Methods for software quality improvement

A study performed upon the C10v2 system for Metrima AB

Amir Sejdinovic  
asc01001@student.mdh.se

Farshid Atachi  
fai01001@student.mdh.se

21 May 2006

Examiner: Christer Sandberg  
Supervisor at Metrima AB: Carl Falk  
Supervisors at Mälardalen University:  
Daniel Sundmark and Andreas Emerdahl

Department of Computer Science and Electronics, IDE  
Mälardalen University, Västerås
Summary

The concept of "quality" has been the essential cause of competition and development over the years. Software development is one of the subjects where the need of new quality improvement methods constantly grows. The reason for this expanding need is not just the fact of increased use of the software applications, it is also because the software becomes more and more complex.

This thesis gives a description of a study made at Metrima as an examination of one of the Metrima’s core products, C10v2. C10v2 is an embedded product and one of the leading commercial products for broadband connectivity for metering applications. During this thesis we examined a number of methods that can improve software quality and reduce the time and money spent on debugging. The result of this research is a detailed discussion about Code standards, Code inspections, Static analysis tools, Compilers and Dynamic analysis tools. The choice of these methods is based on several factors like: the time it takes to apply this method, the resources it requires, Metrima’s situation today and financial factors. Every chapter contains a subchapter describing the new model suited for the C10v2 C code management.
Acknowledgement

This master thesis was performed at the Department of Computer Science and Electronics, IDE, at Mälardalen University in Västerås, Sweden, in collaboration with a product development company Metrima in Stockholm. The work was performed by Amir Sejdinovic and Farshid Atachi as the final educational step at the Mälardalens University.

Our thanks goes mainly to our supervisor at Metrima Carl Falk, hardware/software engineer, for leading and giving us the opportunity to perform our master thesis at Metrima Development Centre and making us feel as a part of Metrima. Also thanks to Ulf Jonsson, hardware/software engineer at Metrima for his support. Further we would like to thank our supervisors at the Mälardalen University, Daniel Sundmark and Andreas Emmerdahl for their inspiration and support and our examiner Christer Sandberg.

Finally we would like to thank the companies we were in contact with for their support during this thesis.
Index

1 INTRODUCTION ........................................................................................................................................... 1
1.1 ABOUT METRIMA ................................................................................................................................. 1
1.2 BACKGROUND .......................................................................................................................................... 1
1.3 DELIMITATIONS .................................................................................................................................... 1
1.4 DEFINITIONS AND ABBREVIATIONS .................................................................................................... 2
1.5 RELATED WORK ...................................................................................................................................... 3
1.6 THESIS OUTLINE ................................................................................................................................... 3
1.7 SYSTEM OVERVIEW .............................................................................................................................. 4
   1.7.1 General description .......................................................................................................................... 4
   1.7.2 Detailed Platform Description ...................................................................................................... 4
   1.7.3 Development Environment Description ........................................................................................ 9
2 PROBLEM DESCRIPTION ............................................................................................................................ 14
   2.1 STATIC ANALYSIS ............................................................................................................................. 14
   2.2 DYNAMIC ANALYSIS ....................................................................................................................... 14
3 METHODOLOGY ........................................................................................................................................ 16
   3.1 MARKET ANALYSIS ........................................................................................................................... 16
   3.2 LITERATURE STUDY .......................................................................................................................... 16
   3.3 DIALOG WITH THE EMPLOYS AT METRIMA .................................................................................. 16
   3.4 INTERNET RESEARCH ..................................................................................................................... 16
   3.5 BUG REPORT DATABASE ................................................................................................................. 16
4 MARKET ANALYSIS .................................................................................................................................. 17
   4.1 ABB .................................................................................................................................................... 17
   4.2 ERICSSON .......................................................................................................................................... 18
   4.3 DIFFERENCE BETWEEN ERICSSON AND ABB .............................................................................. 20
5 CODE STANDARDS ................................................................................................................................... 21
   5.1 A STUDY ABOUT CODE STANDARDS .............................................................................................. 21
      5.1.1 Header files .................................................................................................................................. 22
      5.1.2 C files .......................................................................................................................................... 22
      5.1.3 Indent style .................................................................................................................................. 23
      5.1.4 Programming Rules .................................................................................................................... 24
   5.2 WHICH QUALITY METHODS ARE USED AT METRIMA TODAY .................................................... 27
      5.2.1 C10v2 code standard .................................................................................................................. 27
   5.3 WHICH METHODS ARE USED AT OTHER COMPANIES ............................................................... 32
   5.4 NEW MODEL SUITED FOR THE C10v2 ............................................................................................ 33
6 CODE INSPECTIONS .................................................................................................................................. 35
   6.1 A STUDY ABOUT CODE INSPECTIONS ........................................................................................... 35
      6.1.1 Code reading techniques ............................................................................................................ 36
      6.1.2 Software Inspection .................................................................................................................... 37
   6.2 WHICH QUALITY METHODS ARE USED AT METRIMA TODAY ................................................... 40
   6.3 WHICH METHODS ARE USED AT OTHER COMPANIES ............................................................... 42
      6.3.1 ABB .............................................................................................................................................. 42
6.3.2 Ericsson ............................................................................................................... 44
6.4 NEW MODEL SUITED FOR THE C10v2 ................................................................... 45

7 STATIC ANALYSIS TOOLS .................................................................................. 50

7.1 A STUDY ABOUT STATIC ANALYSIS TOOLS .................................................. 51
7.1.1 PC-Lint ........................................................................................................... 52
7.1.2 RSM .............................................................................................................. 54
7.2 WHICH QUALITY METHODS ARE USED AT METRIMA TODAY .................. 56
7.3 WHICH METHODS ARE USED AT OTHER COMPANIES ..................................... 58
7.3.1 LINT .............................................................................................................. 58
7.3.2 SP lint .......................................................................................................... 59
7.3.3 Coverity ...................................................................................................... 59
7.3.4 RSM .............................................................................................................. 59
7.4 NEW MODEL SUITED FOR THE C10v2 .......................................................... 59

8 DYNAMIC ANALYSIS TOOLS ........................................................................ 61

8.1 A STUDY ABOUT DYNAMIC TESTING ................................................................. 61
8.2 CODE COVERAGE .......................................................................................... 62
8.3 TEST TOOLS FOR DYNAMIC ANALYSIS ...................................................... 63
8.3.1 LabVIEW .................................................................................................... 64
8.3.2 CANTATA++ ................................................................................................. 64
8.3.3 CTC++ and CMT++ .................................................................................... 65

9 COMPILERS ....................................................................................................... 66

9.1 IS GCC THE ONLY SOLUTION? ........................................................................ 66
9.2 GNU COMPILER COLLECTION – GCC ........................................................ 66
9.2.1 Code Coverage using GCC, gcov .............................................................. 67
9.2.2 Code Profiling using gprof ........................................................................ 67
9.3 DEFAULT COMPILER OPTIONS ..................................................................... 68
9.4 NEW COMPILER OPTIONS SUITED FOR THE C10v2 .................................... 68
9.4.1 Control options ........................................................................................ 69
9.4.2 Optimization ............................................................................................. 70

10 CONCLUSIONS AND FUTURE WORK ...................................................... 73

11 REFERENCES .................................................................................................. 74

11.1 ARTICLES ..................................................................................................... 74
11.2 BOOKS ......................................................................................................... 75
11.3 INTERNET ..................................................................................................... 75

12 APPENDIX ....................................................................................................... 77

APPENDIX A – CODE INSPECTION RESULTS USING THE CHECK-LIST AND AD-HOC METHOD ... 77
APPENDIX B - CONFIGURATION FILE FOR PC-LINT ........................................ 80
1 Introduction

1.1 About Metrima

Metrima (NorthNode from the beginning) was established in 1999 by three engineers from the Royal Institute of Technology, KTH. A union with three other companies was made in the end of 2004 in order to offer better and more competitive products. Today the company has grown and employs more than hundred people working in three cities in Sweden, Stockholm where the main office is located, Linköping and Karlskrona. Metrima’s business concept is to provide solutions for automated measurements of electricity, water, gas and district heating. Together with several types of alarms like moist, fire and temperature these techniques gives Metrima the opportunity to provide products for a new more fair living putting the Metrima towards a bright future.

1.2 Background

Our cooperation with Metrima started via [Exjobb]. The subject of the thesis was to find techniques for software quality improvement for one of Metrima’s products C10v2. This is a unit that handles measurement values of electricity, temperature, alarms, water and communication with a superior system. See Figure 1.

1.3 Delimitations

The economic delimitations were one of our largest difficulties since many of those tools we wanted to evaluate did not have support for the development environment used at Metrima. In order to evaluate these tools the manufactures needed to make some reconfigurations in their evaluating products that they wanted payment for.

Secondly, due to the large amount of work made within software quality and testing, we had to limit our study to the techniques mostly used in industry. We focussed on techniques which produce good results on short time frames.
### 1.4 Definitions and Abbreviations

| **Basic Block** | A sequence of code with a single entry and exit point, without internal branches. |
| **BGA** | Ball Grid Array, a ball grid array is a type of microchip connection methodology. Ball grid array chips typically use a group of solder dots, arranged in concentric rectangles to connect to a circuit board. |
| **bNet** | BrightNet protocol. This communication protocol is developed at Metrima. |
| **CMM** | Capability Maturity Model  
A model which is used for measuring development in management processes as it gets more experienced. This standard is used to measure organizations progress through five levels - initial, repeatable, defined, managed, and optimizing. [SQE] |
| **DAQ-card** | A DAQ (*data acquisition*) system is tool used to gather, analyze and record information. This card produced by National Instrument, NI, works as an interface between the computer and the target being tested. More information about the DAQ-card can be found at [NI] |
| **GCC** | GNU Compiler Collection. It is a collection of compilers for different programming languages, including C, C++ and Java. |
| **gprof** | Part of the binutils package, a collection of GNU utilities for managing and manipulating binary object files. |
| **I2C** | Inter-Integrated Circuit Communication, a type of bus communication which is used to connect integrated circuits (ICs). I2C is a multi-master bus, which is used for multiple chips connection through the same bus and where each chip can act as a master by initiating a data transfer. |
| **IDE** | Integrated Development Environment |
| **Regression testing** | Helps to ensure that changes made to the software code do not break other previously working parts of the software. |
| **TPI** | Test Process Improvement, a practical step-by-step guide to structured testing. [TPI] |
| **Negative testing** | Test where the expected result is an error or failure. |
1.5 Related Work

Over the years a huge amount of research has been made regarding software quality improvement. Universities around the world pay great attention on this topic. Lot of books and white papers are written about software quality. There are many organisations that hold seminars regarding this issue. All of these institutions and organisations seem to agree that this is a present issue and that research about it need to go on.

One of the books about software quality improvement that we used as a reference in our thesis is written by Jeff Tian, professor at Department of Computer Science and Engineering at Southern Methodist University, Dallas. [Tian05a]

Another example on similar software quality improvement jobs we can refer to is Oliver Laitenberger´s report about software reading techniques. [Lait95a]

Sara Eliason and Jenny Karnehed, Göteborg University, wrote a master thesis about techniques for system development. The authors describe how to use the V-model (a number of steps describing the validation procedure of software applications) for software verification and validation. [EK99a]

Bordin Sapsomboon wrote a report as a fulfilment of the requirement for the degree of Doctor of Philosophy. The report describes use of software inspections. [Sap99a]

In Sweden a great number of companies engage university students to perform some kind of analysis or research to improve their software quality. These works can vary in many ways. Some works are done to evaluate a specific tool that companies would like to use for testing of their system. Other works focus on a more wide level where students should learn the working procedure of the company and then find methods for improving that job. Further it is not uncommon for students to make patterns of how and which testing tools should be used for best software quality improvement.

David Albertsson and Peter Sandberg did one such thesis at Mälardalens University, MDH for Ericsson. [AS05a]

Apart from the university world many companies have developed advanced tools for all kind of software quality analysis. These tools can perform a various tests on the source code. These tests may be:

- Source code checks, similar to the compiler but a higher level
- Memory violations checks
- Code coverage like statement and branch coverage
- Checks for improving the standard of source code
- Profiling analysis

A couple of these tools will be explained in chapters 7 Static Analysis Tools and 8 Dynamic Analysis Tools.

