful information for the manager to know
             how often polling for RP update is useful.
             0-31 units"
        ::={rPMBH 3}


rPOpsHWICHANGE OBJECT-TYPE
        SYNTAX      OCTET STRING(SIZE(1))
        MAX-ACCESS  read-write
        STATUS      current
        DESCRIPTION
            "OPS-pointer for sponataneous signal HWICHANGE"
        ::={rPMBH 4}
```

```
rPPromMaxLen OBJECT-TYPE
        SYNTAX      INTEGER(0..255)
        UNITS       "Bytes"
        MAX-ACCESS  read-write
        STATUS      current
        DESCRIPTION
            "PROM data maximum length is the maximum allowed length in bytes
             before a boards PROM is considered faulty
             0-255"
        ::={rPMBH 5}


rPOpsRPMMFANSTAT OBJECT-TYPE
        SYNTAX      OCTET STRING(SIZE(1))
        MAX-ACCESS  read-write
        STATUS      current
        DESCRIPTION
            "OPS-pointer for spontaneous signal RPMMFANSTAT"
        ::={rPMM 1}


rPOpsRPMMPOWSTAT OBJECT-TYPE
        SYNTAX      OCTET STRING(SIZE(1))
        MAX-ACCESS  read-write
        STATUS      current
        DESCRIPTION
            "OPS-pointer for spontaneous signal RPMMPOWSTAT"
        ::={rPMM 2}


rPPowerStateA OBJECT-TYPE
        SYNTAX      INTEGER(1..2)
        MAX-ACCESS  read-only
        STATUS      current
        DESCRIPTION
            "State of power A
             1 = No fault
             2 = Permanent fault"
        ::= {rPMM 3}


rPTemporaryPowFaultsNumA OBJECT-TYPE
        SYNTAX      INTEGER(0..255)
        MAX-ACCESS  read-only
        STATUS      current
        DESCRIPTION
           "Total number of temporary faults on power A since last restart of
             RPMMR"
        ::={rPMM 4}


rPPowerStateB OBJECT-TYPE
        SYNTAX      INTEGER(1..2)
        MAX-ACCESS  read-only
```

```
        STATUS      current
        DESCRIPTION
            "State of power B
             1 = No fault
             2 = Permanent fault"
        ::={rPMM 5}


rPTemporaryPowFaultsNumB OBJECT-TYPE
        SYNTAX      INTEGER(0..255)
        MAX-ACCESS  read-only
        STATUS      current
        DESCRIPTION
            "Total number of temporary faults on power B since last restart of
             RPMMR"
        ::={rPMM 6}


rPPowerStatusTrap NOTIFICATION-TYPE
        OBJECTS     {rPPowerStatus,
                     rPTemporaryPowFaultsNum,
                     rPPowPathChange}
        STATUS      current
        DESCRIPTION
             "Is sent to inform manager about sudden changes of the
             power status. This information could be polled
             continously for too."
        ::={rPMM 7}


rPPowerStatus OBJECT-TYPE
        SYNTAX      INTEGER(1..3)
        MAX-ACCESS  read-only
        STATUS      current
        DESCRIPTION
            "rPPowerStatus is to be changed when a change in power
             status is detected
             1 = No fault
             2 = Permanent fault
             3 = Temporary fault"
        ::={rPMM 8}


rPTemporaryPowFaultsNum OBJECT-TYPE
        SYNTAX      INTEGER(0..255)
        MAX-ACCESS  read-only
        STATUS      current
        DESCRIPTION
            "Number of temporary power faults since last trap sent is a 1 byte
             integer. This value SHOULD be zeroed after trap has been sent.
             (If counter32: However, a Counter32 is not read for its value but
             rather the difference between two readings.)
             Is only valid when rPPowerStatus is set permanent (2) or
             temporary (3)"
```

```
        ::= {rPMM 9}


rPPowPathChange OBJECT-TYPE
        SYNTAX      INTEGER(1..2)
        MAX-ACCESS  read-only
        STATUS      current
        DESCRIPTION
            "Power path change is an integer which informs the manager
             which path that had a power change
             1 = A
             2 = B"
        ::={rPMM 10}


fanDataTable OBJECT-TYPE
        SYNTAX      SEQUENCE OF FanDataEntry
        MAX-ACCESS  not-accessible
        STATUS      current
        DESCRIPTION
            "Table (conceptual) containing up to 8 fanDataEntries"
        ::={rPMM 11}


fanDataEntry OBJECT-TYPE
        SYNTAX      FanDataEntry
        MAX-ACCESS  not-accessible
        STATUS      current
        DESCRIPTION
            "A conceptual row containing the fan unit number as index and air
             temp, LED state, read result, communication error details and
             alarm information as relevant data for each fan. The table also
             includes status information about HWI data and data status
             for the fan"
        INDEX {fanUnitNumberInd}
        ::={fanDataTable 1}


FanDataEntry ::= SEQUENCE {
                            fanUnitNumberInd   INTEGER(1..8),
                            fanReadResult      INTEGER(0..4),
                            fanCommErrorDet    INTEGER(0..3),
                            fanAlarmInfo       OCTET STRING(SIZE(2)),
                            fanAirTemp         OCTET STRING(SIZE(2)),
                            fanMIALEDState     INTEGER (0..3),
                            fanDataStatus      INTEGER(0..3),
                            fanHWIStatus       INTEGER(0..3),
                            fanHWIReadRes      INTEGER(0..5),
                            fanHWIPromData     OCTET STRING(SIZE(1..255))}


fanUnitNumberInd OBJECT-TYPE
        SYNTAX      INTEGER(1..8)
        MAX-ACCESS  read-only
```

