Power and Fan Control Automated Testing

Mojgan Javadi
Abstract

Power and Fan Control Automated Testing

Mojgan Javadi

Ericsson is one of the few famous providers of telecommunication equipments and related services to mobile and fixed network operators globally. Over 1,000 networks in more than 180 countries use Ericsson network equipment and 40 percent of all mobile traffic are passing through Ericsson systems.

A new version of a unit called Power and Fan Module (PFM) is under development at Ericsson AB. It is a fan controller system and power supplier unit for the Evolved Generic Ericsson Magazine (EGEM). While the older versions of PFM has designed and developed by a third party company, due to some strategic reasons, Ericsson decided to redesign and build it internally. The new version is designed to provide the same functionality as the old one extended and improved with some new features.

This project currently is in the verification phase and a number of tests need to be automatically performed to assure that old features are still working and no new faults has been introduced as the result of adding the new features. To test this hardware a principle of the hardware testing technique called Built-In Self Test (BIST) is used. It is a cost-effective solution that makes remote hardware tests easier.

The main task of this master thesis is investigating to find an efficient and reliable way of establishing connection to a specific range of hardware (including PFM) and automatically perform built-in tests that are already embedded on that hardware to test its functionality. The ultimate goal is to develop a tool to automatically test a hardware and as the result increase efficiency and reduce the need of manual testing, money and human resources. PFM is chosen to be tested with this generic tool which is called Pbist. Moreover this tool is aimed to be integrated into an existing test framework called JUnit Common Auto Tester (JCAT). In this thesis, the design and implementation of such a tool is discussed for Windows platform. Furthermore, the solutions and methods to migrate this tool from Windows to Linux has been investigated and discussed.
CHAPTER 1 .......................................................................................................................... 1

**INTRODUCTION** .............................................................................................................. 1

1.1 BACKGROUND ............................................................................................................... 2

1.2 PROBLEM DESCRIPTION .............................................................................................. 2

1.3 AIMS AND OBJECTIVES ............................................................................................... 3

1.4 THESIS ORGANIZATION ............................................................................................... 3

CHAPTER 2 ............................................................................................................................ 4

**BACKGROUND** ..................................................................................................................... 4

2.1 PFM (POWER AND FAN MODULE) ................................................................. 4

2.2 DENIB PROTOCOL .................................................................................................... 5

2.2.1 MCL2 Protocol ..................................................................................................... 6

2.2.2 Introduce DENIB Unit Objects and CODEs ......................................................... 6

2.2.3 DENIB Message Size ......................................................................................... 8

2.2.4 DENIB Data Types ............................................................................................. 9

2.2.5 DENIB Message Format .................................................................................... 11

2.2.6 How DENIB Works ........................................................................................... 15

2.2.7 Error Messages .................................................................................................. 18

2.2.8 PFM in EGEM .................................................................................................... 20

2.3 FRAMEWORK ............................................................................................................. 21

2.3.1 JUnit Common Auto Tester (JCAT) .................................................................. 21

2.3.2 JCAT Highlights ............................................................................................... 22

CHAPTER 3 ............................................................................................................................ 23

**PBIST TEST AUTOMATION** .............................................................................................. 23

3.1 BUILT-IN SELF TEST (BIST) ................................................................................... 23

3.1.1 BIST Architecture ............................................................................................. 24

3.2 PRODUCT BUILT-IN SELF TEST (PBIST) ............................................................. 25

3.2.1 PBIST Related CODEs .................................................................................... 26

3.2.2 PBIST Structure ............................................................................................... 26

3.3 RUNNING PBIST TESTS ............................................................................................ 28

3.3.1 Serial Port ........................................................................................................ 29

3.3.2 Test Configuration File .................................................................................... 29

3.3.3 Activate Pbist Controller ................................................................................ 31

3.3.4 Execute a Pbist Test (ONBR>0) ..................................................................... 31

3.3.5 Checking the Output ....................................................................................... 31

3.4 INTERACTION WITH PFM ....................................................................................... 32

3.5 HOW TO TEST A SUT WITH JCAT ........................................................................ 34

3.5.1 JCAT Framework ............................................................................................ 34

3.5.2 JCAT SUT Specific Extension ....................................................................... 34

3.5.3 JCAT Testcases and Test Suites ..................................................................... 35

3.5.4 JCAT Structure ............................................................................................... 35

CHAPTER 4 ............................................................................................................................ 36

**IMPLEMENTATION** .......................................................................................................... 36

4.1 UML .............................................................................................................................. 36

4.1.1 SerialPortCommunication Class ..................................................................... 39

4.1.2 Denib Class ...................................................................................................... 39

4.1.3 Testcase and Testcases Class ......................................................................... 40

4.1.4 ReadXML Class .............................................................................................. 40
4.1.5 PbistMessage Class ........................................................................................................... 40
4.1.6 ReadBuffer Class ............................................................................................................. 40
4.1.7 Pfm Class ......................................................................................................................... 40
4.2 USER INTERFACE .................................................................................................................. 41
4.3 HOW TO TEST PFM WITH JCAT ....................................................................................... 43
  4.3.1 Running PfmVoltageMonitoringTestCase ......................................................................... 46

CHAPTER 5 .................................................................................................................................. 48

CONCLUSION ............................................................................................................................. 48

BIBLIOGRAPHY ........................................................................................................................ 50

APPENDIX A ............................................................................................................................ 52
LIST OF FIGURES

Figure 1: Power and Fan Module ................................................................. 5
Figure 2: Data Encapsulation ................................................................. 6
Figure 3: DENIB message format ............................................................. 11
Figure 4: Read CONTR Object’s Name ....................................................... 15
Figure 5: Read an Object of type OTYP_4STCTL ........................................ 16
Figure 6: Read a NVSTR object (ProdIndivData) ......................................... 17
Figure 7: A Correct Message ................................................................. 18
Figure 8: A Wrong Message ................................................................ 18
Figure 9: Error message ...................................................................... 18
Figure 10: Chain Structure Sub-rack ......................................................... 20
Figure 11: JCAT Layered Architecture ...................................................... 21
Figure 12: JCAT HTML Report ............................................................... 22
Figure 13: BIST Method Structure ............................................................ 25
Figure 14: PFM Phist structure ................................................................. 27
Figure 15: Test Configuration File ............................................................ 30
Figure 16: JCAT Structure .................................................................. 35
Figure 17: Phist UML Diagram ............................................................... 38
Figure 18: PBIST Tool ......................................................................... 42
Figure 19: Example of a log4j file ............................................................ 47
Figure 20: jSviftTerm ......................................................................... 52
Figure 21: SVIFT Editor ....................................................................... 53
Figure 22: JSVIFT Tree ....................................................................... 54
LIST OF TABLES

Table 1: PFM-LOD .................................................................................................................. 8
Table 2: EBYTE Data Type .................................................................................................. 9
Table 3: DENIB Data Type .................................................................................................. 10
Table 4: HFLG Fields ......................................................................................................... 12
Table 5: DMOD, DADR, SMOD, SADR ................................................................................. 12
Table 6: DUnit_Obj Types .................................................................................................. 13
Table 7: CODEs .................................................................................................................... 14
Table 8: DENIB Message Format ........................................................................................ 14
Table 9: RCODE .................................................................................................................. 19
Table 10: Error Message ..................................................................................................... 19
Chapter 1

Introduction

Testing is a critical part of any design and development process. Many companies allocate considerable development time on testing. They have realized the need of spending considerable time and money to deliver a high quality product. In this situation, a powerful competitor has to present a product with the highest quality and less defects to conquer over the other rivals. This requires testing the product from the very early steps of the development until even after it has finished [1].

The standard testing procedure starts once the product requirements specification has been established and lasts until the project is handed to the customers. This procedure has various steps with diverse rules and tools. The number of tests is growing gradually during testing process and most of the time it is not even possible to run some tests manually. Running tests manually require a lot of money and human resources. Therefore testing is expensive and labor intensive. It requires up to 50% of development process costs, and even more for safety-critical applications. As a result testing must be automated as much as possible, thereby significantly minimizing its cost and reducing human errors. The automated tests are reliable, faster, reusable and repeatable in order to [1]:

- Run more tests more often: In an automated environment it is most likely possible to run the same tests in shorter time compared to manual testing. Automation makes it easy to iterate tests running over and over again with less human efforts.

- Run tests, difficult to perform by human: Some testing activities are too hard for a human to perform such as stress testing, testing over nights with repetitive tasks.
Apply resources more efficiently: By removing fruitless and boring tasks from the manual tester tasks, they can be used in more efficient work such as improving test cases, debug other faults found in the product and so on.

Reuse of tests: The benefit of reusing automated tests, if it is possible, is also great. It reduces the cost of employed resources such as human, time and money.

As the result of the discussion above, we conclude that testing is crucial for a product and automation is crucial for a testing process.

In this thesis, we focus on investigating and seeking the best way to automate a series of tests for a specific range of hardware at Ericsson. One such a hardware is the PFM (Power and Fan Module) which is currently under development and in verification phase. Thereby it is used as an instance in this project.

1.1 Background

PFM (Power and Fan Module) is a control unit that is aimed for cooling and supplying power to the racks in the Ericsson’s labs. Currently a new version of PFM is under development in Ericsson as a result of some modifications in its hardware and software. Before PFM is delivered to the labs, it must be put through several tests to make sure that the hardware behaves correctly and is on satisfying quality [2].