1.6 Thesis outline

The thesis begins with a discussion about Metrima and Metrima´s solution for broadband connectivity for metering applications. Chapter 2 describes the specific goals of this thesis.
We finish the chapter by making some hypothesis of the techniques we will use in the thesis. Chapter 3 describes the methods used to get our results. In chapter 4 one of these methods, Market Analysis, is described in detail. Chapters 5, 6, 7, 8 and 9 describes how methods and tools such as Code Standards, Code Inspections, Static Analysis Tools, Dynamic Analysis Tools and the Compilers can improve the quality of C code. The thesis is than completed with a discussion about our conclusions and what we think about the future code development at Metrima. This is described in chapter 10. All references are found in chapter 11.

1.7 System overview

C10v2 is a unit used for data collection through sensors placed within an apartment. The chapters 1.7.1 and 1.7.2 describes the functionality of this unit.

1.7.1 General description

Figure 1, below, shows the link between C10v2 sensors and the BrightHome centre. All data measured by the C10v2 is send via Ethernet to the BrightHome centre-server where it can be read by both janitor and the tenants. A similar figure is available at [Met05a].

![System overview](image)

**Figure 1.** System overview. The connection between C10v2 and server.

1.7.2 Detailed Platform Description

C10v2 is the main unit for data collection. The sensors, which are placed at selected places within the apartment, send its data to the C10v2 where it is calculated and forwarded to the BrightHome Centre.

**Target Platform C10v2**

C10v2 is used for collecting data through the sensor ports and saving the data for analysis. In Figure 2 different connection points are marked out and described in Table1.
**Figure 2. C10v2**

<table>
<thead>
<tr>
<th>Connection</th>
<th>Name</th>
<th>Signal</th>
<th>Input/output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>L</td>
<td>24VAC Line</td>
<td>In</td>
</tr>
<tr>
<td>2</td>
<td>N</td>
<td>24VAC Neutral</td>
<td>In</td>
</tr>
<tr>
<td>3</td>
<td>VDC+ Out</td>
<td>+6 VDC Out</td>
<td>Out</td>
</tr>
<tr>
<td>4</td>
<td>VDC- Out</td>
<td>-6 VDC Out</td>
<td>Out</td>
</tr>
<tr>
<td>5</td>
<td>RS232</td>
<td>+5VDC</td>
<td>In/Out</td>
</tr>
<tr>
<td>6</td>
<td>Ethernet</td>
<td>Ethernet 10BaseT</td>
<td>In/Out</td>
</tr>
<tr>
<td>11</td>
<td>12VDC</td>
<td>+12V</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>GND</td>
<td>GND</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>RS-485</td>
<td>RS-485A</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>RS-485</td>
<td>RS-485B</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>I/O</td>
<td>M-Bus+</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>I/O</td>
<td>M-Bus-</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>OneWire+</td>
<td>OneWire+ Temperature</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>OneWire-</td>
<td>OneWire- Temperature</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>F-</td>
<td>Fire detector -</td>
<td>In</td>
</tr>
<tr>
<td>20</td>
<td>F+</td>
<td>Fire detector +</td>
<td>Out</td>
</tr>
<tr>
<td>21</td>
<td>S04 -</td>
<td>Pulse -</td>
<td>In</td>
</tr>
<tr>
<td>22</td>
<td>S04 +</td>
<td>Pulse+</td>
<td>In</td>
</tr>
<tr>
<td>23</td>
<td>S03 -</td>
<td>Cold water (KV-)</td>
<td>In</td>
</tr>
<tr>
<td>24</td>
<td>S03 +</td>
<td>Cold water (KV+)</td>
<td>In</td>
</tr>
<tr>
<td>25</td>
<td>S02 -</td>
<td>Warm water (VV-)</td>
<td>In</td>
</tr>
<tr>
<td>26</td>
<td>S02 +</td>
<td>Warm water (VV+)</td>
<td>In</td>
</tr>
<tr>
<td>27</td>
<td>S01 -</td>
<td>Electricity -</td>
<td>In</td>
</tr>
<tr>
<td>28</td>
<td>S01 +</td>
<td>Electricity +</td>
<td>In</td>
</tr>
<tr>
<td>29</td>
<td>GND</td>
<td>Alarm GND</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>12VDC</td>
<td>Alarm Plus</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>Alarm</td>
<td>Alarm direct</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>Alarm</td>
<td>Alarm delay</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>Siren</td>
<td>Siren Out -</td>
<td>Out</td>
</tr>
</tbody>
</table>
Table 1. C10v2 connection pins.

<table>
<thead>
<tr>
<th></th>
<th>Siren</th>
<th>Siren Out +</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>34</td>
<td>Moist</td>
<td>Moist-</td>
<td>In</td>
</tr>
<tr>
<td>35</td>
<td>Moist</td>
<td>Moist+</td>
<td>In</td>
</tr>
</tbody>
</table>

Ethernet dongle

The Ethernet dongle, Figure 3, is both a programming and debugging tool. The communication protocol over Ethernet, between the dongle and the host-PC is TCP/IP. In the IDE provided by Ubicom, configuration information for the dongle is given. [Ubi05d]

![Figure 3. The reprogramming dongle and an over layer figure of the target.](image)

By placing the cord on the reprogramming pins in the figure above and connecting the dongle to Ethernet, the target can easily be reprogrammed and debugged.

D80 Display

C10v2 is usually connected to a display called D80, see Figure 4. This product was developed to give the tenants the possibility to control their consumption. This display contains an installation program which is used to configure and verify the communication with C10v2 and all the pulse sensors. [D80]

![Figure 4. Display D80.](image)
Ubicomp ip2022

C10v2 has an Ubicomp ip2022 CPU as a main and communication processor. The ip2022 chip gives access to basic hardware, with complements from on-chip software (SDK) [Ubi05a]. Ubicomp ip2022 implement an enhanced Harvard architecture with independent 16 bit address and 8-bit dual-port data buses. An advantage of this architecture is that the current instruction can be executed while next instruction is fetched from the program memory. It is equipped with built-in Ethernet interface, 8 interrupt pins, 8 analogue and digital pins. [Ubi05b] Ip2022 is capable of a speed up to 160 MHz but is set on 30 MHz in C10v2. [C10v2] Figure 5 below is a block-diagram system overview, which illustrates how the different ip-modules are connected to the ip2022 through the OS.

![Ubicomp ip2022 block-diagram](image)

**Figure 5. Ubicomp ip2k chip.**

**ipOS**

Ubicomp’s operating system, ipOS, offers functionality for TCP/IP, timers, interrupt handling, hardware- and software UART’s, I/O-handling etc. In the C10v2 project the ipOS operates as a single task i.e. it does one thing at a time [ipOSM]. This configuration is made by turning off the multitasking features in the Configuration tool. In single-task mode each physical interface and some operating system functions need to be polled. I/O drivers expose polling functions to be used for this function. The S0 pulses, power failure, smoke detector pulses, serial communication and Ethernet communication are detected with interrupt signals.

**Real time clock (RTC) and data logging**

The C10v2 holds an external RTC which generates an interrupt every minute. This interrupt is used to handle jobs that should be performed every, minute, hour, day, month and year. Every hour values measured for the temperature, hot water, cold water and electricity are stored in a ring buffer. Values for 35 days are stored in the ring buffer.
Alarms

There is an alarm queue that stores the twelve most recent alarms. An alarm does not necessary need to be caused by an error; it could also be a message that tells the upper management system that a change is made. An alarm is sent when a change is detected. For example, an alarm is sent if the C10v2 detected that the smoke detector does not work. The next time an alarm is sent is when the smoke detector is working.

The following seven alarms can be generated:
1. Smoke detector malfunction
2. Smoke detector has detected smoke
3. Power failure
4. Power outage
5. S0-channel 4 has changed state
6. Temperature sensor 1-5 disappeared
7. Alarm queue full

Communication

All communication is done by Metrima’s communication protocol, brightNet (bNet). A message can be up to 32 bytes long. The format of C10v2 messages is shown in Figure 6. The only check that is made on the serial channel, RS232, is the correctness of bNet-message length i.e. no CRC.

<table>
<thead>
<tr>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>31</td>
</tr>
<tr>
<td>Id</td>
</tr>
<tr>
<td>Len</td>
</tr>
<tr>
<td>Echo</td>
</tr>
<tr>
<td>Data</td>
</tr>
</tbody>
</table>

Figure 6. The format of the C10v2 message

Id = Message identifier
Len = Number of significant bytes in a message
Echo = Correct return-message check
Data = Data sent.

On-Chip Memory in Ip2022

The address map in Figure 7 is divided in following two parts:
1. Program memory is used for mapping the .elf file and for debugging analysis.
2. External memory can be accessed from the C code and is used to store log data.
1.7.3 Development Environment Description

This is a description of the development environment. In this chapter debugging possibilities are carefully described.

Unity IDE

The code is written in Unity which is a Windows based development environment created by Ubicom. Unity contains among other features an editor, support for handling projects, a module configuration tool and a complete GNU-based development environment with functionality for assembling, compiling, debugging and linking.

Unity makes GNU compiler GCC easier to use by hiding GCC behind a simple interface, but it also hides GCC’s power and flexibility.

Software Development Kit (SDK) and ipModule Configuration Tool

SDK is arranged in modules, which are collections of related functionalities. The ipModule is managed by the Ubicom’s configuration tool, see Figure 9.

Compilation chain

Ip2k-elf-gcc is the complier used by Unity for compiling the code. The files are linked together with a linker called ip2k-elf-ld, see Figure 8. Compilation chain.
Configuration Tool

With the ip2k configuration tool, see Figure 9, the software can be configured specifically for every project [Ubi05d]. The configuration tool helps programmers work with modules and configuration points (ipOS, ipStack), shown in Figure 9. Configuration Points are elements of functionality within a package that can be configured. Examples of configuration points might be whether to use best-fit or first-fit algorithm in the operating system ipOS, or which pins to use for a UART.
The Configuration tool generates config.mk and config.h files, see Figure 10. These files contain the macro definitions necessary to control the modules and all option settings during compilation. [Ubi05c]

The .c_c file is used by Unity to keep information about the project, like names of the source files, debugger settings, display fonts, etc. Configurations for the selected modules are set in the .lpj file and can be modified with the Config Tool in Unity.

**Summary of files**
The compiler creates an object file for each .c and .s source file. The linker, (ip2k-elf-ld), steps through all these files and then creates an executable file. Table 2 and Table 3 show a short description of the files in the project.

<table>
<thead>
<tr>
<th>File Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.c_c</td>
<td>Project file used by Unity to store specific information about the project.</td>
</tr>
<tr>
<td>.elf</td>
<td>Executable binary file. Used by the micro CPU.</td>
</tr>
<tr>
<td>.lpj</td>
<td>Project file for the Configuration Tool. Used to store project-specific information.</td>
</tr>
<tr>
<td>.o</td>
<td>Object file created by the compiler.</td>
</tr>
</tbody>
</table>

Table 2. Project specific files.

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Config.h</td>
<td>Contains configuration information generated by the Configuration Tool for C header files.</td>
</tr>
<tr>
<td>Config.mk</td>
<td>Contains configuration information generated by the Configuration Tool for the makefile.</td>
</tr>
<tr>
<td>Makefile.inc</td>
<td>Compiling directions for files referenced in Makefile.gen</td>
</tr>
<tr>
<td>Makefile.gen</td>
<td>Contains a list of all project files have to be compiled</td>
</tr>
<tr>
<td>ip2kelf.ld</td>
<td>Linker configuration with memory section addresses.</td>
</tr>
</tbody>
</table>

Table 3. Ubicom environment configuration files.

**Debugging with Unity**

A debugging session can be started when an executable file and a target are available. The debugging takes place on the target while the program is running. All the regular debugging tools are available and are shown in the tool bar, see Figure 11.