```
        STATUS      current
        DESCRIPTION
            "fanUnitNumber is a number representing fan 1-8. This object
             represents the indexing of the fanDataTable"
        ::={fanDataEntry 1}


fanReadResult OBJECT-TYPE
        SYNTAX      INTEGER(0..4)
        MAX-ACCESS  read-only
        STATUS      current
        DESCRIPTION
            "Read result is an integer informing how well data collection has
              been made from a fan
                 0 = Successfully read fan data (Fan alarm information, fan
air
                     temperature and fan MIA LED state)
                 1 = Not used
                 2 = Fan fault, no fan connected
                 3 = Fan fault, communication error
                 4 = Requested fan unit is out of range
              5-255 = Not used"
        ::={fanDataEntry 2}


fanCommErrorDet OBJECT-TYPE
        SYNTAX      INTEGER(0..3)
        MAX-ACCESS  read-only
        STATUS      current
        DESCRIPTION
            "Communication error details. An addition to the previously read
             result if there was a communications error (fanReadResult == 3).
                 0 = Framing error
                 1 = Checksum error
                 2 = Protocol error
                 3 = Addressing mode error
              4-255 = Not used"
        ::={fanDataEntry 3}


fanAlarmInfo OBJECT-TYPE
        SYNTAX      OCTET STRING(SIZE(2))
        MAX-ACCESS  read-only
        STATUS      current
        DESCRIPTION
            "Two bytes: The first byte mirrors fanReadResult (integer 0-4) and
             the second byte represents the fan alarm information (0*00 -
0*ff).
             Fan alarm information is a 1 byte string about the physical state
             of a fan, where each bit out of the one byte describes a state.
             Is only valid if fanReadResult is 0.
            BYTE no. 2:
             B0: 0 = Temperature < 65*C
                 1 = Temperature > 65*C
```

```
            B1: 0 = Temperature < 55*C
                1 = Temperature > 55*C
            B2: 0 = Temperature sensor working
                1 = Temperature sensor out of range
            B3: 0 = Motor current OK
                1 = Motor current out of range
            B4: 0 = Fan internal communication OK
                1 = Fan internal communication error
            B5: 0 = Fan motor regulation OK
                1 = Fan motor regulation error
            B6: 0 = Power branch A (-48) OK
                1 = Power branch A (-48) error
            B7: 0 = Power branch B (-48) OK
                1 = Power branch B (-48) error"
        ::={fanDataEntry 4}


fanAirTemp OBJECT-TYPE
        SYNTAX      OCTET STRING(SIZE(2))
        MAX-ACCESS  read-only
        STATUS      current
        DESCRIPTION
            "Two bytes: First byte mirrors fanReadResult (integer 0-4)
             and the second byte represents fan air temperature (integer
             0-138). This states the temperature in the fan air.
             Is only valid if fanReadResult is 0.
              BYTE no. 2:
               0-104 = Plus (0*C to 104*C)
             128-138 = Minus (-1*C to -10*C)"
        ::={fanDataEntry 5}


fanMIALEDState OBJECT-TYPE
        SYNTAX      INTEGER(0..3)
        MAX-ACCESS  read-write
        STATUS      current
        DESCRIPTION
            "Fan MIA LED state. Is only usable if fanReadResult is 0.
             When fanMIALEDstate is changed by the manager the agent will
             transmit the order to RPMM. RPMM will send back
             an attempt to change the LED. The attempt is
             stored in fanMIALEDSetRes.
                 0 = LED off
                 1 = Slow flash
                 2 = Fast flash
                 3 = LED on
             4-255 = Not used"
        ::={fanDataEntry 6}


fanDataStatus OBJECT-TYPE
        SYNTAX      INTEGER(0..3)
        MAX-ACCESS  read-only
        STATUS      current
```

```
        DESCRIPTION
            "Fan Data status is an integer. It represents the fan status
             in regards of data availability.
                0 = Fan connected, fan data available
                1 = Not used
                2 = Fan fault, no fan connected
                3 = Fan fault, communication error
             4-255 = Not used"
        ::={fanDataEntry 7}


fanHWIStatus OBJECT-TYPE
        SYNTAX      INTEGER(0..3)
        MAX-ACCESS  read-only
        STATUS      current
        DESCRIPTION
            "Fan HWI status is a 1 byte integer. It represents the fan status
             in regards of HWI data availability.
                0 = Fan connected, HWI data available
                1 = Not used
                2 = HWI fault, no fan connected
                3 = HWI fault, communication error
            4-255 = Not used"
        ::={fanDataEntry 8}


fanHWIReadRes OBJECT-TYPE
        SYNTAX      INTEGER(0..5)
        MAX-ACCESS  read-only
        STATUS      current
        DESCRIPTION
            "Fan and HWI read result is a 1 byte integer
                 0 = Successful read, all data read
                 1 = Successful read, more data to read
                 *SHOULDNT BE NECESSARY, can read all at once*
                 2 = HWI fault, no fan connected
                 3 = HWI fault, communication error
                 4 = The requested fan unit is out of range
                 5 = The requested data is out of range
             6-255 = Not used"
        ::={fanDataEntry 9}