The method used to test this hardware is called Pbist (Product Built-In Self Test). It is a software tool that is used to run a series of test cases automatically on the PFM. This method is inspired by the BIST (Built-In Self Test) technique which is used to test complex integrated circuits. But Pbist tool is a generic software tool and, as we discussed earlier, PFM is only a sample to evaluate this tool. It is going to be used for several specific hardware units in the future; they are called DENIB-related hardware units. This name is chosen because the proposed communication protocol with these hardware units is called DENIB protocol. In later chapters we will discuss and exemplify the usage of DENIB protocol to communicate with PFM [3, 4].

1.2 Problem Description

As mentioned above there are a series of hardware units at Ericsson that need to be tested. They are called DENIB-related hardware units since the DENIB interface is used as the way of interacting and connecting to the hardware. A generic tool, which automatically performs Pbist tests on these units, must be designed. It should support protocol verification, fan control verification, power control verification, alarm verification (by inducing faulty conditions) etc. This tool is going to be integrated within the present Test Framework called JCAT. It should be
approved as preferred global tool for Test Automation Framework. JCAT aims to unite some of the small automation islands within Ericsson to provide a collaboratively developed solution.

In this thesis, we strive to design and develop a suitable generic automatic tool to test the DENIB-related hardware including PFM. In this project, PFM is used as an instance of DENIB-related hardware. The tool must be applicable in the real environment and could be used for the futures needs.

1.3 Aims and Objectives

The objective of this thesis is to find a reasonable way to automatically run tests on DENIB-related hardware units. To achieve this, an automated tool should be designed and developed. The expected outcome is to:

- reduce manual testing,
- increase test coverage,
- provide a better overview of test results, and
- release products of higher quality, as a consequence of the above beneficial points.

However implementing and verifying individual test cases will be outside the scope of this project.

1.4 Thesis Organization

The rest of the report is organized as follows:

Chapter 2 explains terms and technologies that were used in this project.

Chapter 3 describes the theories and researches used to find the efficient and suitable way to implement the automated test tool for DENIB-related hardware.

Chapter 4 discusses the implementation of the tool and the material we used to achieve the goal.

Chapter 5 contains the conclusion of the project and discusses possible future works that can be carried out as a complementary to this project.
Chapter 2

Background

This chapter covers the background terminology and technologies that are used along this project.

2.1 PFM (Power and Fan Module)

What is PFM? This hardware is used in equipment racks at Ericsson’s labs. It is aimed for cooling and supplying power to the racks in the Ericsson’s labs. It is a multi-purpose unit that vents and thus cools residential systems that are installed in a standard rack EGEM (Evolved Generic Ericsson Magazine). It also supplies power by distributing it through the whole sub-rack via the backplane. A microcontroller is used in this hardware as its brain. It is employed to control and test the functionality of the hardware, e.g., controls the fan motor voltage by using a temperature sensor. By updating the firmware of the microcontroller, new features can be added to PFM in order to extend its capabilities. It is composed of a power board and three high capacity fans. Normally, two PFMs are used in the racks and one of them is for redundancy [2, 5]. PFM is shown in Figure 1.
2.2 DENIB Protocol

The DENIB protocol is aimed to be used for master/slave communication. The protocol is message based; a master sends a request causing a response message from the slave. These messages are called DENIB messages [4, 6].

The general features of this protocol are as follows:

- **Compressed Self Expanding header:** This allows the protocol to be flexible for various applications and yet efficient. Most fields of the header are minimum three bits, but expand automatically as their values require. The minimum header overhead is 5 bytes.

- **Use of general standardized objects:** These objects define the types of requests and responses that is valid and the formats of associated data of the message, if any. The objects are generalized to serve more than one purpose. The idea of this concept is to reduce product documentation (by reference to the objects) and enable reuse of software code (as the idea of protocol itself).

- **One header field is reserved for future expansions of the protocol:** Possible such expansions might be new addressing modes or new header fields.

DENIB messages require some form of packaging into the frame and it is carried out by MCL2 protocol. This is achieved by adding a header and checksum to the raw DENIB message. The purpose of framing DENIB message as you see in Figure 2 is to separate messages from each other.
2.2.1 MCL2 Protocol

A message-oriented data is encapsulated by MCL2 protocol into a frame. These frames always use a header field at the beginning, and a checksum byte at the end of each message. MCL2 provides the means for transmission of messages of MCL2_Protocol type between equipments. One of such protocols is DENIB. The maximum size of the frame is 73 bytes [6].

The encapsulation is illustrated by the following figure:

![Figure 2: Data Encapsulation](image)

2.2.2 Introduce DENIB Unit Objects and CODEs

DENIB is used as a way of interaction with several pieces of hardware at Ericsson. These hardware are composed of a number of internal standardized objects which are called “D_unitObj” objects. They supervised and/or controlled via DENIB protocol. But how can DENIB control these objects?

These objects are controlled by using their identified CODE values and data fields, each object has its associated CODE value and data field. These concepts are coherently explained in the example below.

**Example:** The following table is detailed description of controller object and its associated CODE values and data fields. This is only one instance to show what an object is and what CODE values are. If you are interested to know all objects, which are controlled by DENIB protocol in different products refer to appendix A [7].
**D_UnitObj:** The microcontroller object of PFM is a type of CONTR, which is called PFM-LOD. The controller object supports CODE read, Info, Echo, Name, and etc. Look at the following CODEs in detail as an example.

---

**CODE_Name:**

Function: Read CONTR instance name:

If this CODE is used, the name of the hardware unit’s microcontroller is asked.

**DATA:**

- Request
  - No data
- Response
  - STRING (INSTNAME): The name of this CONTR object instance

---

**CODE_ExtInfo:**

Function: Read controller extra information

If this CODE is used, it specifies a request to get the information, which is hold by hardware unit’s microcontroller.

**DATA:**

- Request
  - No data
- Response
  - A 16-byte sequence of binary data, typically containing HW and FW identification values.
The different objects of PFM are indicated in Table 1. Note that each object has a unique name e.g. the controller object of type CONTR is named “PFM_LOD”, the object of FININFO2 type is named “fininfo”. As you might notice different names could be pointed to the same D_UnitObj type but to different object number of the same type. For example see the first D_UnitObj object of type Rosan (ONBR=0) is AirTemp, while the second object (ONBR = 1) of the same type is FanSpeed1.

<table>
<thead>
<tr>
<th>D_UNITOBJ NAME</th>
<th>D_UNITOBJ TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: PFM-LOD</td>
<td>CONTR</td>
</tr>
<tr>
<td>1-1: fininfo</td>
<td>FININFO2</td>
</tr>
<tr>
<td>1-2: FFLAGS</td>
<td>ROFLB</td>
</tr>
<tr>
<td>1-2-1: IFLAGS</td>
<td>ROFLB</td>
</tr>
<tr>
<td>1-3: conf</td>
<td>CONF</td>
</tr>
<tr>
<td>1-4: ProdIndivData</td>
<td>NVSTR</td>
</tr>
<tr>
<td>1-4-1: Boot-CXC</td>
<td>NVSTR</td>
</tr>
<tr>
<td>1-4-2: Boot-Rev</td>
<td>NVSTR</td>
</tr>
<tr>
<td>1-4-3: Boot-CXC</td>
<td>NVSTR</td>
</tr>
<tr>
<td>1-4-4: FW-Rev</td>
<td>NVSTR</td>
</tr>
<tr>
<td>1-5: MIA</td>
<td>4STCTL</td>
</tr>
<tr>
<td>1-6: AirTemp</td>
<td>ROSAN</td>
</tr>
<tr>
<td>1-6-1: FanSpeed1</td>
<td>ROSAN</td>
</tr>
<tr>
<td>1-6-2: FanSpeed2</td>
<td>ROSAN</td>
</tr>
<tr>
<td>1-6-3: FanSpeed3</td>
<td>ROSAN</td>
</tr>
<tr>
<td>1-6-4: MON-M48-A</td>
<td>ROSAN</td>
</tr>
<tr>
<td>1-6-5: MON-M48-B</td>
<td>ROSAN</td>
</tr>
<tr>
<td>1-6-6: MON-M48-C</td>
<td>ROSAN</td>
</tr>
<tr>
<td>1-6-7: MON-M48-D</td>
<td>ROSAN</td>
</tr>
<tr>
<td>1-6-8: MON-M48-OUT</td>
<td>ROSAN</td>
</tr>
<tr>
<td>1-7: FanFW</td>
<td>FWHAND</td>
</tr>
<tr>
<td>1-8: Stat</td>
<td>STATSAV</td>
</tr>
<tr>
<td>1-9: EWD</td>
<td>Pbist</td>
</tr>
</tbody>
</table>

Table 1: PFM-LOD

2.2.3 DENIB Message Size

The maximum possible DENIB message size is 40 bytes and the minimum DENIB message size is 3 bytes. For the expansion/compression data size, DENIB uses expansion bits, it is described in the next subsection [7].
2.2.4 DENIB Data Types

DENIB uses two special data types, EBYTE and DENIB data type. They are explained in details in continue [6, 7].

**EBYTE Data Type:** The minimum size of an EBYTE data type is one byte. The lower 7 bits (A0…..A6) presents a binary format of parameter A. If the number of bits exceeds over 7 bits (i.e. unsigned values above 127), then A_E1, the expansion indicator bit is set to 1(A_E1 = 1) and the new byte is inserted immediately. It represents 6 more bits (A7…..A13) to indicate the parameter A and also includes a new expansion indicator bit A_E2 for the future expansion.