**Debugging tools**

Some of the tools available in Unity used for debugging are:
- **Register view**, used to view and edit the contents of the CPU and peripheral registers.
- **Memory view**, used to view the contents of memory. Two different memory views are available (Memory 1 and Memory 2 Window) for different sections of the memory.
- **Breakpoints**, gives ability to halt the program execution.
- **Watchers**, registers and user-specified expressions are shown with their values and their memory addresses, see Figure 12.
- **Stack Window**, Figure 13, a read only window where a list of subroutines on the hardware call/return stack is shown.
**Disassembler**, Figure 14, assembly language instructions and the memory section addresses are shown corresponding to the source code.

![Figure 12. Watch window in Unity](image)

![Figure 13. Stack window in Unity.](image)

**Figure 12. Watch window in Unity**

**Figure 13. Stack window in Unity.**

By adding a variable to the Watcher and putting a breakpoint at a specific line where the variable is being used, the execution instruction for this line can be seen on an assembler level in the Disassembler.

![Figure 14. Disassembler window in Unity.](image)

**Figure 14. Disassembler window in Unity.**
2 Problem Description

The assignment was to find general code quality improvement methods that can be used at Metrima and to create a pattern for the developers when writing and maintaining C-code. Beside our own research about the subject our supervisor at Metrima had own proposals for achieving quality improvement that should be examined. Together we selected the following subjects for this thesis.

2.1 Static analysis

Code standards

A code standard similar to the K&R is used by the developers of C10v2 when the software was written. The purpose of this chapter was to review this standard and complete it if necessary. How well the code standard has been used by the developers of C10v2 system was also inspected.

Code inspections

The methods for code inspections were examined. A complete check-list for manual code inspection should be made.

Static analysis tools

A static test tool used at Metrima is SPliint. The option-flags for this tool need to be verified. Is this the best tool for this problem? What other tools are usually used for this kind of problems and how effective are those?

The compiler

The compiler used in this project is a GCC-compiler with standard options. The capability and ability to influence the compiler needs to be tested. Other compilers should be tested and whether this GCC-compiler is the best suited solution or not has to be examined.

2.2 Dynamic analysis

Automatic Test system

An automatic test system is used to automate manual tests. Available test tools at Metrima should be examined and other test tools examined during the research should be described.

Code coverage and Profiling
No code coverage or profiling testing has been made for the C10v2 system. Different techniques, suited for this target had to be tested. Enough information had to be gathered to find out whether these techniques could be used in this project.
3 Methodology

From early stage of our research we realized that software quality is a very wide subject. Due to the extent of the subject we had to make some delimitations in this thesis. In collaboration with our supervisor we decided to use the following working methods to reach our goal.

3.1 Market Analysis

A market analysis will give us a wider perspective on how other companies work to improve and maintain good code quality. This was done by interviewing Christer Persson at ABB, Software Architecture and Processes and Sigrid Eldh, verification expert at Ericsson.

3.2 Literature study

During this thesis we have read a number of books regarding software development, software quality improvement and C-code quality improvement. The literature is used to find the most suitable software quality improvement methods for C10v2.

3.3 Dialog with the employes at Metrima

At Metrima there are many experienced developers and designers. These resources have been big help when during this thesis.

3.4 Internet research

The information for a number of the subjects that have been discussed in this thesis is found on the Internet. Some of these documents are science papers. Other documents are manuals for different tools tested during this thesis.

3.5 Bug report database

The bug report database, MMK, used at Metrima is bought in from Vendel Data AB [VDAB]. This database contains the bugs found during the testing of C10v2. The bugs reported are both hardware and software related. Together with our supervisor at Metrima we reviewed a number of these bugs. Many of these bugs are solved by using more defensive programming i.e. extra controls ensuring that incoming data and messages are correct. The code quality improvement recommendations for the C10v2 are partly based on these solutions.
4 Market Analysis

Companies have different ways of improving code quality since they have different view of what good quality is. There are different standards to measure quality, e.g. TPI and CMM. The market analysis was made to understand how other companies, ABB and Ericsson, work to get and maintain good code quality.

4.1 ABB

Christer Persson, works as Software Architecture and Processes, at ABB. Persson and his team of six co-workers currently work with an ABB department and investigate the maturity of each department, measured with the CMM model.

Persson explained with the Figure 15 below how the software developing departments at ABB should work to reach good quality.

*Figure 15. Development procedure for software developers at ABB.*

The code needs to be inspected in three different stages and pass all this stages to be ready for release.

1. Unit testing:
   In this stage the code in the different modules is tested by the programmers. This can be done in different ways.
   - With tools e.g. Lint, PC-Lint.
- Different checklists. In chapter 6 Code Inspection one of these checklists can be found.
- Working in pair

2. Automatic testing:
   When the module has been checked by the methods mentioned above, a build of the program is done for the automatic tests. The tests are performed in IBM Rational test suite. This test system hold 55 different test suites each containing 5-10 tests. The program has to pass these tests before entering next stage.

3. If the program does not pass these tests the programmers have to find the cause and solve the “problem”. This goes on, till the program passes all the tests.

4. Manual testing:
   In this stage the program will be tested manually by a group of testers. The functionality that has been added is tested in different ways:
   - The tester gets a description that leads him/her through the new function step by step where the wanted result is clearly stated.
   - Try to get the program to crash.

5. If the program does not meet the expected quality, the programmers have to make necessary corrections.

4.2 Ericsson

Sigrid Eldh works at Ericsson as verification expert. Eldh spends her time researching, writing books and travelling around the world to different departments of Ericsson and give lectures on how testing should be performed on different type of products.

Eldh is a founding member of SAST (Swedish Association of Software Testing), founder of ASTA (Australian Software Testing Association) and chair of the Swedish Board for Software Testing.

Sigrid Eldh could not give any specific details on how Ericsson work with testing and code quality control since different departments have different ways of testing. But the recommendations and advices from Eldh clarified how a company should work to reach their intentions.

A successfully application development department requires a good test unit. This test unit is in charge of the testing. The size of this unit depends on number of programmers. Programmers tend to lose their ability to find bugs in code that they have worked with for a long time. To prevent this, it is necessary for programmers to have been in a test unit. This gives the programmers a chance to learn how to write readable and testable code. Another way to solve this problem is to work in pairs and rotate job often. When one is writing the code, the other is there to control and give advice. “Four eyes sees better than two”.

By rotating jobs the code can always be criticized with “new eyes” and all the programmers are familiar with the whole code.
Eldh considered the following points to be important when code quality is inspected in a company.

- For how long time has the programmers worked with this code, is it being seen with new eyes?
- Has the whole system ever been tested completely?
- Has any black box testing been done on the system?
- Who have written the test cases, the programmers or the test unit?
  - Which technique have been used to create the test cases?
  - How often is the system tested with this technique?
- Are there any negative tests included in the test system?

It is important for companies to learn from their mistakes and to use these experiences in the future. The desirable result of these experiences would be new test cases, new methods for testing and updated checklists.

**New test cases**

A system for writing test cases that Eldh strongly recommended was to draw state diagrams (state transition testing). This system can be used at any level of testing e.g. code level, function level and it is a relatively easy way to reach high test coverage. It is also a good way to keep things organized and understandable for outsiders. If needed, any state can be divided in more accurate states, which lead to more accurate test cases, see Figure 16. Testing a full system is impossible. By using this technique you can be more selective and chose the states that are considered to be in largest need of testing, as long as you know what is tested and what is not.

![Figure 16. State transition testing.](image)

**New methods for testing**

Which level of testing can be performed in a company depends on how mature they are for testing. The programmers and the testers have to feel that the methods used are suited for the job. Test software used at Ericsson includes:

- CTC
- Lint
- Rational Test RealTime
When none of above programs works with an Ericsson application, developers must create own simulators and test programs in order to test the software.

Updated checklists

A checklist has to be kept updated; every time a bug seems to be recurring a new point has to be added in the checklist. This does not necessary mean that the checklist gets bigger after each inspection.

4.3 Difference between Ericsson and ABB

Despite the wide selection of products within each company the tools and techniques for quality control are similar. Since there are so many departments at both Ericsson and ABB most of the tools are well recognized by the persons interviewed, even if the techniques used for each product varies a lot.
5 Code standards

This is a summary of the code standards used at Metrima today and how this subject is followed at other companies. During our research and market analysis we found out that a code standard has several advantages in code quality improvement:

- Makes code easier to inspect for any person.
- Easier to get feedback if the developer is known.
- Reduces the number of errors made by a mistake.
- Easier to find relevant information if the code has a well defined structure.
- Encouraging developers to do the right things.
- Faster application development (less errors -> less rework).

As in chapter 6 Code inspections the goal of this chapter is to clarify Metrima’s situation today and use of code standards in other companies by discussing following:

- A study about code standards.
- Which quality methods are used at Metrima today?
- Which methods are used at other companies?
- New model suited for the C10v2.

5.1 A study about Code standards

Writing a code that is easy to read, easy to maintain and efficient is difficult. More difficult is to comply with available standards. In the C10v2 case we showed exactly this behaviour. Complying with standards requires experience as everything else in software development. During our research about coding standards we saw a number of different standards. The truth is that there is no true coding standard that will be appropriate for every company. We believe that it is better to use a simple standard that will increase maintainability and readability of the code and that it is followed by everyone in the project instead of using an accepted standard that might be too complicated to use. There are three simple rules that should be followed by every programmer whose intention is to increase maintainability and readability of his/her code. These rules are general. Later in this chapter we will show a number of specific programming rules, see section 5.1.4 Programming Rules. These general rules are [CÅAB04a]:

- **Be clear**
  Express your self in such a way that it is easy to understand the code.

- **Be consistent**
  Consistent way of writing code and comments increases readability.

- **Be abstract**
  Focus on the task to be done. This makes the code easier to maintain as it probably only affects limited parts of code.

Following examples are a mix of several coding standard recommendations that are used across universities in Sweden but also writings based on ISO standards from companies like PRQA [PRQA].
5.1.1 Header files

Header files should be well commented and have format described below, see Figure 17:

- Copyright and the name of developer
- Description of the file
- #ifdef
- #include:s
- #define:s
- other #ifdef
- typedef:s
- function declarations
- #endif

```c
/******************************************************************************
 * Copyright 2006 Metrima AB  
 *  
 * Carl Falk  
 *  
 * This file contain...  
 *  
 ******************************************************************************/

#define EXAMPLEL_H
#define EXAMPEL_H
#include <test.h>  /*this file contain...*/

#define NDEBUGING  /*#define TESTING*/  /*compile code...*/

#define TRACE(ignore) ((void) 0)  /* do not compile macro */
#endif

#define TESTING
#define TRACE(line) (arr[line] = 1)  /* TESTING macro*/
#endif

void handleReceivedMessage(u8_t, u8_t *);  /*this function should handle...*/
#endif  /* EXAMPLEL_H */
```

Figure 17. Example on h-file.

5.1.2 C files

C files should have the following format, see Figure 18:

- Copyright
- Description of the file
#include:
#define:
module-specific definitions
Function definitions (function description should not be changed during code updates). In this case it is better to have an extra line Update descr./date/name where the latest change is described. If all update history is wanted then it should be in a separate file.

```c
/**************************************************************************
 * Copyright 2006 Metrima AB
 * *  
 * This file...
 * *
***************************************************************************/

#include <test.h>
#include <test2.h>
#include <test3.h>

#define MAX 5

int arr[5];  /*this array....*/

/**************************************************************************
 * Function : (Description)  *
 * Input : u8 t... (Description)  *
 * *  
 * Return : void  *
 * Update descr/date/name : updates made in.../060101/Carl Falk  *
***************************************************************************/

void handleReceivedMessage(u8_t, u8_t*)
{
    /*code_...*/
}
```

Figure 18. Example on C file.

## 5.1.3 Indent style

Within software development (programming) an indent style is a convention developed to govern the application’s structure. This convention considerably improves the maintainability and readability of the code. There are several indent styles that differs very little from each other. In this chapter we will present two most commonly used styles:

- K&R
- BSD

### K&R

K&R style originate from Kernighan and Ritchie’s book *The C Programming Language*. Users of K&R style claim that the real advantage of this style is readability. This statement is based by the fact that the code is grouped when using K&R resulting in more code visible at
once. An example of K&R style is compared with the style used at Metrima which is a slightly modified K&R style, see Table 4.