fanHWIPromData OBJECT-TYPE
        SYNTAX      OCTET STRING(SIZE(1..255))
        MAX-ACCESS  read-only
        STATUS      current
        DESCRIPTION
            "The sequential HWI PROM data"
        ::={fanDataEntry 10}


fanUnitNumber OBJECT-TYPE
        SYNTAX      INTEGER(1..8)
```

```
        MAX-ACCESS  read-only
        STATUS      current
        DESCRIPTION
            "fanUnitNumber is a number representing fan 1-8. This
             object is not supposed to be read, is instead supposed
             to be contained in a notification-type (which does not
             support not-accessible objects)"
        ::={rPMM 12}


fanMIALEDSetResTrap NOTIFICATION-TYPE
        OBJECTS     {fanMIALEDSetRes}
        STATUS      current
        DESCRIPTION
           "Is sent to manager when receiving an order from manager to change
            the LED-state of appointed fan unit. Does not promise anything
            but indicates that the RP will do make a best effort"
        ::={rPMM 13}


fanMIALEDSetRes OBJECT-TYPE
        SYNTAX      OCTET STRING(SIZE(8))
        MAX-ACCESS  read-only
        STATUS      current
        DESCRIPTION
            "This object does not guarantee that the LED is really set for the
            intended fan unit. It only indicates that the regional program
will
            make a best effort (Regional program programmed that way!).
            However this object indicates that there might
            be a fan problem. Each byte represents one fan unit. Each byte can
            have values 0-5.
            NOTE: This is a single object that is used for all fan units.
            Byte#:
             0 = Setting of LED will be executed
             1 = Not used
             2 = Fan fault, no fan connected
             3 = Fan fault, communication error
             4 = Requested fan unit is out of range
             5 = Invalid MIA LED state requested"
        ::={rPMM 14}


fanDataorHWIStatusTrap NOTIFICATION-TYPE
        OBJECTS     {fanDataorHWIStatus}
        STATUS      current
        DESCRIPTION
           "Is sent to manager when change in fan status or in HWI status"
        ::={rPMM 15}


fanDataorHWIStatus OBJECT-TYPE
        SYNTAX      OCTET STRING(SIZE(1))
        MAX-ACCESS  read-write
```

```
        STATUS      current
        DESCRIPTION
            "An eight bit string. Each bit represents one fan unit. This
             object can be polled by the manager to find out that some
             kind of fan unit problem has occurred (data communication
             or HWI status). When fault the agent sets the bit in question
             and is then seen by the manager by polling. The manager
             is then responsible for resetting the bit for that fan unit.
             0*00 = normal
             0*01 = fault in fan unit one
              .
              .
             0*80 = fault in fan unit eight
             Example: 0*0C (00001010) means fault in both fan unit 2 and 4."
        ::={rPMM 16}


--RPFD MIB group

--Different functions for the rpfd group

rPFDStartFunc          OBJECT IDENTIFIER ::={rPFD 1}

rPFDRPTestInitFunc     OBJECT IDENTIFIER ::={rPFD 2}

rPFDEMTestFunc         OBJECT IDENTIFIER ::={rPFD 3}

rPFDRPTestFunc         OBJECT IDENTIFIER ::={rPFD 4}

rPFDEMErrorFunc        OBJECT IDENTIFIER ::={rPFD 5}

rPFDRPErrorDataReqFunc OBJECT IDENTIFIER ::={rPFD 6}

rPFDRPSupCircTestFunc  OBJECT IDENTIFIER ::={rPFD 7}


rPOpsRPEMERROR OBJECT-TYPE
        SYNTAX      OCTET STRING(SIZE(1))
        MAX-ACCESS  read-write
        STATUS      current
        DESCRIPTION
            "rPOpspointer for signal RPEMERROR "
        ::={rPFDStartFunc 1}


rPMaxEMNum OBJECT-TYPE
        SYNTAX      INTEGER{sixteen(16),
                            sixtyfour(64)}
        MAX-ACCESS  read-write
        STATUS      current
        DESCRIPTION
            "Maximum number of EM
             16 or 64"
        ::={rPFDStartFunc 2}
```

```
rPTestTable OBJECT-TYPE
        SYNTAX      SEQUENCE OF RPTestEntry
        MAX-ACCESS  not-accessible
        STATUS      current
        DESCRIPTION
            "The conceptual (not real) table listing the test state of
             the RP"
        ::={rPFDRPTestInitFunc 1}


rPTestEntry OBJECT-TYPE
        SYNTAX      RPTestEntry
        MAX-ACCESS  not-accessible
        STATUS      current
        DESCRIPTION
            "A conceptual (not real row) entry in the rPTestTable. The table
              is indexed by the rPTestType"
        INDEX       { rPTestType }
        ::={rPTestTable 1}