<table>
<thead>
<tr>
<th>BYTE</th>
<th>MSB</th>
<th>BIT 6</th>
<th>BIT 5</th>
<th>BIT 4</th>
<th>BIT 3</th>
<th>BIT 2</th>
<th>BIT 1</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A_E1</td>
<td>A6</td>
<td>A5</td>
<td>A4</td>
<td>A3</td>
<td>A2</td>
<td>A1</td>
<td>A0</td>
</tr>
<tr>
<td>2</td>
<td>A_E2</td>
<td>A13</td>
<td>A12</td>
<td>A11</td>
<td>A10</td>
<td>A9</td>
<td>A8</td>
<td>A7</td>
</tr>
<tr>
<td>3</td>
<td>A_E3</td>
<td>A20</td>
<td>A19</td>
<td>A18</td>
<td>A17</td>
<td>A16</td>
<td>A15</td>
<td>A14</td>
</tr>
</tbody>
</table>

*Table 2: EBYTE Data Type*

As it is illustrated in Table 2, the numbers in the Byte column indicates the order of sending bytes over the interface. First, byte 1 is sent and if more bytes are needed to present the parameter A then the byte related expansion indicator is set to 1 and the next byte is added immediately.

Take into consideration that the insertion of the new byte only depends on the value of parameter A. If its value is not big, one byte may be enough otherwise it has the ability to expand to more bytes if it is required. Therefore, when no more bytes are required, the expansion indicator bit of the last byte is set to zero (=0).

**Example1:** EBYTE data type is used to present the parameter A (65)

65 = 2^6 + 2^0 (A’s values is presented by one byte)

<table>
<thead>
<tr>
<th>BYTE 1</th>
<th>bit 7 (MSB)</th>
<th>bit 6</th>
<th>bit 5</th>
<th>bit 4</th>
<th>bit 3</th>
<th>bit 2</th>
<th>bit 1</th>
<th>bit 0 (LSB)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A_E1=0</td>
<td>A6=1</td>
<td>A5=0</td>
<td>A4=0</td>
<td>A3=0</td>
<td>A2=0</td>
<td>A1=0</td>
<td>A0 =1</td>
</tr>
<tr>
<td>2^6</td>
<td>2^5</td>
<td>2^4</td>
<td>2^3</td>
<td>2^2</td>
<td>2^1</td>
<td>2^0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

BYTE1 shows the value of parameter A. It doesn’t exceed 127, the maximum value. Thus it is fitted in one EBYTE.
If the value of parameter A exceeds 127 then the expansion bit sets to 1 otherwise it is 0 as you see in the example-

**Example 2:** The EBYTE data type, the representation of parameter A (165)

$$160 = 2^7 + 2^5 + 2^0$$ (A’s value is presented by two bytes)

<table>
<thead>
<tr>
<th>BYTE 1</th>
<th>bit 7 (MSB)</th>
<th>bit 6</th>
<th>bit 5</th>
<th>bit 4</th>
<th>bit 3</th>
<th>bit 2</th>
<th>bit 1</th>
<th>bit 0 (LSB)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A_E1=1</td>
<td>A6=0</td>
<td>A5=1</td>
<td>A4=0</td>
<td>A3=0</td>
<td>A2=0</td>
<td>A1=0</td>
<td>A0 =0</td>
</tr>
<tr>
<td>Expande</td>
<td>2^7</td>
<td>2^5</td>
<td>2^4</td>
<td>2^3</td>
<td>2^2</td>
<td>2^1</td>
<td>2^0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BYTE 2</th>
<th>bit 7 (MSB)</th>
<th>bit 6</th>
<th>bit 5</th>
<th>bit 4</th>
<th>bit 3</th>
<th>bit 2</th>
<th>bit 1</th>
<th>bit 0 (LSB)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A_E2=0</td>
<td>A13=0</td>
<td>A12=0</td>
<td>A11=0</td>
<td>A10=0</td>
<td>A9=0</td>
<td>A8=0</td>
<td>A7 =1</td>
</tr>
<tr>
<td></td>
<td>2^13</td>
<td>2^12</td>
<td>2^11</td>
<td>2^10</td>
<td>2^9</td>
<td>2^8</td>
<td>2^7</td>
<td></td>
</tr>
</tbody>
</table>

In this example the value of parameter A is above 127 and it doesn’t fit in one EBYTE. Hence the expansion bit E_A1 is set to 1, which means one more byte must be added immediately. The value, 165 now stored in two EBYTES.

**DENIB Data Type.** The minimum size of a DENIB (Double Expanding Nibble) is also one byte. Each byte however used for representation of two values. Writing DENIB (A: B) is short for:

<table>
<thead>
<tr>
<th>BYTE</th>
<th>MSB</th>
<th>BIT 6</th>
<th>BIT 5</th>
<th>BIT 4</th>
<th>BIT 3</th>
<th>BIT 2</th>
<th>BIT 1</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A_E1</td>
<td>A2</td>
<td>A1</td>
<td>A0</td>
<td>B_E1</td>
<td>B2</td>
<td>B1</td>
<td>B0</td>
</tr>
<tr>
<td>2</td>
<td>EBYTE (A9...A3)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>EBYTE (B9...B4)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 3: DENIB Data Type**

where A0, A1, A2 are the three lower order bits of A (A0 identifies least significance bit, higher numbers indicate more significant bits) and B0, B1, B2 have the corresponding meaning for the bits of B. Byte numbers 2 and 3 are extension bytes, that are only present if the values of A and B can not fit in 3 bits of byte 1(i.e. their binary value exceeds 7). The extension bytes are inserted separately for A and B. Table 3 shows the order of inserting extra bytes for A and B, if both are present. The presence of an extension byte for A and/or B is indicated once the expansion indicator bits, A_E1 (A extended) and B_E1 (B is extended) are set. If any of these bits are zero,
the corresponding extension byte is not present. Byte numbering shows the order in which, bytes are sent over the interface.

**Example 1**: DENIB data type representation of (45:60)

<table>
<thead>
<tr>
<th>BYTE 1</th>
<th>bit 7 (MSB)</th>
<th>bit 6</th>
<th>bit 5</th>
<th>bit 4</th>
<th>bit 3</th>
<th>bit 2</th>
<th>bit 1</th>
<th>bit 0 (LSB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>B_E1= 1</td>
<td>B2=1</td>
<td>B1=0</td>
<td>B0=0</td>
<td>A_E1=1</td>
<td>A2=1</td>
<td>A1=0</td>
<td>A0 =1</td>
<td></td>
</tr>
<tr>
<td>expanded</td>
<td>2^7</td>
<td>2^6</td>
<td>2^5</td>
<td>expanded</td>
<td>2^4</td>
<td>2^3</td>
<td>2^2</td>
<td>2^1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BYTE 2</th>
<th>A_E2=0</th>
<th>A9=0</th>
<th>A8=0</th>
<th>A7=0</th>
<th>A6=0</th>
<th>A5=1</th>
<th>A4=0</th>
<th>A3 =1</th>
</tr>
</thead>
<tbody>
<tr>
<td>2^9</td>
<td>2^8</td>
<td>2^7</td>
<td>2^6</td>
<td>2^5</td>
<td>2^4</td>
<td>2^3</td>
<td>2^2</td>
<td>2^1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BYTE 3</th>
<th>B_E2=0</th>
<th>B9=0</th>
<th>B8=0</th>
<th>B7=0</th>
<th>B6=0</th>
<th>B5=1</th>
<th>B4=1</th>
<th>B3 =1</th>
</tr>
</thead>
<tbody>
<tr>
<td>2^9</td>
<td>2^8</td>
<td>2^7</td>
<td>2^6</td>
<td>2^5</td>
<td>2^4</td>
<td>2^3</td>
<td>2^2</td>
<td>2^1</td>
</tr>
</tbody>
</table>

In this example, as you see, the value of parameter A is 45, therefore it does not fit in the first three bites. So its associated expansion bit, A_E1 is set to 1 and one more byte is added to store the value of parameter A. The same process also is applied for the value of parameter B.

### 2.2.5 DENIB Message Format

Figure 3 shows an overview of the DENIB message format. In the figure, each BYTEFIELD represents a byte of the message. The order of the BYTEFIELDS are as follows; HPNR, HFLG, DMOD, DADR, SMOD, SADR, OTYP, ONBR, CODE, SQNR, DATA, ECHK. A complete description of the message fields is shown in Table 4. In this section, the byte fields of the message are explained in detail [4, 7].

![Figure 3: DENIB message format](image)
1. BYTEFIELD1 (HPNR: HFLG): HFLG is used to show which optional fields are presented in the header.

<table>
<thead>
<tr>
<th>HEX DECIMAL VALUE</th>
<th>NAME</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x01</td>
<td>HFLG_SQNR</td>
<td>If this bit is then a SQNR field presents in the header otherwise not.</td>
</tr>
<tr>
<td>0x02</td>
<td>HFLG_ECHK</td>
<td>If this bit sets, a ECHK field presents immediately after the message data, else not. The presence of the ECHK field forces the receiver to test it for validity. The use is optional.</td>
</tr>
<tr>
<td>0x04</td>
<td>HFLG_REQU</td>
<td>If this bit is set, the message is a request. Otherwise it is a response.</td>
</tr>
</tbody>
</table>

Table 4: HFLG Fields

The massage issuer specifies the value of HFLG. Only value 1 is currently allowed for the HPNR field. This field is aimed for the future extension of the protocol. Note that the value of HFLG and HPNR is always kept unchanged from requests to responses.

2. BYTEFEILD2 and BYTEFEILD3 (DMOD, DADR, SMOD, SADR): The Destination, Source modes and addresses of D_UnitObj objects. The entire DENIB (DMOD: DADR) and DENIB (SMOD: SADR) fields swapped in a possible response. This means that the DENIB (DMOD: DADR) is turned to the source and the DENIB (SMOD: SADR) is turned to the destination. Special values for DMOD and SMOD are shown in Table 5.