<table>
<thead>
<tr>
<th>K&amp;R</th>
<th>Metrima</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (arr_length &lt;= SIZE) {</td>
<td>if(arr_length &lt;= SIZE){</td>
</tr>
<tr>
<td>read(arr, arr_length);</td>
<td>read(arr, arr_length);</td>
</tr>
<tr>
<td>arr_length++;</td>
<td>arr_length++;</td>
</tr>
<tr>
<td>} else {</td>
<td>} else{</td>
</tr>
<tr>
<td>arr_length = 0;</td>
<td>arr_length = 0;</td>
</tr>
<tr>
<td>read(arr, arr_length);</td>
<td>read(arr, arr_length);</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
</tbody>
</table>

Table 4. Comparison between K&R and Metrima code writing standard.

Disadvantage of this style is that the beginning brace can be difficult to find.

**BSD**

The BSD (Berkeley Software Distribution) style originates from the University of California at Berkeley. Likewise K&R style the BSD style is very common style in computer programming. A comparison between BSD and K&R style is shown in Table 5.

The advantages of this style appear first when the code increases in size and complexity. Example in Table 5 demonstrates the difference in writing “{“ between BSD and K&R style; compare it with the example in Table 4. Another difference is the space after the if-statement.

<table>
<thead>
<tr>
<th>BSD</th>
<th>K&amp;R</th>
</tr>
</thead>
<tbody>
<tr>
<td>if(arr_length &lt;= SIZE)</td>
<td>if (arr_length &lt;= SIZE) {</td>
</tr>
<tr>
<td>{</td>
<td>read(arr, arr_length);</td>
</tr>
<tr>
<td>read(arr, arr_length);</td>
<td>arr_length++;</td>
</tr>
<tr>
<td>arr_length++;</td>
<td>} else {</td>
</tr>
<tr>
<td>}</td>
<td>arr_length = 0;</td>
</tr>
<tr>
<td>else</td>
<td>read(arr, arr_length);</td>
</tr>
<tr>
<td>{</td>
<td>}</td>
</tr>
</tbody>
</table>

Table 5. Comparison between BSD and K&R style.

**5.1.4 Programming Rules**

**Documentation**

The code should be implicit documented in the form of meaningful names for variables and functions. In own defined header-files all function declarations and variables should have a comment. In the corresponding c-file every function should begin with a function-header as shown in chapter 5.1.2 C files. By following this rule the code maintainability is considerably increased.
Naming convention

The choice of a meaningful name of a variable, function, structure etc. is important factor for increasing code understandability, readability, maintainability and traceability from requirements to source code [MHEN97b]. The aim of this section is to highlight this importance. Programmers of the C10v2 use an own defined naming convention that is used most of the time. An example of naming convention described in Metrima’s code standard is:

INCORRECT: con_spr_sys = true;
CORRECT: connection_to_superior_system_is_up = true;

Boolean-, NULL- and Zero values

Expressions using NULL has different meaning (semantic) than expressions using zero or booleans. For that reason it is important to explicitly compare the values with the right “truth” type. Always check function returning values and pointers for their correctness, see Table 6.

<table>
<thead>
<tr>
<th>Boolean</th>
<th>Zero</th>
<th>NULL</th>
</tr>
</thead>
<tbody>
<tr>
<td>#define TRUE 1</td>
<td>int tmp;</td>
<td>int *tmp = NULL;</td>
</tr>
<tr>
<td>#define FALSE 0</td>
<td>tmp = myFunc(...);</td>
<td>int x = 2;</td>
</tr>
<tr>
<td></td>
<td>if( tmp == 0 )</td>
<td>tmp = &amp;x;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>if(tmp != NULL)</td>
</tr>
<tr>
<td></td>
<td>tmp = myFunc(...);</td>
<td>myFunc(tmp);</td>
</tr>
<tr>
<td></td>
<td>if( tmp == TRUE )</td>
<td></td>
</tr>
</tbody>
</table>

Table 6. Checking boolean, null and zero values.

Macros

When using macros it is very important to use parentheses to separate the parameters. Consider following example: [CAAB04b]

INCORRECT:

```c
#define MYMACRO(x, y) x*y
tmp = MYMACRO(a + b, c - d) * t
/*This expands to*/
tmp = a + b * c - d * t;  /*WRONG*/
```

CORRECT:

```c
#define MYMACRO(x, y) ((x) * (y))
tmp = MYMACRO(a + b, c - d) * t
/*This expands to*/
tmp = ((a + b) * (c - d)) * t;  /*RIGHT*/
```

Arithmetic computations

C10v2 code is full of arithmetic computations. For that reason it is important that the code is well defined. The use of ( ), parentheses, increases the readability of such a code. This habit
also reduces the number of errors made by mistake. Below is one example from the file onewire.c in C10v2 application that would be easier to inspect if parentheses were used.

    for (i = 0; i < NO_OF_TEMPERATURE_SENSORS; i++)
        EEPROM_writeString((ONE_WIRE_ADDRESSES_OFFSET + i * 8), wire_id, 8);

Question that arises when this code is inspected is, does the programmer mean:

    EEPROM_writeString(ONE_WIRE_ADDRESSES_OFFSET + (i * 8)), wire_id, 8);

or

    EEPROM_writeString((((ONE_WIRE_ADDRESSES_OFFSET + i) * 8), wire_id, 8);

Initialisation

Always give the variables an initial value at the line where it is first declared. This reduces the risk to undefined behaviour. This rule should be applied on all types including pointers and booleans.

Global variables

In C10v2 project global variables are frequently used. This kind of programming can cause unwanted behaviour and should be avoided as much as possible. In this section we will describe some of the issues caused by using global variables. An alternative way to solve these problems is also presented.

Issue

- By using global variables unnecessary limitations are introduced in form of allowing equally named variable several times. This can lead to using a variable that have a different value then is expected.

Solution

- Use prefixing when declaring global variables in order to distinguish between global and local variables. This measure prevents that wrong variables are used by accident. With prefixing we mean that all global variables begin with a specified prefix. For example:

    int counter; /*Global variable declaration at Metrima*/
    int g_counter; /*The prefix g_ show that this variable is a global variable*/

- Use static variables if the variable should only be used within the file where the variable is declared (these variables are called extern static variables). To be able to use a static variable within the whole file it must be declared outside any function or else it only can be used by the function where the variable was declared.

Syntax

Programmers should in best manner avoid putting multiple instructions on the same line. This style of programming makes any inspector of the code confused since it is not clear what should be returned. One such example, however very simple one, within the C10v2 application is the code below. This code is found in init.c file, line 101.
if( (randomWait*=2) > 16){
    randomWait = 16; // 16 * 2h max waiting time
}

According to [FredRa] such a code should be rewritten to:

randomWait *= 2;
if( randomWait  > 16 ){
    randomWait = 16; // 16 * 2h max waiting time
}

A good rule is to never use “=” inside an if-statement. The same issue is shown in the following example. This code is found in the taskManager.c file.

if( lastLinkState != 255)
  if ( lastLinkState != read_pin(eth0__S1_IPETHERNET_LINK_LED_PORT, eth0__S1_IPETHERNET_LINK_LED_LED)) {
    lastLinkState = read_pin(eth0__S1_IPETHERNET_LINK_LED_PORT, eth0__S1_IPETHERNET_LINK_LED_LED);
    linkCnt++;
  }
}

To increase the readability of the program this code should be rewritten as:

if( lastLinkState != 255){
  tmp = read_pin(eth0__S1_IPETHERNET_LINK_LED_PORT, eth0__S1_IPETHERNET_LINK_LED_LED);
  if ( lastLinkState != tmp) {
    lastLinkState = tmp;
    linkCnt++;
  }
}

Rewriting the code above does not only increases the readability, it also improves the functionality since less function calls are made.

5.2 Which quality methods are used at Metrima today

Programmers of the C10v2 code have made their own standard of how to write their C code. A comparison between two files, taskManager.c and powerManagement.c shows that there are differences in how this standard is used. Even if there are no big differences the code is still not uniform. The use of this standard are shown in tables 7-15.

5.2.1 C10v2 code standard

Upper/lower-case letters
1. Constants should be written with upper-case letters. An underscore should separate two words building one constant.
Example: \texttt{MEGA\_KONSTANT}

2. \#defines should be written with upper-case letters. An underscore should separate two words building one define constant.
   Example: \texttt{MEGA\_DEFINITION}

3. Enumerated statements should be written with upper-case letters.
   Example: \texttt{STATES}

4. Type definitions should begin with upper-case letter.
   Example: \texttt{Mytype}

5. Variables should be written with lower-case letters. An underscore should separate two words building one variable.
   Example: \texttt{my\_var}

6. Function names should have following format.
   Example: \texttt{mySuperFunction();}

7. Exceptions are made for some special functions dealing with hardware.
   Example: \texttt{EEPROM\_writeByte();}

\textbf{Other recommendations}

1. Functions should have comments about what they are intended to do and what should be returned.

2. Functions should have a comment that shows the last time this function was inspected. If some changes are made within the functions this comment must be changed too.

3. It must be a blank line between variable declarations and the code.

4. Operators must be separated by a space.
   Example:
   \begin{verbatim}
   CORRECT: a = b + c;
   INCORRECT: a=b+c;
   \end{verbatim}

5. if, while, for, do, switch statements should have following format:
   \begin{verbatim}
   while(my\_char < SIZE) {
   i++;
   my\_char += KONSTANT;
   }
   \end{verbatim}

6. Tabs should be three spaces large. Tab button should not be used.

7. Function declarations should be in respective header-files.

8. Use clear names when declaring variable and function names.
Example:

CORRECT: connection_to_superior_system_is_up = true;
INCORRECT: con_spr_sys = true;

9. Use spaces after “,” and “;”
Example: int a, b, c; for(i = 0; i < 10; i++)

10. Write new line after every statement.
CORRECT
void foo()
{
    int i;
    char j;
}
INCORRECT
void foo()
{
    int i; char j;
}

11. Use blank line after every case in a switch-case.
Example:
switch (a) {
    1:
        nop;
        break;
    2:
        nop;
        break;
    default:
        break;
}

Example:

The example below is intentionally not translated to English. We prefer not to translate it because it better illustrates Metrima’s situation today.

C file

//filnamn: super.c
//datum: 2000-05-19
//beskrivning: filen ämnar demonstrera textkonventionen
//Skapad av: KB
#include "super.h"

char my_char; //Global byte

//Funktion: räknar....bla bla
//Parametrar: char my_char - antal loopar..
//Return: räknat värde...
char mySuperFunction(char my_char) 
{
    char i;
    char pajas;

    while (my_char < SIZE) {
        i++;
        my_char += KONSTANT;
    }

    switch (pajas) {
    pajas1: 
        nop;
        break;

    pajas2: 
        nop;
        break;

    default: 
        break;
    }

    return(i);
}

Header file

#ifndef __super__
#define __super__
#define SIZE 255

typedef struct _Mystruct { 
    char value1;
    int  value2;
} Mystruct;

enum STATES {ON, OFF};
const char KONSTANT = 1;

char mySuperFunction(char my_char);
#endif

According to Metrimas code standard

#ifndef __super__
#define __super__
#define SIZE 255

typedef struct _Mystruct { 
    char value1;
    int  value2;
} Mystruct;

enum STATES {ON, OFF};
const char KONSTANT = 1;

char mySuperFunction(char my_char);
#endif

Table 7. File header according to Metrimas code standard.

taskManager.c

/*
 * taskManager.c
 * Utför diverse bakgrundsaktiviteter
 */
Table 8. Use of file header in `taskManager.c`.

```c
/powerManagement.c
/*
 * powerManagement.c
 * Håller koll på powerdown-mode och powerfunktioner
 * Återställer värden på räknare. Om noden har krashat så tas värdena från
 * senaste timövergången med
 * fom ver 1.6.7 så används 10 minuters backup värdena för att återställa efter
 * krash.
 * */
```

Table 9. Use of file header in `powerManagement.c`.