RPTestEntry ::= SEQUENCE{
                        rPTestType  INTEGER(1..4),
                        rPTestState INTEGER(0..1)}


rPTestType OBJECT-TYPE
        SYNTAX      INTEGER(1..4)
        MAX-ACCESS  read-write
        STATUS      current
        DESCRIPTION
            "Type of RP test to be done. The tests can be done in parallel,
             which means that the table is four rows long.
             1 = PS-test
             2 = DS-test
             3 = CPU-test
             4 = EM-test"
        ::={rPTestEntry 1}


rPTestState OBJECT-TYPE
        SYNTAX      INTEGER(0..1)
        MAX-ACCESS  read-write
        STATUS      current
        DESCRIPTION
            "Sets appointed test passive or active
             0 = Passive
             1 = Active"
        ::={rPTestEntry 2}


rPEMTestTable OBJECT-TYPE
```

```
        SYNTAX      SEQUENCE OF RPEMTestEntry
        MAX-ACCESS  not-accessible
        STATUS      current
        DESCRIPTION
            "The conceptual (not real) table listing the state in which the
             test of an EM is in"
        ::={rPFDEMTestFunc 1}


rPEMTestEntry OBJECT-TYPE
        SYNTAX      RPEMTestEntry
        MAX-ACCESS  not-accessible
        STATUS      current
        DESCRIPTION
             "A conceptual (not real row) entry in the rPTestStateTable. Each
              row consists of EMTestState and the corresponding CM-number"
        INDEX       {rPEMAddress}
        ::={rPEMTestTable 1}

RPEMTestEntry ::= SEQUENCE{
                          rPEMAddress   INTEGER(1..64), --INTEGER(0..64),
                          rPEMTestState INTEGER(0..1),
                          rPCMNumber    INTEGER(0..15),
                          rPEMTestOrder INTEGER(0..3)}

rPEMAddress OBJECT-TYPE
        SYNTAX      INTEGER(1..64) --Does not like an index 0 in compilation
        MAX-ACCESS  read-only
        STATUS      current
        DESCRIPTION
           "The EM-address
            0-64"
        ::={rPEMTestEntry 1}


rPEMTestState OBJECT-TYPE
        SYNTAX      INTEGER(0..1)
        MAX-ACCESS  read-only
        STATUS      current
        DESCRIPTION
            "EM-address test mode
             0 = Passive
             1 = Active"
        ::={rPEMTestEntry 2}


rPCMNumber OBJECT-TYPE
        SYNTAX      INTEGER(0..15)
        MAX-ACCESS  read-only
        STATUS      current
        DESCRIPTION
           "The CM-number which is in control of the EM tested
            0-15"
        ::={rPEMTestEntry 3}
```

```
rPEMTestOrder OBJECT-TYPE
        SYNTAX      INTEGER(0..3)
        MAX-ACCESS  read-write
        STATUS      current
        DESCRIPTION
            "Test order of EM is an integer.
             0 = Start EM routine test, no acknowledge is wanted by CP
             1 = Test specified EM and send test result
             2 = Test all EM and send test result
             3 = Start EM routine test, acknowledge wanted before test"
        ::={rPEMTestEntry 4}


rPFirstTest OBJECT-TYPE
        SYNTAX      INTEGER(0..3)
        MAX-ACCESS  read-write
        STATUS      current
        DESCRIPTION
            "First test is a 1 byte integer
             1 = PS-test
             2 = DS-test
             3 = CPU-test"
        ::={rPFDRPTestFunc 1}


rPAckOrder OBJECT-TYPE
        SYNTAX      INTEGER(0..4)
        MAX-ACCESS  read-write
        STATUS      current
        DESCRIPTION
            "If, how and when to acknowledge a test order
             0 = No acknowledge
             1 = Acknowledge after PS-test
             2 = Acknowledge after DS-test
             3 = Acknowledge after CPU-test
             4 = Acknowledge before test start"
        ::={rPFDRPTestFunc 2}


rPTestTime OBJECT-TYPE
        SYNTAX      INTEGER(0..40)
        MAX-ACCESS  read-write
        STATUS      current
        DESCRIPTION
            "How long an RP-test should run
              0 = Normal test time (0.1MS/P.I.)
             40 = Long test time  (4MS/P.I.)"
        ::={rPFDRPTestFunc 3}


rPEMErrorTrap NOTIFICATION-TYPE
        OBJECTS     {rPEMAddress2, rPEMErrorType, rPEMAlarmWord}
```

```
        STATUS      current
        DESCRIPTION
           "RPEMError is sent to the manager when an EM error has occured
            NOTE: SCB-boards does not have any EM:s to control"
        ::={rPFDEMErrorFunc 1}


rPEMAddress2 OBJECT-TYPE
        SYNTAX      INTEGER(0..64)
        MAX-ACCESS  read-only
        STATUS      current
        DESCRIPTION
           "The EM-address
            0-15 or 0-63
            NOTE: SCB-boards does not have any EM:s to control"
        ::={rPFDEMErrorFunc 2}


rPEMErrorType OBJECT-TYPE
        SYNTAX      INTEGER(0..2)
        MAX-ACCESS  read-only
        STATUS      current
        DESCRIPTION
            "Type of EM error is a 1 byte integer. This object
             can be polled by a managing entity to find out that
             a problem has occurred.
             0 = No error
             1 = Signalling error
             2 = EM alarm
             NOTE: SCB-boards does not have any EM:s to control"
        ::={rPFDEMErrorFunc 3}


rPEMAlarmWord OBJECT-TYPE
        SYNTAX      OCTET STRING(SIZE(1))
        MAX-ACCESS  read-only
        STATUS      current
        DESCRIPTION
            "Alarm word is a 1 byte string
            NOTE: SCB-boards does not have any EM:s to control"
        ::={rPFDEMErrorFunc 4}


rPTestOrderSupCirc OBJECT-TYPE
        SYNTAX      INTEGER(0..3)
        MAX-ACCESS  read-write
        STATUS      current
        DESCRIPTION
            "Test order of supervision circuit in RP.
             0 = Testing program and memory parity circuit
             1 = Testing microprogram parity circuit
             2 = Testing PHC
             3 = Testing address limit circuit"
        ::={rPFDRPSupCircTestFunc 1}
```