<table>
<thead>
<tr>
<th>HEXDECIMAL VALUES</th>
<th>NAME</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>AMOD_PHYS</td>
<td>Physical address mode. Address matches if DADR=D_UnitAdr. The request message is terminated and replied to by the unit having a match address.</td>
</tr>
<tr>
<td>0x01</td>
<td>AMOD_BCST</td>
<td>Broadcast address mode. The specified address matches all units. So the message replied by all units, and never terminated. The corresponding address field value must be zero and may never be used as SMOD. Possible responses always use SMOD=AMOD_PHYS.</td>
</tr>
<tr>
<td>0x02</td>
<td>AMOD_RELP</td>
<td>Relative physical address mode. When used as DEMOD, The DADR is first decremented, if it then becomes zero, the address matches, and the unit will terminate and then respond to the message. When use as SMOD, the SADR value is incremented. The incrimination or decriminiation before the content of the message is further evaluated.</td>
</tr>
<tr>
<td>0x03</td>
<td>AMOD_RELB</td>
<td>Relative Broadcast Mode. When used as DMOD, the DADR value is first decremented, if it then becomes zero, the message is terminated. The message is responded to all units. May never used as SMOD. Possible responses use SMOD=AMOD_RELP.</td>
</tr>
</tbody>
</table>

Table 5: DMOD, DADR, SMOD, SADR
1. BYTEFIELD4: All D_unitObj, their corresponding CODE values and an overview of their functionality are listed in the Table 6.

<table>
<thead>
<tr>
<th>HEX VALUE</th>
<th>NAME</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>OTYP_CONTR</td>
<td>The D_Iface controller itself. This object handles parameters that are common to all object types or no object type in particular.</td>
</tr>
<tr>
<td>0x01</td>
<td>OTYP_NVSTR</td>
<td>A Read/Write non-volatile string of bytes.</td>
</tr>
<tr>
<td>0x02</td>
<td>OTYP_EVFLB</td>
<td>A byte of event flags.</td>
</tr>
<tr>
<td>0x03</td>
<td>OTYP_ROFLB</td>
<td>A read only flag byte.</td>
</tr>
<tr>
<td>0x04</td>
<td>OTYP_4STCTL</td>
<td>A four state control.</td>
</tr>
<tr>
<td>0x05</td>
<td>OTYP_ROSAN</td>
<td>A read only signed analogue value, using scale factor.</td>
</tr>
<tr>
<td>0x06</td>
<td>OTYP_8ROSBN</td>
<td>A read only 8 bit signed binary value.</td>
</tr>
<tr>
<td>0x07</td>
<td>OTYP_NSTCTL</td>
<td>A general state control (any number of states)</td>
</tr>
<tr>
<td>0x08</td>
<td>OTYP_GROUP</td>
<td>A grouping of other objects</td>
</tr>
<tr>
<td>0x09</td>
<td>OTYP_OUTB</td>
<td>A byte of output bits</td>
</tr>
<tr>
<td>0x0b</td>
<td>OTYP_FWHAND</td>
<td>A handler for firmware upgrades</td>
</tr>
<tr>
<td>0x0c</td>
<td>OTYP_STATSAV</td>
<td>A handler for saving of object status</td>
</tr>
<tr>
<td>0x7f</td>
<td>OTYP_CONF</td>
<td>A configuration object.</td>
</tr>
<tr>
<td>0x80</td>
<td>OTYP_FININFO1</td>
<td>A fan controller general info object</td>
</tr>
<tr>
<td>0x81</td>
<td>OTYP_FININFO2</td>
<td>A fan controller object for general override</td>
</tr>
</tbody>
</table>

**Table 6: DUnit_Obj Types**

4. BYTEFEILD5 (ONBR): As mentioned above, OTYP field identifies the type of the objects. Possibly there are several objects of one type with different names. The field ONBR specifies the object number of a specific type. Objects are usually numbered contagiously from zero upwards. For example if we have three NVSTR objects, the numbers are given to the objects is zero, one, two respectively.

5. BYTEFEILD5 (CODE): CODEs specify the requests. For example, if the CODE field of the request message is CODE_Name and the OTYP field is specified as CONTR type, then it is a request to get the name of the controller object of the System Under Test (SUT). Table 7 shows the different CODEs for different requests.

<table>
<thead>
<tr>
<th>HEX DECIMAL VALUES</th>
<th>NAME</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>CODE_Read</td>
<td>Read a D_UnitObj.</td>
</tr>
<tr>
<td>0x01</td>
<td>CODE_Write</td>
<td>Write to a D_UnitObj.</td>
</tr>
<tr>
<td>0x02</td>
<td>Code_Start</td>
<td>Start or enable a D_UnitObj function.</td>
</tr>
<tr>
<td>0x03</td>
<td>CODE_Stop</td>
<td>Stop or disable a D_UnitObj function.</td>
</tr>
<tr>
<td>0x04</td>
<td>CODE_Name</td>
<td>Get the name of the D_UnitObj.</td>
</tr>
<tr>
<td>0x06</td>
<td>CODE_Err</td>
<td>Error. If the message is a response, it indicates that a request was not carried out because of the errors. The contents of this message indicates the kind of error.</td>
</tr>
<tr>
<td>0x07</td>
<td>CODE_Echo</td>
<td>Echo ack the message data field.</td>
</tr>
</tbody>
</table>

13
<table>
<thead>
<tr>
<th>DEC</th>
<th>CODE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x09</td>
<td>CODE_Info</td>
<td>The message is for information.</td>
</tr>
<tr>
<td>0x0a</td>
<td>CODE_Clear</td>
<td>Clear a D_UnitObj function.</td>
</tr>
<tr>
<td>0x0b</td>
<td>CODE_Set</td>
<td>Set a D_UnitObj function.</td>
</tr>
<tr>
<td>0x0c</td>
<td>CODE_Extinfo</td>
<td>A common use is to define object variations such as read-only or write-only.</td>
</tr>
</tbody>
</table>

Table 7: CODEs

6. BYTEFEILD6 (SQNR): This field is optional. It is present if and only if the HFLG_SQNR flag/bit is set. If used in a request, the response must/will contain the same value. The intended use is for a message issuer to handle separation of concurrent request/response.

7. BYTEFEILD7 (DATA): The format of the data field for most messages is described together with the description of the object types. See example 3 in the next section.

8. BYTEFEILD8 (ECHK): If HFLG_ECHK is set in the first byte, an ECHK field is present immediately after the message data, else not. The presence of the ECHK field imposes receiver to test it. The use is optional.

Table 8 depicts the summary of the DENIB message format. The message fields are sent in order from top to bottom. All bytes down to the DATA field are referred to as the DENIB message header.

<table>
<thead>
<tr>
<th>DATA</th>
<th>PRESENCE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>DENIB(HFLG:HPNR)</td>
<td>Mandatory</td>
<td>HFLG: Header Flags. Descriptor of the header format. Indicates which of the optional fields are present. HPNR: Header Protocol number. Used for future expansions of the protocol. For this version this value is always 1.</td>
</tr>
<tr>
<td>DENIB(DMOD:DADR)</td>
<td>Mandatory</td>
<td>DMOD: Destination address mode. DADR: Destination Address</td>
</tr>
<tr>
<td>DENIB(SMOD:SADR)</td>
<td>Mandatory</td>
<td>SMOD: Source address Mode SADR: Destination Address</td>
</tr>
<tr>
<td>OTYP(one octet/Byte)</td>
<td>Mandatory</td>
<td>OTYP: Determines which objects type this command/request/response refers to.</td>
</tr>
<tr>
<td>DENIB(ONBR:CODE)</td>
<td>Mandatory</td>
<td>ONBR: Object Number. Distinguishes between more than one object of the OTYP type. Numbering is always contiguous, starting at zero. CODE: Used as command code for requests as well response code.</td>
</tr>
<tr>
<td>EBYTE(SQNR)</td>
<td>Optional</td>
<td>SQNR: Sequence number of a message.</td>
</tr>
<tr>
<td>Data Field</td>
<td>Determined By CODE and OTYP</td>
<td>The content of DATA field is described in example 3.</td>
</tr>
<tr>
<td>ECHK(one octet/Byte)</td>
<td>Optional</td>
<td>Extra Checksum. Sum of all bytes of the DENIB message, except the ECHK byte itself truncated to 8 bits.</td>
</tr>
</tbody>
</table>

Table 8: DENIB Message Format
2.2.6 How DENIB Works

In this section the creation of DENIB message is illustrated step by step. Follow the examples in continue to see the structure of the sent request and received response [4, 6].

Example 1: Read the name of the Controller object from PFM

Note: The DENIB message header, from first byte down to the data field, remains the same in the response string.

- The first byte value shows the message header which is specified by MCL2 protocol.
- Second byte with value “1” shows that the message is a DENIB message.
- HFLG is set to 0x04, it indicates that this message is a request and HPNR is set to 0x01, as it should be always 1.
- The destination and source mode are set to 0x02, which shows AMOD_RELP mode is chosen. For the PFM case the relative addressing is used.\(^1\)
- OTYP is chosen 0x00 since the object type is CONTR.

Relative Address: It is an integer, which indicates the distance from the beginning point of an object or any other data structure up to the specified element, by assuming that all elements of the object are the same in all aspects. For example, In array A composed by elements “a,b,c” the relative addresses of elements are a equals 0, b equals 1, and c equals 2, respectively.
- ONBR is 0x00. It shows the first object, out of possibly many CONTR objects. The CODE value is 0x06, it is a request to read the name of the object of type CONTR.
- The last byte, specified by MCL2 protocol is the checksum.