According to Metrimas code standard

```c
//Funktion:   räknar....bla bla
//Parametrar: char my_char   - antal loopar..
//Return:     räknat värde...
//Granskad:   2003-03-04 av KB
char mySuperFunction(char my_char)
{
 // the function does not end here
```

Table 10. Function header according to Metrimas code standard.

```c
taskManager.c
// Gör det som ska göras varje ny dag
void dayTasks()
{
    EEPROM_writeByte(LAST_PASSED_DAY_OFFSET, &last_taskmanager_day);
}
```

Table 11. Use of function header in `taskManager.c`.

```c
/powerManagement.c
// Skriv ner till EEPROM: Orsak till omstart, Timvärden för S0 och temperatur,
// Senast var timpekaren pekar, tid & datum för nedgång
void powerDown()
{
 // the function does not end here
```

Table 12. Use of function header in `powerManagement.c`.

According to Metrimas code standard

```c
while(my_char < SIZE) {
    i++;
    my_char += KONSTANT;
}
```

Table 13. While-loop according to Metrimas code standard.

```c
taskManager.c
Not used in this file.
```

Table 14. While loops are not used in `taskManager.c`.

```c
/powerManagement.c (writeLastMainFrameDownData)
i = 5;
```
while (i--)
    average_temperature[i] = hour_temperatures[i].....
    memcpy(&data[1+8+16], (u8_t *)average_temperature, 10);

Table 15. Use of while loop in powerManagement.c.

5.3 Which methods are used at other companies

This section covers our market analysis about code conventions that are used at ABB and Ericsson.

ABB

Regarding ABB there is not much information gathered during the interviews about code standards. We can still make some assumptions with the complementing e-mails we got from ABB. Once again these answers are not a general ABB standard. It only explains what is done at the departments we analysed. According to Christer Persson a company need to have all kind of standards, not only code standards, to reach level five in the CMM. This particular department was on its way to CMM level 2. This, indirectly, means that there got to be some kind of code standard used at the ABB. An explanation why we lack this information is that we talked to a person that is on a higher department level and therefore did not know all system and testing details. There are also standards written by the programmers themselves, likewise Metrima’s standard, which is not mandatory.

Ericsson

The information about Ericsson code standards came from a prior Ericsson employee and present professor at the KTH, Kungliga Tekniska Högskolan, Ingmar Olsson. The recommendations that we present below are all included in a book written at the Ellemtel, a company owned by Ericsson at that time. [MHEN97a]. Some of the rules are:

Naming

- Use meaningful names.
  “stringLength” instead of “strL”
- Use English names for identifiers.
  “Car” instead of “bil”
- Be consistent when naming functions, types, variables and constants.

Organizing the code

- Each header file should be self-contained. It should group type definition, declarations and macros.
- Avoid unnecessary inclusion. (#include)
- Enclose all code in header files within include guards (#ifdef.)

Comments

- Each file should contain a copyright comment.
- Each file should contain a comment with a short description of the file content.
Every file should declare a local constant string that identifies the file.  
Use // for comments.  
All comments should be written in English.

**Rules and Recommendations**

- Do not change a loop variable inside a for-loop block.  
- Update loop variables close to where the loop-condition is specified.  
- All flow control primitives (if, else, while, for, do, switch and case) should be followed by a block, even if it is empty.  
- Statements following a case label should be terminated by a statement that exits the switch statement.  
- All switch statements should have a default clause.  
- Use break and continue instead of goto.  
- Do not have too complex functions.  
- Prefer explicit to implicit type conversions.  
- Do not let assertions change the state of the program.  
- Remove all assertions from production code.  
- Check for all errors reported from functions.  
- Avoid duplicated code and data.  
- Make non-portable code easy to find and replace.  
  
```
#ifdef INT32
typedef int sint32;
#else
typedef long sint32;
#endif
```
- Do not mix coding styles within a group. For each project, or group of closely related classes, you should select a coding style. Code written by one programmer might be maintained by another.
- In names that consist of more than one word, the words are written together and each word that follows the first begins with an uppercase letter.
  
```
int max_timeout_time = 1000;      // Not recommended  
int maxTimeOutTime = 1000;        // Recommended  
```
- The names of variables and functions should begin with a lower-case letter.  
- Do not use characters that can be mistaken for digits, and vice versa.  
- The names of macros should be in uppercase.  
  
```
#define SQUARE(x) (x)*(x)      /* Recommended */
```

**5.4 New model suited for the C10v2**

The recommendations in this chapter are made on basis of all three above sections. The largest problem regarding Metrima’s code standard is not how it is written. This standard would increase readability of the code if it were used by all programmers all the time. The issue here is how to comply with this standard? In the tables 7-9, 10-12 and 13-15 we compared the actual code of the C10v2 with the Metrima’s own standard. Almost all of these examples show how this standard is misused. The result of this could be decreased readability of the code.
Because the native programming style (indent style) of the C10v2 application is K&R (see section 5.1.3 Indent style) it is not necessary to rewrite this code to another style. What is necessary is to make sure that this style is always followed.

Our recommendations are:

- Write all h-files as shown in section 5.1.1 Header files.

- Write all c-files as shown in section 5.1.2 C files.

- Use K&R style. Make sure to write { } even at those lines that only have one line of code. This is defensive programming. In the future the programmer might add an extra line under the if- or else-statement and cause unwanted behaviour because the brackets were not used where they should.

- Document all code. Every function must have a comment on what it does. See example in section 5.1.2 C files.

- Use Metrima’s variable name convention. See section Naming convention in chapter 5.1.4 Programming Rules.

- Always use defensive programming. See section Boolean-, NULL- and Zero values in chapter 5.1.4 Programming Rules.

- If macros are used then they must be written with parenthesis. See section Macros in chapter 5.1.4 Programming Rules.

- Always use ( ) to clarify what the code should do. This is very important since the precedence of operators might cause unwanted behaviour.

- Always give the variables an initial value at the line where it is first declared. Making this to a habit reduces the risk of unwanted behaviour, especially when using pointers.

- If possible avoid global variables. This increases the risk of dependence and naming issues between the modules.

- Write your code as simple as possible. Do not use nested operations on the same line. It makes the code very complex and difficult to inspect. See section Syntax in chapter 5.1.4 Programming Rules.
6 Code Inspections

This chapter contains a summary of the code inspection techniques investigated. Code inspection is a formal testing technique developed to review source code alone or within a group. An inspection is performed by asking questions about the program source code and analyzing the code with a checklist. The goal with this chapter is to describe the following:

- A study about Code inspections
- What code inspection methods are used at Metrima today
- What code inspection methods are used at other companies
- A new model suited for the C10v2

6.1 A study about Code inspections

This part is about techniques that can be used by the C10v2 programmers when writing and checking their code. All of the below listed methods comes from the books about software quality assurance or articles from the academic world. Using these methods a simple inspection model for C10v2 code quality improvement is made. The model is based on the techniques that are in some manner applicable on the C10v2 code. Requirements and Design parts are to be checked during the Inspection analysis. Techniques below the Static analysis box, in Figure 19, are most easily applicable on the C10v2 code. By using the proposed techniques difficulties experienced when integrating Ubicom with other dynamic tools are avoided.

![Figure 19. Model for C10v2 code quality improvement.](image-url)
6.1.1 Code reading techniques

Code reading techniques are step-by-step procedures performed by inspectors to uncover software defects. [Tian05b] There are several different reading techniques. Below are two of those techniques we would recommend to use for the C10v2 code. Our recommendation would be to use both of these methods regularly and to make these inspections at least once before any release. Our recommendations are based on difficulties involved to apply the different techniques. A technique that is difficult to apply will probably never be used.

The difference between the ad-hoc and check-list method is that the ad-hoc method is very simple and does not need that much or none preparation. It is preferable that the programmers are familiar with the application because they then can find defects at a higher abstraction level of the code. The check-list method is stricter. The programmers of the application do not necessarily need to do this inspection. However, the inspectors are responsible that all points in the checklist are checked.

Ad-hoc

This is the simplest way of code reading. The inspectors are left to inspect the code by their own experience with the help of the code specification. The advantage of this technique is that experienced inspectors are free to use their knowledge without having some overhead technique that limits their work.

Check-list

Checklists are used since Fagan’s software inspection technique was introduced, see chapter 6.1.2 Software Inspection below. It has a somewhat different approach compared to the ad-hoc technique. The inspectors here are guided through the code. They are given a series of specific questions that are intended to focus the inspector towards common code defects. Below is a general checklist written by the [WH95] and [SR97b]. A C10v2 specific check-list can be found in chapter 6.4 New model suited for the C10v2.

General issues
- Will each loop terminate?
- Will the program terminate?
- Is there any unreachable code?
- Are there any off-by-one iteration errors?
- Are () in arithmetic expression evaluation used as required to achieve desired results?

Initialisation
Check variable and parameter initialisation:
- At program initiation.
- At start of every loop.
- At function/procedure entry.

Calls
Check function call formats:
- Pointers
Parameters
Use of ‘&’

Strings
- Are all strings terminated with null-character, ‘\0’?
  Not all compilers generate a warning for array bounds overflow. The ip2k-elf-gcc compiler used at Metrima does not generate an error for:
- Are any string limits exceeded?

Pointers
- Make sure pointers are initialised.
- Are all pointers based on correct storage attributes?

Logic Operators
Verify the proper use of:
- ==
- =
- || and so on.
- Check every logic function for proper ()

Maintainability and Testability
- Is the code understandable (variable names, sufficient comments)?
- Is there a module header?
- Are coding conventions followed?

Error Handling
- Are all probable error conditions handled?
- Are error messages and return codes used?
- Are the default branches in CASE statements handled correctly?

The inspectors focus on reading source code with feedback to the author either through meetings or other means of communication. There are two types of defects that can be avoided by this technique:
- Indirect defects
  An example of indirect software defect can be a bad comment. The indirect danger is the risk to rewrite the code to make it correspond to the comment. Thus makes a fully functional code useless.
- Direct defects
  Direct defect would then be a fault made by a programmer that static analysis tool also should find.

Code reading techniques provide a systematic way of inspecting these kinds of defects. In fact, some of these code defects can only be found by the human inspection. This is the reason why code reading should not be avoided.

6.1.2 Software Inspection

Software inspection is one of the most used quality assurance practises in industry. It is a systematic way of finding defects in the system at an early stage. The technique was first presented in 1976 by Michael Fagan at IBM and has since then become the most influential work in software inspections. According to the author Jeff Tian almost all the other inspection
processes and techniques have their roots in Fagan’s work. Some reports identified several companies that have managed to achieve defect removal of >99%. [SR97a] This calculation is calculated as:

\[
\frac{\text{Number of defects found prior release}}{\text{Number of defects found prior release + number of defects reported by the customers during first } n \text{ months}}
\]

During an inspection the team checks the code and its documentation against a prepared checklist. The checklist may consist of points such as:

- Consistency use of definitions of data types and structures with the design and the standards in the company
- Review of algorithms and computations for their correctness

The main difference between the code reading techniques and inspection technique is that inspections require more preparations and it can not be done by one person. Fagan’s inspection technique involves five major steps (in some books a planning-step is also described) [Tian05c], see Figure 20.

1. **Overview** – The team arrange a meeting for an overview of the inspection goals. Both design and the code are presented.

2. **Preparation** – All members in the review-group have to individually examine the code and make note of all errors and questions that comes up during the review.

3. **Inspection** – The group meets and share their thoughts and notes with each other. Then the whole group gets a chance to give comments at each others notes. (It is not every time that the error really is an error, it could be a mistake done by the code inspector.) Once a real defect is found, no attempts should be made to find a solution.

4. **Rework** – All defects noted during the inspection-phase should be solved by the designer or programmer.

5. **Follow-up** – Depending on the number of defects the group should decide if new inspection should be done or if it is satisfying just to inspect the rework made during rework-phase.
Minimum resource requirements for an inspection

As mentioned above inspections require more preparations and it can not be done by one person. The fact is that inspections require at least three people. A description of the inspection team and their responsibilities is given below.