```
rPErrorCase OBJECT-TYPE
        SYNTAX      OCTET STRING(SIZE(1))
        MAX-ACCESS  read-write
        STATUS      current
        DESCRIPTION
            "Error case 1 byte
             B0: Parity error in program or data memory
             B1: Parity error in microprogram memory
             B2: PHC alarm
             B3: Program execution fault or fault detected during routine
             test
             The manager has to reset this object when change has been read"
        ::={rPFDRPSupCircTestFunc 2}


rPErrorCaseTrap NOTIFICATION-TYPE
        OBJECTS     {rPErrorCase}
        STATUS      current
        DESCRIPTION
           "Is sent to manager when rPErrorCase is changed/set"
        ::={rPFDRPSupCircTestFunc 3}


rPErrorCode OBJECT-TYPE
        SYNTAX      OCTET STRING(SIZE(1))
        MAX-ACCESS  read-write
        STATUS      current
        DESCRIPTION
            "Error code is a 1 byte hex string informing about the RP error
             code type
             The manager has to reset this when change has been read"
        ::={rPFDRPErrorDataReqFunc 1}


rPErrorData OBJECT-TYPE
        SYNTAX      OCTET STRING(SIZE(4))
        MAX-ACCESS  read-write
        STATUS      current
        DESCRIPTION
            "RP error data is a four byte hex string that specifies the error
             code in more detail
             The manager has to reset this when change has been read"
        ::={rPFDRPErrorDataReqFunc 2}


rPErrorCodeandDataTrap NOTIFICATION-TYPE
        OBJECTS     {rPErrorCode, rPErrorData}
        STATUS      current
        DESCRIPTION
           "Is sent to manager when rPErrorCode and rPErrorData is changed/
set"
        ::={rPFDRPErrorDataReqFunc 3}
```

```
rPOSPhysState  OBJECT IDENTIFIER ::= {rPOS 1}

rPOSMagAddr    OBJECT IDENTIFIER ::= {rPOS 2}


rPBootProgId OBJECT-TYPE
        SYNTAX      OCTET STRING(SIZE(32))
        MAX-ACCESS  read-only
        STATUS      current
        DESCRIPTION
            "Product identity of boot program in non-volatile memory is a 32
             byte string"
        ::={rPOSPhysState 1}


rPDramSize OBJECT-TYPE
        SYNTAX      INTEGER(0..2147483647)
        MAX-ACCESS  read-only
        STATUS      current
        DESCRIPTION
            "DRAM memory size is a 4 byte integer"
        ::={rPOSPhysState 2}


rPOsProp OBJECT-TYPE
        SYNTAX      OCTET STRING(SIZE(46))
        MAX-ACCESS  read-only
        STATUS      current
        DESCRIPTION
            "OS properties is a 3-46 byte string
                 1: Number of valid bytes that follow (2-45)
                 2: Properties vector (Bit# = 1 when true)
                   B0: Support for compressed code
                       More info in byte 4 when set
                   B1: Support for no rotation of load data
                   B2: Is able to receive RPADDRTORP
                   B3: Support for advanced start of program execution and
                       individual load of RSU
                   B4: Support for TEST SYSTEM debugging of RP
                   B5: Support for Extended Error Information
                  B6: Support for individual load of RSU, but NOT support for
                       advanced start
                   B3 and B6 can not both be true

                 3: Properties vector reserved (always 0)
                 4: The highest number of the compression algorithm supported
                   (0, No support for compression algorithm)
                 5: DRAM parity
                   B1-B0: 00 - Available     (Enabled)
                          01 - Not available (Disabled)
```

```
                          11 - Not defined

              6-46: Reserved (always 0)"
        ::={rPOSPhysState 3}


rPPhysTypeNum OBJECT-TYPE
        SYNTAX      OCTET STRING(SIZE(3))
        MAX-ACCESS  read-only
        STATUS      current
        DESCRIPTION
            "Physical RP type number and OS software compatibility number is a
             2 or 3 byte ASCII string
             1: Number of valid bytes to follow (1 or 2)
             2: Physical RP number
             3: OS software compatibility number"
        ::={rPOSPhysState 4}


rPProcBoardId OBJECT-TYPE
        SYNTAX      OCTET STRING(SIZE(31))
        MAX-ACCESS  read-only
        STATUS      current
        DESCRIPTION
            "Processor board identity is a 31 byte ASCII string
             (Obsolete RP:s have 32 byte strings, but is not considered here)
              1-24: Processor board ID
             25-31: Processor board revision"
        ::={rPOSPhysState 5}


rPStatCount OBJECT-TYPE
        SYNTAX      OCTET STRING(SIZE(21))
        MAX-ACCESS  read-only
        STATUS      current
        DESCRIPTION
            "Statistic counters for RP connected to RPB-S is a 21 byte string
             in total.
                 1: 0 (dummy)
               2-5: Number of received signals
               6-7: Number of overruns
               8-9: Number of received bad frames
             10-11: Number of received frames shorter than min.length
             12-15: Number of transmitted signals
             16-17: Number of overruns
             18-19: Number of transmitted FRMF frames
             20-21: Number of transmitted REJ frames"
        ::={rPOSPhysState 6}


rPFreeMemFFC OBJECT-TYPE
        SYNTAX      OCTET STRING(SIZE(6))
        MAX-ACCESS  read-only
        STATUS      current
```