As it is seen in Figure 5, the actual response is placed after the DENIB header. By converting these hex values to char the name of the Controller object is achieved. For example in this figure the controller object is PFM-LOD.

**Example 2:** Read an Object of type OTYP_4STCTL

![Diagram of Sent Request and Received Response](image_url)

**Figure 5: Read an Object of type OTYP_4STCTL**

**Note:** The DENIB message header, from first byte down to the data field, remains same in the response string.

The 4STCTCL object type represents a four state indicator of any kind. For example, PFM have a few number of LEDs, each LED has four different states including fast, slow, on and off. Figure 6 illustrate the request to read the present state of the object of type 4STCTCL. In this example a request is sent to the object of type 4STCTCL on PFM which is an LED to get its present state. The possible responses are 0, 1, 2 or 3 which says if the LED is off, slow, fast or on respectively. As it is seen in the figure, the state is 0, which means that the LED is off.
**Example 3:** Read an NVSTR object (ProdIndivData)

This example is a bit different compared to the previous ones. The NVSTR object type represents a size TOTSIZ non-volatile string, typically stored in EEPROM, which can be read and written. It is a long string containing the product name, revision, serial number and some other general information about the product. To get this information about the product a read request must be sent to the object called ProdIndivData. To read this string, two values are needed to be placed in the DATA part, in the request message. They are called STARTP and NUM. STARTP is the starting byte position, range 0…TOTSIZ-1. The error Err_BadRange is returned if STARTP is outside of this range. NUM is the number of bytes to read. ProdIndivData is a long string and depend on the NUM (the number of bytes to read); breaks to some sequences of data. Note that in this example the STARTP is 0 and the number of bytes to read in each request is specified for 16. The request in each sequence and the relevant response are shown in the Figure 7.

![Figure 6: Read a NVSTR object (ProdIndivData)](image-url)
2.2.7 Error Messages

An error message is sent as a response to the incoming request in the case of an error. For example, consider a hardware unit with one controller. To get the name of its controller object, a request with the code CODE_Name is sent to the object of type CONTR. Notice that the request must be sent to object 0, because there exists only one microcontroller in this unit. The correct request is shown in Figure 7 [7].

If a request is sent to get the name of the object 3 of the same type, then as it is depicted in Figure 8, an error message is received. Since there is only one microcontroller object with ONBR = 0. The reasons of the failure are shown in detail in Figure 9.

---

**Figure 7: A Correct Message**

| Sent: e7 01 41 21 20 00 06 5f |
| Received: eb 01 01 21 20 00 06 50 46 4d 0 e8 |

**Figure 8: A Wrong Message**

| Sent: e7 01 41 21 20 00 36 5f |
| Received: e9 01 01 21 20 00 37 06 11 85 |

**Figure 9: Error message**
What is RCODE?

<table>
<thead>
<tr>
<th>DATA (ONE BYTE EACH)</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>RCODE</td>
<td>The header CODE value of the message that caused the error/failure.</td>
</tr>
<tr>
<td>ERRNR</td>
<td>A predefined value according to the table XX.</td>
</tr>
</tbody>
</table>

Table 9: RCODE

The Error Messages

<table>
<thead>
<tr>
<th>HEX VALUE</th>
<th>NAME</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>Err_BadHflg</td>
<td>The equipment does not support this HFLG value.</td>
</tr>
<tr>
<td>0x01</td>
<td>Err_BadHpnr</td>
<td>The equipment does not support this HPNR value.</td>
</tr>
<tr>
<td>0x03</td>
<td>Err_BadEchk</td>
<td>The extra checksum ECHK was incorrect.</td>
</tr>
<tr>
<td>0x10</td>
<td>Err_BadObjType</td>
<td>The addressed object type is not used in this kind of document or does not exist.</td>
</tr>
<tr>
<td>0x11</td>
<td>Err_BadObjNr</td>
<td>The object number is out of range for the addressed type of equipment.</td>
</tr>
<tr>
<td>0x12</td>
<td>Err_BadCode</td>
<td>The CODE value is not valid for this object type.</td>
</tr>
<tr>
<td>0x20</td>
<td>Err_BadData</td>
<td>The size of the data field is not correct for this OTYP object type and CODE request code.</td>
</tr>
<tr>
<td>0x21</td>
<td>Err_BadResp</td>
<td>The request would cause a response that exceeds the maximum allowed by the protocol.</td>
</tr>
</tbody>
</table>

Table 10: Error Message
2.2.8 PFMs in EGEM

This structure shown in Figure 10 is used to reside PFMs in a sub-rack. It is called Chain Structure because each board is directly connected to the next board through the backplane. In this design each switch is directly connected to a PFM and both PFMs are connected to each other. The process of interacting with PFM is as follows:

Switch A sends a message to PFM2. The message is unpacked in PFM2 to check the address. If it is the right address and points to the PFM2 then a response is returned back to the switch A directly. Otherwise if it does not match, PFM2 forwards the message to PFM1. The response to this message is sent back from PFM1 to PFM2 and then to the Switch A. The same process takes place for the Switch B. Note that PFM2 is directly connected to Switch A and PFM1 is directly connected to Switch B [6].
2.3 Framework

Before explaining JCAT framework it is important to know what a test framework is. A test framework is a set of libraries, codes or tools that can be used to build up a test environment.

2.3.1 JUnit Common Auto Tester (JCAT)

JCAT is a fairly huge test framework that is currently used at Ericsson for testing and verifying equipments in telecommunication systems. JCAT is a lightweight, flexible and extensible framework. The intended goal is to have a cheap and easy to use framework, which is written in Java. Plenty of open source libraries have been used to create a robust and low maintenance framework [8].

JUnit is the actual unit testing framework for the projects written in Java and has been used as the JCAT foundation. But whereas JUnit is not suitable for the high level of testing steps such as function and system testing, it has been extended by adding some features. The extensions simplify the automation for higher forms of testing such as function or system testing. The Figure 11 shows the JCAT layered architecture. The core of JCAT framework is JUnit which is for unit testing. But a tool was needed to operate better on the system level. Thereby JUnit is extended with some functionality like event and traffic listening and etc., and the final product is JCAT. The Framework is an open source and maintained by a community at Ericsson. JCAT is a common framework that can be used in the whole company, which means that testers can reuse code, test cases and test suites developed in other parts of Ericsson. The tester does not handle the development of the framework, so tester can use his/her time to develop test cases and test suites that are suitable for the system, which is going to be test.

![Figure 11: JCAT Layered Architecture](image-url)
2.3.2 JCAT Highlights

JCAT is very quick to analyze the result of tests which need to be run frequently. It has a test case logging to allow faster analysis of test results such as what passed and what failed and what was the root cause for the failures with various features. Among them, it provides step by step results of test cases, links to HTML logs of each step inserted into the main log of the test suit. It can view HTML reports as test execution progresses. Moreover, since JCAT uses JUnit API it can easily integrate into other tools, e.g., Eclipse, ant, script, Cruise Control. Apart from HTML, JCAT can generate other different output formats. It can store the test suit execution result into SQL database or XML format. It is also equipped with a built in XML-based mechanism to exclude test cases with known faults to avoid having to analyze known faults. The HTML format of logging is illustrated in Figure 12. It provides some simplifications like [9]:

- test results shown in a tree according to how they are run, not according to how they are developed
- events received are color marked to show whether they were expected, ignored, or unexpected
- all passed test steps are ”collapsed” when opened, so that user can instantly see failed test steps

![Figure 12: JCAT HTML Report](image)
Chapter 3

Pbist Test Automation

In this section, the idea of choosing BIST technique as the solution to test DENIB-related hardware units is described. As a result of using BIST technique, a method called Pbist is invented and used to test these hardware units. Here the overview of Pbist is illustrated, and its properties and advantages are discussed.

3.1 Built-In Self Test (BIST)

Built-in Self Test, or BIST, is the technique of designing additional hardware and software features into integrated circuits to allow them to perform self-testing, i.e., testing of their own operation (functionally, parametrically, or both) using their own circuits, thereby reducing reliance on an external automated test equipment (ATE) [3, 10].

Automated test equipment (ATE) is computer-controlled equipment that tests electronic devices for functionality and performance. ATE includes the control hardware, sensors, and software that collects and analyzes the test results. ATE is considered cost efficient for high-volume testing.

The main reasons for the widespread development of BIST techniques are the fast-rising costs of ATE testing and the growing complexity of integrated circuits. It is now common to see complex devices that have functionally varied blocks built on different technologies inside them. Such complex devices require high-end mixed-signal testers that possess special digital and
analog testing capabilities. BIST can be used to perform these special tests with additional on-chip test circuits, eliminating the need to acquire such high-end testers. BIST is also the solution to the testing of critical circuits that have no direct connections to external pins, such as embedded memories used internally by the devices. In the near future, even the most advanced tester may no longer be adequate for the fastest chip, a situation wherein self-testing may be the best solution for.

Advantages of implementing BIST include:

- lower cost of test, since the need for external electrical testing using an ATE will be reduced, if not eliminated;
- better fault coverage, since special test structures can be incorporated onto the chips;
- shorter test times if the BIST can be designed to test more structures in parallel;
- easier customer support;
- capability to perform tests outside the production electrical testing environment.

The last advantage mentioned can actually allow the consumers themselves to test the chips prior to mounting or even after they are in the application boards.

Disadvantages of implementing BIST include:

- additional BIST hardware overhead;
- reduced access times;
- additional pin (and possibly bigger package size) requirements, since the BIST circuitry need a way to interface with the outside world to be effective;
- possible issues with the correctness of BIST results, since the on-chip testing hardware itself can fail.