- **Moderator** – coordinates the inspection and leads the discussion
  Selection of the moderator is very important part of the inspection method. The moderator must be able to:
  - To select inspection team
  - Understand the information being inspected
  - Lead the team in a effective discussion
  - Handle disputes
  - Recognize key issues and keep focus on them
  - Assign responsibilities appropriately

- **Producer** – the person whose work is being inspected
  The producer is the person who prepared the information that will be inspected. The producer responsibilities are:
  - To ensure that the product is ready for the inspection
  - To make required information available on time
  - To support the moderator
  - To resolve all problems identified by the inspection team
  - To avoid becoming defensive
  - To clarify all issues that are not clear to the inspectors

- **Inspector** – the person that inspects the product.
  The inspectors are selected based on their knowledge of the product being inspected. Depending on the range of the inspection the inspectors can be selected to represent a cross-section of skills. I.e. if the inspection is about code inspection then a person with software knowledge is to prefer. The responsibilities for the inspectors are:
  - To be familiar with the product being inspected
  - To identify differences between the product, documentation and standards
Example instructions for static analysis technique

Static analysis of source code is a time consuming job. An approximation is to analyse 200 lines of source code per day and person. This is of course also depending on the structure and difficulty of the source code. The text below is an example on how static analysis can be applied.

1. Get an overview of the code by reading it through. Make sure to not go into details yet. Mark the code that you find questionable.

2. Determine the dependencies between the individual functions in the source code, for example with a call tree.

3. Try to gain an understanding of every function (sequences, loops, etc.) you described in case 2 by applying the ad-hoc technique as described earlier.

4. When you are done with the case 3 you can then use the specification to search for inconsistencies in your observations and the actual specification.

5. If the specification does not cover all individual functions then you have to derive it from the component’s specification. Make sure then to write down all of these derivations in case of future examinations.

6. When you feel that you are done with case 4 and 5 try then to isolate these inconsistencies within the source code that you examined in case 1, 2 and 3. For every fault-isolation write something describing about the fault, e.g. if the fault depends on bad initialization, bad interface, bad cosmetic in the source code, bad data types etc.

6.2 Which quality methods are used at Metrima today

During the years developers of the C10v2 have written their own checklist to verify the correctness of the changes made in the source code. This checklist consists of 19 error prone cases important to check for in C10v2 source code.

The check-list is presented below:

1. Invalid array indexing.

2. Copy-paste error.

3. Loop-conditions.

4. Data types int/char/unsigned int
5. Addresses in EEPROM

6. Variable values

7. /**

8. Check that all cases within a switch case have a break statement.

9. Division with 0

10. Array declarations where define statements are used with division.
    E.g., char a[NO_OF_ELEMENTS / 8]
    /* here the define statement must be a multiple of 8 */

11. Make sure that array’s declared with extern are made as a pointers and not as a
    containers with a fixed length.
    E.g.
    ERROR: extern char a[10];
    /* the compiler does not generate a warning for out-of-bound writing*/
    OK: extern char *a;

12. Make sure that 255 are NOT used in cases where it should be 256.

13. Make sure that the stack area does not grow into the heap area. Look in the memory
    map for the ip2k for the right addresses.

14. Data allocated within a function should NOT be sent via an interrupt driven routine.
    This is due to writing on the stack. When the function has finished its execution the
    data is removed from the stack and the interrupt driven routine would then send wrong
    data.

    E.g.: void foo()
    {
      char v[] = {1, 2, 3, 4, 5, 6, 7};
      Neuron_writeData(v);
      // interrupt-driver routine executed after foo()
    }

15. When using memcopy, EEPROM_read/writeString, function check that sign ‘&’ is not
    missed.

16. Check in MEM_map.h file that xxx_END is correctly calculated. E.g.,
    (XXX_BASE + XXX_SIZE) and not (XXX_BASE * XXX_SIZE)

17. An address in the memory can be damaged within a 10 years period if data is written
    on the same address at least one time per hour. It is necessary to check the memory
    addresses used for writing to ensure that this will not happen.
18. Check that there is ( ) when using MEM_map to make sure that arithmetic operations are done correctly.

19. Check that memset function is used with sizeof( ) and not hard typed. E.g., memset(id, 0, 10) when it should be memset(id, 0, sizeof(id)).

By applying check-list method on C10v2 code a number of errors were found. The errors are presented in Appendix A.

6.3 Which methods are used at other companies

During the market analysis one person from ABB and one person from Ericsson were interviewed. Both companies have their own strategy of code inspections but there are similarities between them as well. In this chapter the focus is not on a comparison between Ericsson and ABB. Instead the focus is on those details that these two companies do that could be of importance in C10v2’s code inspection part. The whole interview can be found in the chapter 4 Market Analysis.

6.3.1 ABB

A method used as a quality assurance practice at ABB is pair-programming. This kind of programming is a mix of programming and code reading at the same time since one of the programmers are examining the code during the implementation. The table below presents some of the methods used at ABB for code inspections. It is important to say that we only mean the ABB departments we were in contact with. This is not a general summary for all ABB departments.

<table>
<thead>
<tr>
<th>Methods</th>
<th>ABB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirement analysis</td>
<td>Yes</td>
</tr>
<tr>
<td>Design analysis</td>
<td>Yes</td>
</tr>
<tr>
<td>Review methods:</td>
<td></td>
</tr>
<tr>
<td>• Walkthrough</td>
<td>Yes</td>
</tr>
<tr>
<td>• Inspections</td>
<td>Yes</td>
</tr>
<tr>
<td>• Checklists</td>
<td>Yes</td>
</tr>
<tr>
<td>Complexity measuring methods:</td>
<td></td>
</tr>
<tr>
<td>• Estimation (Lines-of-code)</td>
<td>Yes</td>
</tr>
<tr>
<td>• McCabe Cyclomatic Complexity(^1)</td>
<td>No</td>
</tr>
<tr>
<td>• Hallstead’s Volume(^2)</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 16. Inspection methods used at ABB.

\(^1\) Cyclomatic complexity is a static software metric method. Cyclomatic complexity may be considered as confidence-factor for a program. It was introduced by Thomas McCabe in 1976. Cyclomatic complexity measures the number of linearly-independent paths (i.e. sections of code with no branches) through a program module needed to determine the number of tests.
required to obtain complete coverage. This measure provides a single number that can be compared to the complexity of other programs.

2 Halstead’s volume, V, describes the size of the implementation of an algorithm. Computation of (V) is based on the number of operations performed and operands handled in the algorithm.

Following list is one of the ABB’s code check-lists:

1 Algorithmic analysis. Verify that the function conforms to the design specification.

2 Variables
   a) Check that all variables are initialised;
   b) Check that global and local variables do not interfere.

3 Boundary value inspection
   a) Overflow checks
      i) Check each arithmetic operation for overflow.
      ii) Check pointer and memory operations so that they do not read and write in non assigned memory areas. Beware of operations on multi dimensional arrays and queues.
      iii) Check that all pointer operators are used correctly.
   b) Beware of type casting, both explicit and implicit. Avoid implicit type casting.
   c) Select statements - check that all if- and switch statements have all branches defined including default statements.
   d) Loops - check that all loops have an upper bound on their iteration.
   e) Conditions
      i) Beware of complex conditional statements. Check explicitly for correct use of syntax and logic.
      ii) Beware of assignments in conditional statements.
      iii) Check for wrongful use of syntax regarding logical operators and bit operators, like 
          && - &, || - |, ! - != ..... 

4 Big vs. little endian. Check for correct use of addresses and bit operations. This is especially important when code is designed to execute on several platforms.

5 Concurrency (parallel operations)
a) Atomic transactions - check that each section that is assumed atomic in a function, is in fact atomic. In addition, check that the termination of an atomic section is made properly.

b) Beware of interrupt register manipulations. Make sure that one interrupt does not affect other interrupts unintentionally.

6 Declaration and definitions and parameter passing

a) Check that header file declarations conform to the definitions of function headers.

b) Check that the parameters passed to and from a function conform to the definition of the function.

7 Dead and almost dead code

a) Try to identify dead code.

b) Try to identify code sections which are unlikely to be executed during run-time.

8 Check that the error handling is done according to the specification. Record all the condition that trigger error handling for future use in the dynamic verification.

9 Find the parameter setting that maximises the execution time of the function. This information shall later be used in the dynamic verification phase, in order to verify that the timing requirements are met.

10 Find the parameter setting that maximises the stack use. This information shall later be used in the dynamic verification phase, in order to verify that stack overflow does not occur.

11 Check that all error messages are as clear, helpful and exact as possible.

6.3.2 Ericsson

According to Eldh, see chapter 4.2 Ericsson, Ericsson may be one of the worlds leading companies in software inspections. The table below presents some of the methods used at Ericsson that in some way reflects the code inspections part.

<table>
<thead>
<tr>
<th>Methods</th>
<th>Ericsson</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirement analysis</td>
<td>Yes</td>
</tr>
<tr>
<td>Design analysis</td>
<td>Yes</td>
</tr>
<tr>
<td>Review methods:</td>
<td></td>
</tr>
<tr>
<td>• Walkthrough</td>
<td>Yes</td>
</tr>
<tr>
<td>• Inspections</td>
<td>Yes</td>
</tr>
<tr>
<td>• Checklists</td>
<td>Yes</td>
</tr>
<tr>
<td>Complexity measuring methods:</td>
<td></td>
</tr>
<tr>
<td>• Estimation (Lines-of-code)</td>
<td>Yes</td>
</tr>
<tr>
<td>• McCabe Cyclomatic Complexity</td>
<td>Yes</td>
</tr>
<tr>
<td>• Halstead´s Volume</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Table 17. Inspection methods used at Ericsson.

Another method used at Ericsson is working in pair or XP (Extreme Programming). As we described in the chapter 4.1 ABB, this is an alternative way of code reading that is done during the code development. Unfortunately we did not managed to get any specific details about code inspections used at the Ericsson. This is the reason why a code check-list from Ericsson is not presented.

6.4 New model suited for the C10v2

This chapter is summary of the methods mentioned above. To not repeat ourselves we will here instead refer to the earlier chapters above. The result of our research and market analysis would later on show that many of the techniques we wrote about are even used in the industry. This is one of the reasons why we recommend these for use in the C10v2 project. Another reason is because it is applicable on C10v2 code. These methods require minimal integration requirements with Ubicom’s environment.

Our recommendation would be to perform the following methods on the C10v2 code to assure that the code is verified during the whole application development:

1. Ad-hoc inspections (at least once after every new change in the code). For a description see section 6.1.1 A study about code inspections.
2. Check-list inspections (must be done for every release). For a description see section 6.1.1 A study about code inspections and use Table16.
3. Code inspections (do once for every new release). For a description see section 6.1.1 A study about code inspections.

Table 16 below shows a number of testcases and three different ways to check these. The letters C, M and L mean:

- **C** (Compiler)
  Many of testcases are automatically checked during the compilation. In column C the compiler options necessary for each testcase are shown. A compiler does not find semantically errors and therefore manual testing should also be done. As an example the ip2k-elf-gcc compiler will not generate any warning for the following code:

```c
signed int min = -5;
unsigned int max = 5;
max = min;  //Now max have a different value then expected
```

- **L** (PC-Lint)
  In column L the PC-Lint options necessary for each testcase are shown. Using PC-Lint is a good complement to both compiler and manual testing. For the example above PC-Lint generates following warning:

```c
--- Module: test.c
    max = min;

  test.c 5 Info 732: Loss of sign (assignment) (int to unsigned int)
```
M (Manual)

Some of the testcases can only be checked manually. In the Table 18 all testcases that can be checked manually are marked with letter “X”. Testcases that can not be checked by the compiler or PC-Lint must be checked manually.