```
        DESCRIPTION
            "Free memory size for FFC is a 6 byte string
               1: 0 = FFC can not be performed by this RP
                  1 = FFC can be performed by this RP
             2-5: Number of free bytes for FFC
               6: 0 = The program is stored as it is in CP
                  1 = The program is always stored uncompressed"
        ::={rPOSPhysState 7}


rPNVRamMemSize OBJECT-TYPE
        SYNTAX      INTEGER(0..2147483647)
        UNITS       "Bytes"
        MAX-ACCESS  read-only
        STATUS      current
        DESCRIPTION
            "NVRAM memory size is a four byte integer returning the complete
             size of the NVRAM"
        ::={rPOSPhysState 8}


rPFirstPermSoft OBJECT-TYPE
        SYNTAX      OCTET STRING(SIZE(41))
        MAX-ACCESS  read-only
        STATUS      current
        DESCRIPTION
            "First permanent software unit in NVRAM is a 41 byte ASCII string
              1-8: Unit name
             9-40: SUID string
               41: Property information
                  B0-6: Reserved (always 0)
                    B7: 0 = More units to retrieve
                        1 = No more information"
        ::={rPOSPhysState 9}


rPNextPermSoft OBJECT-TYPE
        SYNTAX      OCTET STRING(SIZE(41))
        MAX-ACCESS  read-only
        STATUS      current
        DESCRIPTION
            "Next permanent software unit in NVRAM is a 41 byte ASCII string
              1-8: Unit name
             9-40: SUID string
               41: Property information
                  B0-6: Reserved (always 0)
                    B7: 0 = More units to retrieve
                        1 = No more information"
        ::={rPOSPhysState 10}


rPFirstCacheSoft OBJECT-TYPE
        SYNTAX      OCTET STRING(SIZE(42))
        MAX-ACCESS  read-only
```

```
        STATUS      current
        DESCRIPTION
            "First cached software unit in filesystem is a 42 byte ASCII
             string
              1-8: Unit name
             9-40: SUID string
               41: Property information
               42: Reserved for property information (always 0)"
        ::={rPOSPhysState 11}


rPNextCacheSoft OBJECT-TYPE
        SYNTAX      OCTET STRING(SIZE(42))
        MAX-ACCESS  read-only
        STATUS      current
        DESCRIPTION
            "Next cached software unit in filesystem is a 42 byte ASCII string
              1-8: Unit name
             9-40: SUID string
               41: Property information
               42: Reserved for property information (always 0)"
        ::={rPOSPhysState 12}


rPFirstMacAddr OBJECT-TYPE
        SYNTAX      OCTET STRING(SIZE(14))
        MAX-ACCESS  read-only
        STATUS      current
        DESCRIPTION
            "First MAC-address in NVRAM is a 14 byte ASCII string
                1: Interface number
             2-13: MAC-address ASCII string. Each 2 bytes represent hex 00-FF
               14: Property information.
                  B0-6:   = 0.
                    B7: 0 = More units to retrieve
                        1 = No more information"
        ::={rPOSPhysState 13}


rPNextMacAddr OBJECT-TYPE
        SYNTAX      OCTET STRING(SIZE(14))
        MAX-ACCESS  read-only
        STATUS      current
        DESCRIPTION
            "Next MAC-address in NVRAM is a 14 byte ASCII string
                1: Interface number
             2-13: MAC-address ASCII string. Each 2 bytes represent hex 00-FF
               14: Property information.
                  B0-6:   = 0.
                    B7: 0 = More units to retrieve
                        1 = No more information"
        ::={rPOSPhysState 14}
```

```
rPProdName OBJECT-TYPE
        SYNTAX      OCTET STRING(SIZE(17))
        MAX-ACCESS  read-only
        STATUS      current
        DESCRIPTION
            "Production name and production date is a 17 byte ASCII string"
        ::={rPOSPhysState 15}


rPSerialNum OBJECT-TYPE
        SYNTAX      OCTET STRING(SIZE(13))
        MAX-ACCESS  read-only
        STATUS      current
        DESCRIPTION
            "Serial number is a 13 byte ASCII string"
        ::={rPOSPhysState 16}


rPProcClockRate OBJECT-TYPE
        SYNTAX      OCTET STRING(SIZE(4))
        MAX-ACCESS  read-only
        STATUS      current
        DESCRIPTION
            "Processor clock rate is described as a four byte string and
             expressed in ASCII notation.
             1: 100kHz figure
             2:   1MHz figure
             3:  10MHz figure
             4: 100MHz figure"
        ::={rPOSPhysState 17}


rPAddrInfoTrap NOTIFICATION-TYPE
        OBJECTS     {rPSubAddr,
                     rPBoardAddr,
                     rPStabPhysAddr,
                     rPLogicIndAddrHW,
                     rPLogicIndAddrMem,
                     rPLogicGroupAddrHW,
                     rPLogicGroupAddrMem}
        STATUS      current
        DESCRIPTION
           "The informations required when informing about address change"
        ::={rPOSMagAddr 1}

rPAddrInfo OBJECT-TYPE
        SYNTAX      INTEGER(0..1)
        MAX-ACCESS  read-write
        STATUS      current
        DESCRIPTION
            "This object is changed by the agent when the RP makes an
             address update.This object is polled by the manager. When
             the manager reads an update (==1) he must reset it to normal
             (=0) again. The manager can then get all address objects from
```