### 3.1.1 BIST Architecture

The basic BIST architecture requires the addition of three hardware blocks to a digital circuit: A pattern generator, a response analyzer, and a test controller. Examples of pattern generators are a ROM with stored patterns, a counter, and so on. A typical response analyzer is a comparator with stored responses. A control block is necessary to activate the test and analyze the responses. However, in general, several test-related functions can be executed through a BIST controller circuit [11].
Consider a hierarchical application of the BIST concept. The system consists of several circuit boards. Each board may contain several VLSI chips. Figure 13 shows such a system. The BIST controller at the system level can simultaneously activate self-test on all boards. The BIST controller on each board, in turn, activates self-test on each chip on that board. A chip BIST controller is responsible for executing self-test on the chip and then transmitting the result (fault-free or faulty) to the test manager of the board containing the chip. The board BIST controller accumulates test results from all its chips and transmits them to the system BIST controller. Using these results, the system BIST controller can isolate faulty chips and boards [11, 12].

![Figure 13: BIST Method Structure](image)

### 3.2 Product Built-In Self Test (PBIST)

The new and highly cost-effective method that is proposed to test the DENIB-related hardware units is called Product built-in self test (Pbist). The principles of this method are inspired by the known BIST technique. To perform this method a new object is added to the DENIB-related units to test their own functionality and also the hardware unit itself. The DENIB protocol is changed to support Pbist test object by adding the new OTYP_PBIST type. Object number ONBR for this object is handled in different way rather than the other DENIB objects. One instance of the PBIST type, OTYP_PBIST is mandatory (with object number 0) and is to be considered as Pbist Controller. All additional instances (with ONBR>0) are to be considered as specific test cases [13].
3.2.1 PBIST Related CODEs

Here the different requests to Pbist Controller are described [13]:

- **CODE_Name**: Read OTYP_PBIST instance name (ONBR= 0/>0)
- **CODE_Info**: Read Pbist info (ONBR= 0/>0)
- **CODE_Start**: Enter Pbist mode (ONBR = 0)
- **CODE_Start**: Excute Pbist test (ONBR>0)
- **CODE_Stop**: Exit Pbist mode (ONBR = 0)
- **CODE_Stop**: Abort Pbist test execution (ONBR>0)

To run Pbist tests, first a request with object number ONBR = 0 and code CODE_Start must be sent. Afterwards, the requests to run Pbist tests could be sent and executed by Pbist Controller.

3.2.2 PBIST Structure

The controller unit of DENIB-related hardware units consist of three blocks, DENIB interpreter, Pbist Controller, Response Analyzer. The structure of Pbist in PFM is illustrated in Figure 14. The steps of Pbist tests execution are described as follows

1. Sending a MCL2 request to the PFM from JCAT on the server or from a standalone pc.
2. The request is received by DENIB interpreter which is responsible to read the MCL2 message and controls the header and checksum. If the received message is the right one, this interpreter removes the header and checksum and extract the DENIB data part. The reason is that only DENIB format is acceptable for Pbist Controller.
3. Next, the DENIB data is sent to the Pbist Controller which is responsible to activate the Pbist mode, execute Pbist tests or stop execution of tests. The DENIB-related hardware units consist of several functions and for each, one or more Pbist tests are considered. Every function must be put under test independently from the other functions.
4. A typical response analyzer is a comparator that compares the result of the test execution with the expected results, which are stored in the hardware itself. The test is passed if the outcome of the test execution is matched with the expected result otherwise it is interpreted as failed test.
5. The result is sent back to the DENIB interpreter to frame it to an MCL2 message and then transmitting the result to the request issuer.
Figure 14: PFM Pbist structure
3.3 Running Pbist Tests

In this section, we provide a general description of the process of running test cases in the form of a Pseudo-code. The details are discussed in the proceeding sections.

```
1. IF (connected to SUT) THEN
2.   READ productName and revision from SUT;
3.   PARSE given xml file;
4.   IF (corresponding productName and revision are found in xml file) THEN
5.     Sending a DENIB request to activate the Pbist controller;
6.     IF (activated) THEN
7.       WHILE (The number of tests>0)
8.         { 
9.           Reading the desire pbist test and its iteration number;
10.          Executing the specified Pbist test;
11.          READ the returned DENIB message;
12.          Extract the test result from the message;
13.          Display it to the request issuer as Pbist test passed or failed
14.         }
15.     ELSE
16.         Error message (PBist could not be activated)
17.     ELSE
18.       ELSE
19.         Error message (It is not the right product)
20.     ELSE
21.         Error message (Connection failed)
```

Algorithm 1

In Algorithm 1, we first check that the connection to the SUT is established. Then, in lines 2-4 product name and revision is read from the SUT and compared with the product names and revisions present in the given configuration file. To do these, the product name and revision should be queried by sending a DENIB message to the SUT and also the configuration file needs to be parsed. After these, the product name and revision of the SUT is searched in the configuration file to find the test cases related to the SUT. When the test cases are found, in line 5, a message is sent to activate the PBIST mode of the microcontroller of the SUT. If this is successful, in lines 7-11, the test messages are sent and the results are retrieved from the received responses. These results will then be shown in an understandable format. In the following sections we describe, in detail, the process of serial port communication (e.g., sending and
receiving messages), parsing the configuration file, activating the PBIST mode, and retrieving the results.

### 3.3.1 Serial Port

First step is communicating with PFM. This communication is established via a serial port named COM1. Serial port programming at its most basic level consists of a series of steps regardless of the operating system that one is operating on. These steps are opening the port, configuring the port, reading and writing to the port, and finally, closing the port. The communication with the port is not mentioned in the Algorithm 1. In the Section 4, where the implementation of Java classes are explained we discuss the process of connecting and communicating with PFM in detail [13].

The Java Communications API (aka. “javax.comm”) is used to access to the serial port. It provides applications access to RS-232 hardware (serial ports). There is also an alternative for “javax.comm” API that is called “RXTX”. It’s an API that supports all types of platforms including Windows, Linux, and Mac.

### 3.3.2 Test Configuration File

A test configuration file is a file that contains information to test a product. It can be in various formats although for a DENIB-related hardware unit it is in XML format. It includes information about the product’s name, revision, identification number of test cases, and the number of iteration to execute each test case. There are a number of test cases for a product with a specific revision. Each of these test cases is a Pbist test aimed to test a specific functionality of the product. For example in the first product “BFBXXX”, revision “R1A” the first Pbist test is employed to disable the fan named EWD on the PFM. An example of a test configuration file is illustrated below:
**Figure 15: Test Configuration File**

### Parsing XML File:
Test configuration is an XML file. To run test cases, first they need to be read from the XML file. This can be done through parsing. By using JDOM API and SAX Builder, the XML markup is converted to a JDOM Document containing the various elements, Children, their attributes and their relationships. This document can be queried to get diverse information. This API is used to parse the XML test configuration file in this project. For the further information on the implementation, we refer the readers to Chapter 4 [15].

### JDOM:
JDOM is a unique Java toolkit for working with XML. This API is used to represent an XML file in a way that is simpler and highly efficient for reading, writing and modification. It is a lightweight API, straight forward, fast and Java specific. JDOM must be integrated with some other existing API such as SAX or DOM. It is used on top of either Document Object Model (DOM) or Simple API for XML (SAX) for different programs and needs [15, 16].

### Test Configuration File Usage:
To execute Pbist tests, first step is to connect to the system under test (SUT). Serial port is considered as the way to communicate with DENIB-related hardware units. As soon as the connection to the product is established, requests are sent via a user interface or a command line application.
The first request after the serial connection is established is asking about the product’s name and revision. If they matched with the corresponding name and revision in the test configuration file, then the next requests could be sent. There are a number of tests for each product with a specific revision, the order of executing and the number of iterating each test are specified in the test configuration file. As you see in the Figure 15, for example, if we connect to product BFBXXX with revision R1A, then the first set of tests are executed.

### 3.3.3 Activate Pbist Controller

When a connection to the SUT is established, the first request is to activate the Pbist Controller. A request with code CODE_Start to the object number 0 (ONBR=0). The returned response from the SUT indicates that the Pbist Controller is in Pbist mode already or Pbist mode is just entered [13].

### 3.3.4 Execute a Pbist Test (ONBR>0)

Pbist Controller must be activated to be able to execute Pbist tests. Thereby, as soon as the Pist Controller is activated, a request is sent to execute a Pbist test. A request with code CODE_Start and object number greater than zero (ONBR>0) is sent to run a specific Pbist test.

### 3.3.5 Checking the Output

In some cases after the execution of a test is finished, the result must be validated by an analyzer. It compares the test execution outcome with the expected result which is already stored in the hardware. If they match it means that the Pbist test is passed otherwise failed. However in some other cases the result happens during the test execution. For example if the test is the request to keep the fans at full speed, then analyzing the result is happened during the test execution.
3.4 Interaction with PFM

The Shelf Manager is responsible for monitoring conditions of modules and other shelf components and controlling their operation in order to keep them working properly. It works together with the Intelligent Platform Management Interface (IPMI) infrastructure to monitor the status of the system and correct problems when necessary. The Shelf Manager reports events and anomalies to a system manager and responds to action requests from the system manager.

GEP3 (Generic Ericsson Processor 3) is an x86 based general processing board within EGEM (Evolved Generic Ericsson Magazine). It is the 3rd generation in the GEP family. GEP3 is designed to fit different systems and applications with special requirements. These are high performance, energy efficiency, cost effective solution with superior capacity per volume and based on state of the art components.
The whole process of sending a message to PFM as it is shown in the figure above is as follows

1. Log in into the GEP server form the workstation. It can be done by using SSH (Secure Shell).

2. After the connection to the GEP is established, messages can be sent from JCAT framework toward PFM. The format of the messages between GEP and other PIUs, such as Switch SCXB is SNMP format. The request sent by JCAT is MCL2 and this messages need to be nested in the SNMP message format and then send to the Switch SCXB.