<table>
<thead>
<tr>
<th>Nr</th>
<th>Test</th>
<th>C</th>
<th>M</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>Initialisation</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Check for correct use of data types: int/char/unsigned int.</td>
<td>-</td>
<td>X</td>
<td>+e732</td>
</tr>
<tr>
<td>2</td>
<td>Check variable initialisation at program initiation.</td>
<td>Wall</td>
<td>X</td>
<td>+e134, +e530, +e727</td>
</tr>
<tr>
<td>3</td>
<td>Check variable and parameter initialisation at start of every loop.</td>
<td>Wall</td>
<td>X</td>
<td>+e134, +e727</td>
</tr>
<tr>
<td>4</td>
<td>Check variable and parameter initialisation at function/procedure entry.</td>
<td>Wall</td>
<td>X</td>
<td>+e134, +e727</td>
</tr>
<tr>
<td>5</td>
<td>Verify that implicit type casting is not used (type casting made by the compiler). Wall</td>
<td>X</td>
<td>+e919</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Check explicit type casting e.g. ( x = (\text{int}) \ var )</td>
<td>Wall</td>
<td>X</td>
<td>+e571, +e792</td>
</tr>
<tr>
<td></td>
<td><strong>Calls</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Check function calls where pointers are used. (Any NULL pointers passed to a function?) Wall</td>
<td>X</td>
<td>+e668</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Check that correct parameters are sent to functions.</td>
<td>-</td>
<td>X</td>
<td>+e418</td>
</tr>
<tr>
<td></td>
<td>The ip2k-elf compiler does not generate a warning when a double variable is assigned to an int variable (possible loss of data). This can be found by checking the parameters sent to the functions.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Check function calls where &amp; is used</td>
<td>-</td>
<td>X</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>Check that memset function is used with sizeof( ) and not hard typed. E.g, memset(id, 0, 10) when it should be memset(id, 0, sizeof(id)).</td>
<td>-</td>
<td>X</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>Check pointer and memory operations so that they do not read and write in non assigned memory areas. (“-fdump-three” flag generates a file with information of where different lines of code are in memory)</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Pointers</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>Check that pointers are initialised.</td>
<td>Wall</td>
<td>X</td>
<td>+e413, +e613</td>
</tr>
<tr>
<td>13</td>
<td>Check that use of pointers begins with if(p != NULL) (run-time error if accessing a NULL )</td>
<td>-</td>
<td>X</td>
<td>-</td>
</tr>
<tr>
<td>14</td>
<td>Make sure that array’s declared with extern are made as pointers and not as containers with a fixed length. E.g, extern char *a; OK: extern char *a; ERROR: extern char a[10];</td>
<td>-</td>
<td>X</td>
<td>-</td>
</tr>
<tr>
<td>15</td>
<td>Verify the proper use of =, ==, !, !</td>
<td>-</td>
<td>X</td>
<td>+e514, +e720, +e731, +e777, +e20, +e720, +e723</td>
</tr>
<tr>
<td></td>
<td>Following code could have a semantically error (min == max) that is not detected by the compiler. int min = 0, mean = 5, max = 10; //no warning generated by ip2k-elf-gcc compiler if ((min &gt; mean) &amp;&amp; (min = max)) For the same code PC-Lint generates following: --- Module: test.c</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
|16| Verify the proper use of `|` and `||`.
|   |   |   |
|   |   |   |
|17| Verify the proper use of `>>`, `<<`.
|   |   |   |
|   |   |   |
|18| Verify the proper use of `&` and `&&`
|   |   |   |
|   |   |   |
|19| When using memcopy, EEPROM_read/writeString, function check that sign `&` is not missed.
|   |   |   |
|   |   |   |
|20| Check for “:” operator
|   |   |   |
|   |   |   |
|21| Invalid array indexing, upper and lower bound
|   |   |   |
|   |   |   |

### Indexing

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Department of Computer Science and Electronics, IDE**

Mälardalen University, Västerås
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>Check loop-conditions for upper and lower bound.</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Ip2k-elf-gcc compiler does not generate a warning about possible out-of-band error in following code.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>int i=0, arr[2];</td>
<td></td>
</tr>
<tr>
<td></td>
<td>for(i = 0; i &lt; 5; i++)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>arr[i] = 5;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>For the same code PC-Lint generates following:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>--- Module: test.c</td>
<td></td>
</tr>
<tr>
<td></td>
<td>arr[i] = 5;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>test.c 7 Info 725: Expected positive indentation from line 6</td>
<td>+e725</td>
</tr>
<tr>
<td></td>
<td>test.c 7 Warning 662: Possible creation of out-of-bounds pointer (3 beyond end of data) by operator '['</td>
<td>+e662</td>
</tr>
<tr>
<td>23</td>
<td>Addresses in EEPROM</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>Array declarations where define statements are used with division. E.g, char a[NO_OF_ELEMENTS / 8] /* here the define statement must be a multiple of 8 */</td>
<td>-</td>
</tr>
<tr>
<td>25</td>
<td>Arithmetic operations</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Check arithmetic operations for overflow</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>int i = 65537; //warning: overflow in implicit constant conversion</td>
<td>+e648</td>
</tr>
<tr>
<td></td>
<td>double a = 9999999999, b = 9999999999; int c = (a + b); //No warning</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>Division with 0</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Ip2k-elf-gcc compiler does not generate a warning for following code:</td>
<td>+e648</td>
</tr>
<tr>
<td></td>
<td>int i = 0;</td>
<td>+e108</td>
</tr>
<tr>
<td></td>
<td>for( i = 0; i &lt; 5; i--)</td>
<td>+w616</td>
</tr>
<tr>
<td></td>
<td>{</td>
<td>+e539</td>
</tr>
<tr>
<td></td>
<td>break;</td>
<td>+e744</td>
</tr>
<tr>
<td></td>
<td>i = i;</td>
<td>+e527</td>
</tr>
<tr>
<td></td>
<td>}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>For the same code PC-Lint generates following:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>--- Module: test.c</td>
<td></td>
</tr>
<tr>
<td></td>
<td>i = i;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>test.c 9 Warning 527: Unreachable</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>Branches</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>Select statements – check that all if- and switch statements have all branches defined including default statements.</td>
<td>-</td>
</tr>
<tr>
<td>29</td>
<td>Unreachable code.</td>
<td>-</td>
</tr>
<tr>
<td>30</td>
<td>Copy-paste error (lines that are almost identical)</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>31</td>
<td>Make sure that 255 is NOT used in cases where it should be 256 and vice versa.</td>
<td>X</td>
</tr>
<tr>
<td>32</td>
<td>Make sure that the stack area does not grow into the heap area. Look in the memory map for the ip2k for the right addresses. (-fdump-three flag)</td>
<td>X</td>
</tr>
<tr>
<td>33</td>
<td>Data allocated within a function should NOT be sent via an interrupt driven routine. This is due to writing on the stack. When the function has finished its execution the data is removed from the stack and the interrupt driven routine would then send wrong data. E.g:</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>void foo()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>{</td>
<td></td>
</tr>
<tr>
<td></td>
<td>char v[] = {1, 2, 3, 4, 5, 6, 7};</td>
<td></td>
</tr>
<tr>
<td></td>
<td>/* interrupt-driver routine executed after foo() */</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Neuron_writeData(v);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>}</td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>Check in MEM_map.h file that xxx_END is correctly calculated. E.g,</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>(XXX_BASE + XXX_SIZE) and not</td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>An address in the memory can be damaged within a 10 years period if data is written on the same address at least one time per hour. It is necessary to check the memory addresses used for writing to ensure that this will not happen.</td>
<td>X</td>
</tr>
<tr>
<td>36</td>
<td>Check that there is () when using MEM_map.</td>
<td>X</td>
</tr>
<tr>
<td>37</td>
<td>Check for correct use of macros, must have ()</td>
<td>X</td>
</tr>
<tr>
<td>38</td>
<td>Check complex conditional statements. Are () used correctly?</td>
<td>X</td>
</tr>
<tr>
<td>39</td>
<td>Check that global and local variables do not interfere</td>
<td>X</td>
</tr>
<tr>
<td>41</td>
<td>Check that all #ifdef have a #endif.</td>
<td>Wall</td>
</tr>
<tr>
<td>41</td>
<td>Signed and unsigned comparison. E.g. (the default options will not generate an error)</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>unsigned int x; signed int y; if(x&gt;y)</td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>Address assignment of a local variable to a variable outside the function scope.</td>
<td>Wall</td>
</tr>
</tbody>
</table>

Table 18. Checklist for C10v2.
7 Static Analysis Tools

This chapter is about static analysis tools evaluated during the research part of this thesis. Static analysis or static testing covers all kinds of tests made on software applications without executing the source code. Static analysis tools are able to find errors like assignment between the unsigned and signed variables. No harm arises until the signed variable is actually negative. Ip2k-elf-gcc compiler does not complain about this assignment. These kinds of faults could be caught by a static analysis tool. Figure 21, is an example of how to use static analysis tools.

Before compiling all testing is intended to ensure the quality of the source code. After compilation all testing is intended to ensure the functional quality of the application. The compiler itself is a static analysis tool but this is a very target specific tool. Static analysis tools can. This is why the Figure 21 is divided in three parts, Static analysis tools, Compiler and Dynamic Analysis tools.

The idea of static analysis tools is to automate the code inspection process and to enforce coding standard into the source code. This statement depends very much on what static analysis tool is used since expensive tools usually offers more functionality then cheaper tools. The tools evaluated in this thesis are able to automate both code inspection and code standard methods based on ISO standards. According to the coding standard of Metrima only the tools able to detect programming errors are of interest for Metrima since the programmers have their own standard. Similar to the Code standards and Code inspections chapters the focus of this chapter will be on clarifying Metrima’s current situation and the use of this method in other companies by discussing following topics:

- A study about static analysis tools
- Which quality methods are used at Metrima today
- Which methods are used at other companies
- New model suited for the C10v2
7.1 A study about Static analysis tools

Static analysis tools can be applicable at several stages during an application development. Figure 22 shows these stages [SAT0]. The idea of this concept is primarily the possibility to have a distributed application development. This means that the code may be developed at several departments and must be integrated with the rest of the application before system testing can be performed. However, this only has influence on the Integration stage which is not needed at Metrima since all code is developed in-house.

![Figure 22. When to use static analyzers.](image)

- **Development and Unit testing**
  This stage encourages the use of static analyzers during code development. The focus is to find errors as soon as they occur. By using these tools during development and unit testing, the programmers can also introduce coding standards in an easier way.

- **Integration and System testing**
  Integration and system testing level comes after development and unit testing. This test is made by the programmers and serves to verify that the whole system is checked for errors before it is sent to the QA team.

- **Acceptance testing and QA**
  Acceptance and QA testing teams can use static analyzers to generate different analysis metrics needed to compare the present release with the earlier releases or just for future analysis. The reports made at this level can also be used to give feedback to the programmers and to see in which grade code standards are used.

Below is a description of the tools evaluated in this thesis. These tools were chosen according to the following criteria:

1. Does it cost to evaluate this product or is there an evaluation copy?
2. What is the price of the product?
3. Is this tool used at other companies and what are the user experiences?
4. Is this tool user-friendly (i.e., easy to install and use)?

An evaluation of the answers on these questions resulted in following evaluated tools:
7.1.1 PC-Lint

The static analysis tool from Gimpel Software, PC-Lint, is a static analysis tool that checks C source code. It is one of the longest continuously advertised software tools in human history [PCL1]. Today, PC-Lint support both C and C++. This is the main difference between PC-Lint and Splint, see chapter 7.3.2 Splint. It uses K&R and ANSI as the assumed standard for C. The first release of the PC-Lint goes back to early 1985. Since then Gimpel Software has developed several tools similar to PC-Lint. Some checks performed by PC-Lint are:

- **Redundant code**
  If a particular function, for example, is not being called, then the function is truly redundant.

- **Value tracking**
  PC-Lint does this by remembering variable values. If later, these variables are used in an inappropriate context so that, for example, they can cause an out-of-bounds exception, a warning is generated.

- **Array indexing error**

- **Checks on Weak Definials**
  Weak definials (defines) are macro definitions, typedef’s, declarations, structs. Redundancy (more than one identical item) is one possible error that may occur using these definitions.

- **Possibly uninitialized variable**

- **Strong type checking**
  Strong types are user defined types. Example:

  ```c
  typedef int Count;
  typedef int Bool;
  Count c;
  Bool b;
  ...
  b = c; //this can lead to unwanted behaviour.
  ```

  Not all compilers perform this strong type checking.

**Code analysis with PC-Lint**

The PC-Lint manual contains about 2000 error/warning messages divided in four different warning levels. However, not all of these errors/warnings are suitable for the C. Table 19 shows what errors/warnings are to be used to check C code.

<table>
<thead>
<tr>
<th>Error/Warning suitable to check C code</th>
<th>Warning level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax errors</td>
<td>1-199 (see PC-Lint manual)</td>
</tr>
<tr>
<td>Fatal errors</td>
<td>300-399 (see PC-Lint manual)</td>
</tr>
<tr>
<td>Warnings</td>
<td>400-699 (see PC-Lint manual)</td>
</tr>
<tr>
<td>Informational</td>
<td>700-799 (see PC-Lint manual)</td>
</tr>
<tr>
<td>Elective Notes</td>
<td>900-999 (see PC-Lint manual)</td>
</tr>
</tbody>
</table>

Table 19. PC-Lint errors/warnings used for C code.
Example on Error/Warning flags

Following flags are examples of the flags mentioned in the above table. For further information refer to the PC-Lint manual.