```
             the RP to find out what is wrong. This object is meant to
             prevent extensive polling from the managing side. The idea
             is to find out that some type of address change has occurred.
             no_change = 0 (normal)
             change    = 1 (unusual RP address event)"
         ::={rPOSMagAddr 2}


rPSubAddr OBJECT-TYPE
         SYNTAX      OCTET STRING(SIZE(1))
         MAX-ACCESS  read-only
         STATUS      current
         DESCRIPTION
             "Subrack address from backplane (plug) is a 1 byte string.
              This is a magazine address.
                 B4: SP
              B3-B0: SA3-SA0"
         ::={rPOSMagAddr 3}


rPBoardAddr OBJECT-TYPE
         SYNTAX      OCTET STRING(SIZE(1))
         MAX-ACCESS  read-only
         STATUS      current
         DESCRIPTION
             "Board address from backplane
                 B6: BP
              B5-B0: BA5-BA0"
         ::={rPOSMagAddr 4}


rPStabPhysAddr OBJECT-TYPE
         SYNTAX      OCTET STRING(SIZE(2))
         MAX-ACCESS  read-only
         STATUS      current
         DESCRIPTION
             "Stabphysaddr from memory is a 2 byte string
                B3-B0: Not used
                B9-B4: BA5-BA0
                  B10: BP
              B14-B11: SA3-SA0
                  B15: SP"
         ::={rPOSMagAddr 5}


rPLogicIndAddrHW OBJECT-TYPE
         SYNTAX      OCTET STRING(SIZE(1))
         MAX-ACCESS  read-only
         STATUS      current
         DESCRIPTION
             "Logical individual address from HW register is a 1 byte string
              B4-B0: Individual address
                 B5: Enable/disable bit"
```

```
        ::={rPOSMagAddr 6}


rPLogicIndAddrMem OBJECT-TYPE
        SYNTAX      OCTET STRING(SIZE(1))
        MAX-ACCESS  read-only
        STATUS      current
        DESCRIPTION
            "Logical individual address from memory is a 1 byte string
             "
        ::={rPOSMagAddr 7}


rPLogicGroupAddrHW OBJECT-TYPE
        SYNTAX      OCTET STRING(SIZE(1))
        MAX-ACCESS  read-only
        STATUS      current
        DESCRIPTION
            "Logical group address from HW register. Is not used.
             B4-B0: Group address
                B5: Enable/disable bit"
        ::={rPOSMagAddr 8}


rPLogicGroupAddrMem OBJECT-TYPE
        SYNTAX      OCTET STRING(SIZE(1))
        MAX-ACCESS  read-only
        STATUS      current
        DESCRIPTION
            "Logical group address from memory is a 1 byte string.
             Is not used either"
        ::={rPOSMagAddr 9}



rPMIBGroups OBJECT IDENTIFIER ::={rPMIB 7}


rPGlobalGroup OBJECT-GROUP
        OBJECTS     {rPGlobalChange}
        STATUS       current
        DESCRIPTION
            "This group contains RP global objects"
        ::={rPMIBGroups 1}


rPGlobalTrapGroup NOTIFICATION-GROUP
        NOTIFICATIONS {rPGlobalChangeTrap}
        STATUS      current
        DESCRIPTION
            "This group contains all global traps for the RP"
        ::={rPMIBGroups 2}
```

```
rPMBHGroup OBJECT-GROUP
        OBJECTS     {rPSlotNumber, rPBoardLEDState, rPHWIPromData,
                     rPSlotDataChange,rPPollPrinciple, rPPollInterval,
                     rPOpsHWICHANGE, rPPromMaxLen}
        STATUS       current
        DESCRIPTION
            "This group contains all objects, that cover the data exchanges
             between CP and RP for the rpmbh-block, in the RP "
        ::={rPMIBGroups 3}


rPMMGroup OBJECT-GROUP
        OBJECTS {rPOpsRPMMFANSTAT, rPOpsRPMMPOWSTAT, rPPowerStateA,
                 rPTemporaryPowFaultsNumA, rPPowerStateB,
                 rPTemporaryPowFaultsNumB, rPPowerStatus,
                 rPTemporaryPowFaultsNum, rPPowPathChange,
                 fanUnitNumberInd, fanMIALEDState, fanAirTemp,
                 fanReadResult, fanCommErrorDet, fanAlarmInfo,
                 fanMIALEDSetRes, fanUnitNumber,
                 fanDataStatus, fanHWIStatus, fanHWIReadRes,
                 fanHWIPromData, fanDataorHWIStatus}
        STATUS  current
        DESCRIPTION
            "This group contains all objects that cover the data exchanges
             between CP and RP for the rpmm-block in the RP"
        ::={rPMIBGroups 5}


rPMMTrapGroup NOTIFICATION-GROUP
        NOTIFICATIONS {fanMIALEDSetResTrap, rPPowerStatusTrap,
                       fanDataorHWIStatusTrap}
        STATUS    current
        DESCRIPTION
            "This group contains all traps for the rpmm-block in the RP"
        ::={rPMIBGroups 6}