3. The message received in the Switch SCXB is an SNMP message but this message can not be forwarded to the PFM directly. As you know so far the standard message to communicate with PFM is MCL2 message. So we need to have an application on the Switch SCXB to extract the MCL2 message from the SNMP message. We call this application MCL2 Detector.

4. After the MCL2 message is retrieved, then it is sent to the PFM.

5. When the message is received by PFM, it must be checked if its header and checksum is acceptable and this is carried by DENIB Interpreter. If the message is acceptable then the header and checksum are removed from the message and the remaining part which is DENIB part is sent to the Pbist Controller.

6. Pbist Controller performs some analysis and get the Pbist test number, execute the test and send the result to the Response Analyzer.

7. In this step the result of the test is compared with the expected result and the result is sent to the DENIB Interpreter. Here the DENIB data is framed and convert to MCL2 message format and send out to the Switch.

8. The MCL2 message arrived in the Switch SCXB is framed by SNMP protocol and send back to the GEP server.
3.5 How to Test a SUT with JCAT

To test an SUT for example PFM with JCAT three main packages are needed [8, 9]:
- JCAT Framework
- JCAT SUT specific extension
- JCAT Test cases and Test suites

3.5.1 JCAT Framework

This package contains all general data and (not just for specific SUT). It contains:
- HTML/database logging
- Generic setups
- Event handling interface
- Traffic handling via traffic control and a traffic interface

3.5.2 JCAT SUT Specific Extension

Different kinds of extensions are available. First type of extension is providing a way to connect and interact with a specific SUT. For example the serial port connection is used as a way to interact with PFM.

The automated test tool that is developed to test the SUT is a combination of a number of components like packages, classes, functions, etc. Those components, which are providing a way to interact with SUT, are known as Test Drivers. A group of classes are used to create a way to interact with the PFM such as “SerialPortCommunication” and “Denib”. These two classes must be residing in the Extension package.

The second type of extension is an event listener. These listeners provide a way to listen to different interfaces such as Snmp, Corba, etc. These listeners are alert to fetch alarms, notifications or other asynchronous events, for example log entries.

The third type of extension is related to traffic management. They allow starting, stopping and gathering of statistics from traffic to define pass/fail criteria per test case.
3.5.3 JCAT Testcases and Test Suites

The classes that contains test cases and test suits are placed in this package. Test cases executed with JCAT is in a hierarchal tree-manner consisting of the following entities.

**Test Suite:** The top Structure that contains an arbitrary number of Suits and Test Cases.

**Test Case:** Contains an arbitrary number of Sub Tests.

**Sub Test:** The tests, which are actually executed. The higher containers are for the structural purpose.

3.5.4 JCAT Structure

A simple overview of the structure of JCAT framework, its interfaces and different types of extensions are illustrated in Figure 16.
Chapter 4

Implementation

The details of the implementation of the Pbist tool are described in this chapter. This tool is implemented in Java because it is going to be integrated with JCAT test framework, which uses JUnit as its core. It is a free, open source and robust framework. On the other hand, Java programming language is used enormously in many projects; one reason is the code written in Java is portable. So it is possible to migrate and execute this module on other platforms such as Linux. Implementation of the new module, GUI, logging feature and process of integration to JCAT are also discussed, in this chapter.

4.1 UML

The UML diagram [17] of the Pbist code is shown in Figure 17. The top section in each box shows the name of the class. The rest includes class attributes and functions of the class. Furthermore the relationships between different classes are illustrated. In the following sections of this chapter we explain each class separately.
Figure 17: Phist UML Diagram
4.1.1 SerialPortCommunication Class

SerialPortCommunication consists of four public methods, “openPort, closePort, serialWrite, serialRead”. Serial port is opened by “openPort” method. This method gets a serial port name string as input and it is a void method. When the serial port is opened, a pointer is assigned to a public attribute of it. The method “serialRead” reads a number of bytes from the serial port input buffer and writes those bytes into a byte array. “serialWrite”, writes the specified byte array to the serial port. Invoking “closePort” method closes the serial port.

4.1.2 Denib Class

Denib class is essentially a data type to store values of OTYP_Obj, CODE values, source and destination addresses, flags and header bytes. It consists of two inner classes; “DenibMessage” and DenibBuffer.

- DenibMessage class provides getters and setters for the attributes of the Denib class. Using the setters, a concrete Denib message is formed and with getters the values of the Denib message is read.

- DenibBuffer class consists of four public methods, “calc_dcrc, package_denib, calc_crc, package_frame”.

  - package_denib: This void method gets an input which is an object form the DenibMessage class. By invoking this method a DENIB message is initialized by putting the different bytes of the DENIB data together.

  - The CRC methods, calc_dcrc and calc_crc, calculate the checksum by one’s compliment method. The ones' complement of a binary number is defined as the value obtained by inverting all the bits in the binary representation of the number (swapping 0's for 1's and vice-versa). The ones' complement of the number then behaves like the negative of the original number in most [18].

CRC (Cyclic Redundancy Checks) is a popular method for determining if transmission has been corrupted. A Checksum is used for error checking while transferring a file [18]. We know that data flows across the network in the form of packets. So, checksum is a computed value that is dependent on the contents of a file. For each packet the computed checksum will be unique. This computed value is transmitted with the packet when it is transmitted. The receiving system checks the checksum and on the basis of checksum it accepts or rejects the packet. This is mainly used when checking the validity of a received packet is necessary.
○ The void Package_frame method gets an input that is an instance of the DenibMessage class. By invoking this method the header and check sum are added to the DENIB data buffer.

4.1.3 Testcase and Testcases Class

The Testcase class is a datatype to store the information of test cases such as description, id, count, etc. While Testcases class is a datatype to store a list of objects of Testcase class.

The test cases information is read from the given XML file by ReadXML class (See section 4.1.4). To store test cases information, ReadXML class uses Testcase and Testcases classes and later pfm class to execute test cases on the hardware.

4.1.4 ReadXML Class

This class provides facilities to parse XML files and get the information of test cases. In this class, method “getFrameWorkDetails” defined in this class is responsible to parse the XML file by using JDOM parser [15]. It gets two input parameters that specify the product name and revision. The given XML file is the test configuration file that consists of a series of test cases for a number of hardware. This file is used to get the test cases, their number of iteration for each test case, their time out, and their description and purpose.

4.1.5 PbistMessage Class

In this class the DENIB requests to the SUT are made. It consists of several methods which each of them represents a request. These methods return back an object of Denib type. It contains the data, which is going to be written on the serial port.

4.1.6 ReadBuffer Class

This class provides an implementation of Runnable interface to execute a thread. The thread is used in pfm class for reading the responses of the hardware sent over the serial port. This class is used when test cases are executed from an XML file or given by manual option of the user interface.

4.1.7 Pfm Class

pfm class is the main class where the execution of the program starts. It is responsible to create the Pbist Graphical User Interface (GUI), manage components of the GUI and their events. All other classes are used by pfm to run the final application. The logging system is handled by this class, which makes users able to see the result of test case executions.
4.2 User Interface

The investigation of finding a reliable way to interact with the unit and performing some tests has ended up with the graphical user interface, shown in Figure 18. We designed this GUI to interact with the hardware.

To start with the GUI, first a serial port must be chosen and the open the port by clicking on the “Open Port button”. As it is seen in the Figure 18 the button “Get Controller Name” in the Controller panel, is to get the name of the unit’s microcontroller is extracted and shown in the Name text field.

The next panel is PBist Controller which is responsible to execute the tests on the unit. When “Update Pbist Info” is tricked, the Name, Number of TCs and PBist Mode are filled up with the name of the Pbiste tests controller, the whole number of Pbist tests, and the Pbist controller mode, respectively.

Test Cases panel shows the different input methods to get test cases and then execute them. A method is called Manual which we give the number of test case by using the combo box and choose the number of test case. By pressing the button “Get TestCase Info” the Name text field is filled with the name of the test. Then by clicking the “Execute TCs” the specified test is performed.

The other input method is browsing an XML file. As soon as it is opened, if it includes the name and the revision of the hardware under test, the test cases start to run.

Notice that there is a check box in the GUI called “Log”. If this box is checked then every action that happens on the GUI is logged in the text area on the button of the GUI. Another logging system also exists which starts to log as soon as the GUI window is opened. This log is saved in a directory in the system that is interacting with hardware.
Figure 18: PBIST Tool
4.3 How to Test PFM with JCAT

In this section we strive to explain the process of developing extensions and test cases for PFM. It is chosen as an example for this short tutorial; however the tool is a general solution to test any other SUT with JCAT. What is needed for this development is a computer system which, Java runtime environment and Eclipse IDE are installed on it [8].

First, the framework and its needed extensions must be setup in order to write the PFM test cases. Two projects should be created, one for extensions and the other for test cases and test suites. Two names ”jcat-pfmpbistdemo-extensions” and “jcat-pfmpbistdemo-testcases” are given to these projects since the SUT name is PFM.

May reader asks why two projects are created? Couldn’t it be one project? And where is the JCAT framework?

In continue you will see the process of creating extensions and test cases packages step by step:

1. The first step is to define PFMPbistDemoTestCase class. This is the Java class that will be extended by all of the test classes. In other words, this class is called before running each test. It extends from the “NonUnitTestCase” class from the JCAT framework and do not do anything for now. The reason to need it now is that “NonUnitTestCase” cannot be instantiated since it is an abstract class. It will be used later.

A new package is defined for “PFMPbistDemoTestCase” class in “jcat-pfmpbistdemo-extensions” package. Let us call this package “se.ericsson.jcat.pfmpbistdemo.fw” and then a class called “PfmPbistDemoTestCase” is defined in this package.

The code to show the changes explained above follows.

```java
package se.ericsson.jcat.pfmpbistdemo.fw;
import se.ericsson.jcat.fw.NonUnitTestCase;
public class PfmPbistDemoTestCase extends NonUnitTestCase {

    public PfmPbistDemoTestCase (String name) {
        super(name);
    }
}
```
2. The second step is to define `PfmPbistDemoTestSetup` class. This Java class extends form "NonUnitFrameworkTestSetup" class which initializes some common JCAT functionality like logging, starting and stopping listeners and traffic handling. Here the "NonUnitFrameworkTestSetup" class is extended because that class is abstract and cannot be initialized. A new sub-package is defined for this class in “jcat-pfmpbistdemo-extensions” package. Let us call it “se.ericsson.jcat.pfmpbistdemo.setup”. Then class “PfmPbistDemoTestSetup” is placed in this package.

The code to show the changes explained above follows.

```java
package se.ericsson.jcat.pfmpbistdemo.setup;
import junit.framework.Test;
import se.ericsson.jcat.fw.NonUnitFrameworkTestSetup;
public class PfmPbistDemoTestSetup extends NonUnitFrameworkTestSetup{
    public PfmPbistDemoTestSetup(Test testName) {
        super(testName, "dummyDumpName");
    }
}
```

3. In this step a test is implemented which checks if the voltage is within the expected range. For this purpose the class “PfmVoltageMonitoringTestCase” is developed. The code is shown below.

```java
package se.ericsson.jcat.pfmpbistdemo.tests;
import se.ericsson.jcat.pfmpbistdemo.fw.PfmPbistDemoTestCase;
public class PfmVoltageMonitoringTestCase extends PfmPbistDemoTestCase {
    public PfmVoltageMonitoringTestCase(String pfmVoltage){
        super(pfmVoltage);
    }
    public void testPfmVoltage(){
        setTestcase("TC-pfm-1", "PFM Voltage");
        setTestInfo("System.getProperty('PFM.Voltage').");
        setTestFailure("Failes if the voltage is out of the expected range");
    }
}
```

As it is seen in the code, this class extends from previously implemented “PfmPbistDemoTestCase” class. The “testPfmVoltage ()” method is a test that monitors the PFM voltage. Note that all testcases name start with “test”. A new sub-package for
“PfmVoltageMonitoringTestCase” class is created in “jcat-pfmpbistdemo-testcases” package. This package is called “se.ericsson.pfmpbistdemo.tests”.

4. The forth step is to define PfmPbistDemoSuite class. And finally, all these developed classes are combined together in “PfmPbistDemoSuite” class. Let us have a look on the code below and go through it line by line:

```java
package se.ericsson.jcat.pfmpbistdemo.suites;
import se.ericsson.jcat.pfmpbistdemo.setup.PfmPbistDemoTestSetup;
import se.ericsson.jcat.pfmpbistdemo.tests.PfmFanSpeedeTestCase;
import junit.framework.Test;
import junit.framework.TestSuite;

public class PfmPbistDemoSuite extends TestSuite {
    public PfmPbistDemoSuite() {
        super("PfmDemoSuite");
        addTest(new PfmVoltageMonitoringTestCase("testPfmVoltage"));
    }
    public static Test suite() {
        return new PfmPbistDemoTestSetup(new PfmPbistDemoSuite());
    }
}
```

As it is shown in the code the “PfmPbistDemoSuite” class extends directly from “TestSuit” class, which is owned by JUnit. The “TestSuit” is a composite of tests. It runs a collection of test cases. The “addTest” method will simply, as the name says, add our test into a test execution list.

But here, the “suite” method needs a special attention. This method could be considered as the main method in a java program. This is the method that starts first after the suite is run and, in this example, Suit method runs “PfmPbistDemoSuite” class, collects all test cases and passes them to the “PfmPbistDemoTestSetup” class. In other words, you add the tests to be run to a TestSuite object and return it to JUnit to run. A new sub-package for “PfmPbistDemoSuite” class is created in “jcat-pfmpbistdemo-testcases”. It is called “se.ericsson.jcat.pfmpbistdemo.suites”.


4.3.1 Running PfmVoltageMonitoringTestCase

This part is easy but there are still some things that need to be mentioned, so let us take it one step at the time. First, it is needed to define the logging appenders. A file needs to be created in both “jcat-pfmpbistdemo-extensions” and “jcat-pfmpbistdemo-testcases”. It is called log4j.properties. This file tells the Apache logger where to append the logging result.

The log4j API [20] is used for the logging purpose. Log4j is a flexible logging library, and an open source project from Apache. This logging API helps developer to see application failures with detailed context. The target of the log output can be a file, an output stream, or many other output targets.

We still need to specify a couple of VM arguments before the test starts. The VM arguments are arguments that are feed to our execution program. When passed to the Eclipse, this option is used to customize the operation of the Java VM (Virtual Machine) used to run Eclipse. One of these arguments is logdir. It is telling where to store the results. This parameter is actually optional but controlling the location of the logging output is something you’ll probably want to do. Another argument is log4j.configuration, which is the address to log4j.properties.

Now the test suite is ready to run. When you run the test suit you see the result in the console in Eclipse IDE. You would be able to see all logging output in either consol or in an HTML page. The output is accessible from logdir directory (the location that is specified to store the logs).
# To set another level of logging use:

# -Djcat.logging=<Level>
# when starting JCAT
# for example -Djcat.logging=debug
# <Level> is a valid log4j level

# Level DEBUG - The DEBUG Level designates fine-grained informational events that are most useful to debug an application.

# Level INFO - The INFO level designates informational messages that highlight the progress of the application at coarse-grained level.

# Level WARN - The WARN level designates potentially harmful situations.

# Level ERROR - The ERROR level designates error events that might still allow the application to continue running.

# Level FATAL - The FATAL level designates very severe error events that will presumably lead the application to abort.

# In addition, there are two special levels of logging available: (descriptions borrowed from the log4j API http://jakarta.apache.org/log4j/docs/api/index.html):

# Level ALL -The ALL Level has the lowest possible rank and is intended to turn on all logging.

# Level OFF - The OFF Level has the highest possible rank and is intended to turn off logging.

log4j.rootLogger=INFO, A1

# Special case for the j2ssh implementation, which seems to have log level set to DEBUG by default
log4j.logger.com.sshtools.j2ssh=OFF
# A1 is set to be a ConsoleAppender.
# A1 uses PatternLayout.
log4j.appender.A1.layout.ConversionPattern=%d{ISO8601} %m
# need to set follow = true for the appender as it gets moved around in junit tests
log4j.appender.A1.follow=true

Figure 19: Example of a log4j file
Chapter 5

Conclusion

Several hardware units at Ericsson use DENIB protocol as the means of the communication, one of such is PFM. These units are composed of a series of objects which each of them represents a physical part of the hardware or its functionality. A new object is developed and added to the collection of objects that are supported by DENIB protocol. This specific object which is called Pbist is developed for testing purposes. Pbist is cheaper and easier to maintain testing approach for hardware units. However, the Pbist tests can not be performed manually and need to be executed automatically. In this project we investigate to find a generic way to perform the Pbist tests on different hardware units.

As a result of the investigation and to show the outcome in practice a tool is designed and developed that is called Pbist. It is an automated test tool which is designed to cover the features for a generic tool that is automatically executes Pbist tests. It is designed in a way that could be used for various hardware units using DENIB protocol for communication. Tester would be able to test the hardware remotely when it is up and under operation. By using this tool, testing would be easier and much more cost-effective than manual testing although some functions of a hardware need to be tested manually and they are not implemented as a Pbist test. In this way the need for the resources such as money and human are reduced. Pbist provides a logging mechanism which makes it possible to use the tool anonymously on long and time consuming stress tests. The provided log file by tool includes the complete information about timing, percentage of the failures, and descriptive description of the test cases. It is a trustful tool that could be used as a long-term solution.
In this report, we have described, in great detail, the integration of the proposed solutions into an internal testing framework used extensively at Ericsson called JCAT. Moreover, the usage and integration of the solutions into various platforms and devices.
Bibliography


Appendix A

**jSviftTerm Software**: It is a software that is used during this thesis. All use cases and examples about DENIB in this thesis report are made by jSviftTerm [21].

jSviftTerm is a generic human user interface software which has been used in this project to learn the DENIB protocol by sending request and getting response to the old version of PFM. Its design is quite simple and user-friendly. An example of jSviftTerm is shown in Figure 20. This is the page will be seen whenever you start the program. Here the desire port can be chosen. In this figure port COM1 as the default port is set (it is in Windows environment). But once you open the combo box you will find your other serial port options.

![Figure 20: jSviftTerm](image)
**SVIFT Editor:** As it is seen in Figure 22 in the left side, Header and Data areas, all fields of the message can be edited. When the send button is pressed, the outgoing formatted message as well as all responses is shown in the main (white) text area.

![SVIFT Editor](image)

*Figure 21: SVIFT Editor*

**SVIFT Tree:** The SVIFT Tree tab builds a tree view of the SVIFT objects contained in the PFM connected to the port. Requests are automatically emitted to build the entire object tree of units, object groups and objects. By selecting an object, its information and controls are visible in right pane.
Figure 22: JSVIFT Tree