rPFDGroup OBJECT-GROUP
        OBJECTS {rPOpsRPEMERROR, rPMaxEMNum, rPTestType, rPTestState,
                 rPEMAddress, rPEMTestState, rPCMNumber, rPEMTestOrder,
                 rPFirstTest, rPAckOrder, rPTestTime, rPEMAddress2,
                 rPEMErrorType, rPEMAlarmWord, rPErrorCode, rPErrorData,
                 rPTestOrderSupCirc, rPErrorCase}
        STATUS  current
        DESCRIPTION
            "This group contains all objects that cover the data exchanges
             between CP and RP for the rpfd-block in the RP"
        ::={rPMIBGroups 7}


rPFDTrapGroup NOTIFICATION-GROUP
        NOTIFICATIONS {rPErrorCaseTrap, rPEMErrorTrap, rPErrorCodeandDataTrap
                      }
        STATUS  current
```

```
        DESCRIPTION
            "This group contains all traps for the rpfd-block in the RP"
        ::={rPMIBGroups 8}


rPOSGroup OBJECT-GROUP
        OBJECTS {rPBootProgId, rPDramSize, rPOsProp,
                 rPPhysTypeNum, rPProcBoardId, rPStatCount,
                 rPFreeMemFFC, rPNvRamMemSize, rPFirstPermSoft,
                 rPNextPermSoft, rPFirstCacheSoft, rPNextCacheSoft,
                 rPFirstMacAddr, rPNextMacAddr, rPProdName,rPSerialNum,
                 rPProcClockRate, rPSubAddr, rPBoardAddr, rPStabPhysAddr,
                 rPLogicIndAddrHW, rPLogicIndAddrMem, rPLogicGroupAddrHW,
                 rPLogicGroupAddrMem, rPAddrInfo}
        STATUS  current
        DESCRIPTION
            "This group contains all objects that cover the data exchanges
             between CP and RP for the rp-os block in the RP"
        ::={rPMIBGroups 9}


rPOSTrapGroup NOTIFICATION-GROUP
        NOTIFICATIONS {rPAddrInfoTrap}
        STATUS      current
        DESCRIPTION
            "This group contains all traps for the rp-os block in the RP"
        ::={rPMIBGroups 10}


END
```

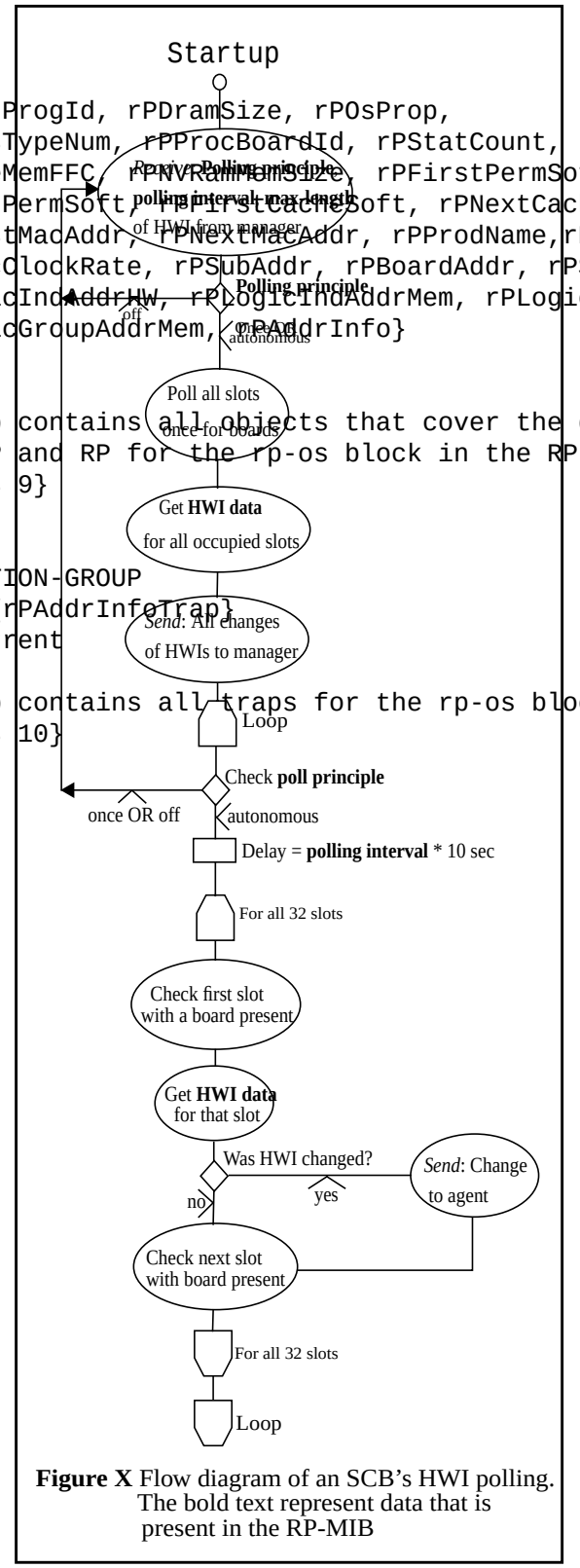

Figure X Flow diagram of an SCB's HWI polling. The bold text represent data that is present in the RP-MIB

APPENDIX B