- Syntax errors
  13 Bad Type – A type adjective such as long, unsigned etc. cannot be applied to the type which follows.

- Fatal errors
  322 Unable to open include file – *Filename* is the name of the include file which could not be opened.

- Warnings
  414 Possible division by 0 – The second argument to either the division operator (/) or the modulus operator (%) may be zero.

- Informational
  712 Loss of precision – (Context) *Type to type* – An assignment is being made from an unsigned quantity to a signed quantity, that will result in the possible loss of one bit of integral precision such as converting from unsigned int to int.

- Elective Notes
  919 Implicit conversion (Context) Type to type – A lower precision quantity was assigned to a higher precision variable as when an int is assigned to a double.

Example of the PC-Lint output

Executing PC-Lint on the C10v2 code generated many warnings. The warnings presented in Table 20 are only a part of the whole report.

<table>
<thead>
<tr>
<th>File</th>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msg_handler.c</td>
<td>343</td>
<td>Info 734: Loss of precision (assignment) (16 bits to 8 bits)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>data[3] = (packet_cnt-1) / 256; //Resend acknowledge</td>
</tr>
<tr>
<td></td>
<td>444</td>
<td>Warning 545: Suspicious use of &amp;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>EEPROM_writeString(S0_RESET_BILL_VALUE_OFFSET, (u8_t *)&amp;S0_total_count, sizeof(S0_total_count)); // Nollställ förra periodens summerade logdata genom att skriva ner aktuell mätarställning</td>
</tr>
<tr>
<td></td>
<td>374</td>
<td>Warning 616: control flows into case/default</td>
</tr>
<tr>
<td></td>
<td></td>
<td>case ICP_REPROGRAM:</td>
</tr>
<tr>
<td></td>
<td>618</td>
<td>Info 718: in8 undeclared, assumed to return int</td>
</tr>
<tr>
<td></td>
<td></td>
<td>data[7] = in8(RA);</td>
</tr>
</tbody>
</table>

Table 20. Warnings from C10v2 code generated by PC-Lint.

A configuration file of the PC-Lint is shown in Appendix B.
7.1.2 RSM

RSM, Resource Standard Metrics, is a static analysis tool focusing on finding errors within the source code but it also provides methods to satisfy ISO9001, TickIT, CMM Levels 3 and 4 compliance. There are two main reasons why this tool became one of those that were evaluated in this thesis. First reason is the easy procedure to get an evaluation copy and the second reason is that this tool was able to perform both code checking for errors and introduce coding standards into the code. It is important to use RSM on compiled code because it otherwise may miss parts of code.

Following text is some of the key features of the RSM:

- Use one tool across all project languages for K&R and ANSI C, ANSI C++ and Java 2.0+ - RSM Options and Reports
- No limitations to the file length, number of files or usage, RSM has processed projects consisting of over 10,000 files and 7,000,000 Lines of Code
- Use RSM to satisfy ISO9001, TickIT, SEI Levels 3 and 4 compliance
- Determine exactly how RSM sees your code - Deterministic Report
- Collect 100's of Source Code Metrics by the function, class, file, and project - RSM Metrics
- Analyze source code for over 50+ latent programming errors which compilers miss - RSM Quality Analysis
- Analyze source code for code style enforcement - RSM Quality Analysis
- Create reports in HTML with hyperlinks to the code - HTML Output Mode

Code analysis with RSM

RSM provides source code analysis (quality notices) for over 50 programming errors that not all compilers catch. Below are some of these checks that are used to check C programs. The same information can be read at the [RSM2]. The numbers of the quality notices below are intentionally not in sequence. This makes it easier to trace it to the origin notice at the [RSM2].

**Quality Notice No. 1**

Emit a quality notice when the physical line length is greater than the specified number of characters.

Rationale: Reproducing source code on devices that are limited to 80 columns of text can cause the truncation of the line or wrap the line. Wrapped source lines are difficult to read, thus creating weaker peer reviews of the source code.

**Quality Notice No. 4**

Emit a quality notice if there exists an assignment operator '=' within a logical 'if' condition.

Rationale: An assignment within an "if" condition is likely a typographical error giving rise to a logic defect. However, some programmers place compound statements into the "if" condition making the code difficult to read.

**Quality Notice No. 5**

Emit a quality notice if there exists an assignment operator '=' within a logical 'while' condition.
Rationale: An assignment within a "while" condition is likely a typographical error giving rise to a logic defect. However, some programmers place compound statements into the "while" condition making the code difficult to read.

**Quality Notice No. 13**
 Emit a quality notice when a 'switch' statement does not have a 'default' condition.
Rationale: A "switch" statement must always have a default condition or this logic construct is non-deterministic. Generally the default condition should warn the user of an anomalous condition which was not anticipated by the programmer by the case clauses of the switch.

**Quality Notice No. 14**
 Emit a quality notice when there are more 'case' conditions than 'break', 'return' or 'fall through' comments.
Rationale: Many tools, including RSM, watch the use of "case" and "break" to insure that there is not an inadvertent fall through to the next case statement. RSM requires the programmer to explicitly indicate in the source code via a "fall through" comment that the case was designed to fall through to the next statement.

```java
    case 'a': // fall through
    case 'b':
    { ... }
```

**Quality Notice No. 18**
 Emit a quality notice when the eLOC within a function exceeds the specified maximum.
Rationale: An extremely large function is very difficult to maintain and understand. When a function exceeds 200 eLOC (effective lines of code), it typically indicates that the function could be broken down into several functions.

**Quality Notice No. 22**
 Emit a quality notice when each if, else, for or while is not bound by scope.
Rationale: Logical blocks should be bound with scope. This clearly marks the boundaries of scope for the logical blocks. Many times, code may be added to non-scoped logic blocks thus pushing other lines of code from the active region of the logical construct giving rise to a logic defect.

**Quality Notice No. 23**
 Emit a quality notice when the '?' or the implied if-then-else construct has been identified.
Rationale: The ? operator creates the code equivalent of an "if" then "else" construct. However the resultant source is far less readable.

**Quality Notice No. 28**
 Emit a quality notice when the cyclomatic complexity of a function exceeds the specified maximum.
Rationale: Cyclomatic complexity is an indicator for the number of logical branches within a function. A high degree of V(g), greater than 10 or 20, indicates that the function could be broken down into a more modular design of smaller functions.

**Example of the RSM output**
Executing RSM on the C10v2 code generated many warnings. The warnings presented in tables 21-26 are only a part of the whole report.

Table 21. Warnings generated by RSM in file moist_alarm.c.

| Notice #1: Line 38, Physical line length exceeds 80 characters |
| Notice #22: Line 37, ‘if’ is not bound with scope braces {} |

Table 22. Warnings generated by RSM in file msg_handler.c.

| Notice #22: Line 143, ‘else’ is not bound with scope braces {} |
| Notice #30: Line 164, TAB character has been identified |
| Notice #14: Line 474, ‘case’ conditions do not equal ‘break’ |
| Notice #18: Line 474, Function eLOC exceeds maximum 200 eLOC |
| Notice #28: Line 474, Cyclomatic complexity exceeds 10 |

Table 23. Warnings generated by RSM in file onewire.c.

| Notice #22: Line 311, ‘if’ is not bound with scope braces {} |
| Notice #1: Line 323, Physical line length exceeds 80 characters |
| Notice #22: Line 330, ‘for’ is not bound with scope braces {} |
| Notice #23: Line 367, ‘?’ ternary operator identified |
| Notice #28: Line 368, Cyclomatic complexity exceeds 10 |
| Notice #27: Line 463, Number of function return points exceed 1 |
| Notice #6: Line 510, Pre-decrement operator ‘--’ identified |

Table 24. Warnings generated by RSM in file powerManagement.c.

Table 25. Warnings generated by RSM in file smoke_alarm.c.

Table 26. Warnings generated by RSM in file taskManager.c.

7.2 Which quality methods are used at Metrima today

C10v2 programmers have during the application development used one static analysis tool, Splint. Splint (Secure Programming Lint), is a programming tool for statically checking C programs for coding mistakes. Splint is developed by the Secure Programming Group at the
University of Virginia, Department of Computer Science. Splint was designed to detect inconsistencies within C implementations. Splint uses four different types of flags for controlling the output:

- **Global flags**
  Controlling initializations and global behaviour
- **Message format flags**
  Message format flags for controlling how messages are displayed
- **Mode selectors flags**
  Describes checking level of Splint. There are four levels:
  1. weak
  2. standard
  3. checks
  4. strict
- **Checking flags**
  Controls checking and what classes of messages are reported

Controlling the output of these flags is described in the Appendix B in Splint manual. By using Splint, programmers can perform stronger checking in an easy and relatively fast way. Below are some of the faults that can be found by using Splint:

- Null dereferences
- Undefined Values
- Memory management
- Sharing
- Function interfaces
- Control Flow
- Buffer sizes

For further information about Splint refer to [Splint1a].

A disadvantage of the Splint is its ability to generate many errors and warnings. A test run of C10v2 code with Splint generated more than 100 warnings. The number of outputs created by Splint can be tough to review. Therefore it is good practice to run Splint at the unit level. See, Figure 22, in section 7.1 *A study about Static analysis tools*. Even though Splint can find errors within the source code, the use of it at Metrima has been sporadic.

Tables 27-28 below show some examples where Splint will complain about the code. The examples are taken from Splint manual.

<table>
<thead>
<tr>
<th>Null.c</th>
<th>Running splint</th>
</tr>
</thead>
<tbody>
<tr>
<td>char firstChar1(char *s)</td>
<td>&gt;splot null.c</td>
</tr>
<tr>
<td>{</td>
<td>Splint 3.0.1</td>
</tr>
<tr>
<td>return *s;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>char firstChar2(char *s)</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td></td>
</tr>
<tr>
<td>if(s == NULL) return ‘\0’;</td>
<td></td>
</tr>
<tr>
<td>return *s;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>

Table 27. Checking the NULL pointer with Splint.
In function `firstChar2(char *s)` no warning is generated because the pointer is checked for NULL.

<table>
<thead>
<tr>
<th>usedef.c</th>
<th>Running splint</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>extern void setVal(/*@out@*/ int *x);</code></td>
<td><code>&gt; splint usedef.c</code></td>
</tr>
<tr>
<td><code>extern int getVal(/*@in@*/ int *x);</code></td>
<td><code>usedef.c:11: Value *x used before definition</code></td>
</tr>
<tr>
<td><code>extern int mysteryVal(int *x);</code></td>
<td><code>usedef.c:13: Passed storage x not completely defined</code></td>
</tr>
<tr>
<td><code>int dumbfunc (/*@out@*/ int *x, int i)</code></td>
<td><code>(*x is undefined): getVal (x)</code></td>
</tr>
<tr>
<td><code>{</code></td>
<td><code>usedef.c:15: Passed storage x not completely defined</code></td>
</tr>
<tr>
<td><code>if (i &gt; 3)</code></td>
<td><code>(*x is undefined): mysteryVal (x)</code></td>
</tr>
<tr>
<td><code>11   return *x; //by default /*@in@*/</code></td>
<td><code>Finished checking --- 3 code warnings</code></td>
</tr>
<tr>
<td><code>else if (i &gt; 1)</code></td>
<td><code>No error is reported for line 18, since the incompletely defined storage x is passed as an out parameter. After the call, x may be dereferenced, since setVal is assumed to completely define its out parameter. The warning for line 15 would not appear if +impouts were used since there is no in annotation.</code></td>
</tr>
<tr>
<td><code>13   return getVal (x);</code></td>
<td></td>
</tr>
<tr>
<td><code>else if (i == 0)</code></td>
<td></td>
</tr>
<tr>
<td><code>15   return mysteryVal (x);</code></td>
<td></td>
</tr>
<tr>
<td><code>else</code></td>
<td></td>
</tr>
<tr>
<td><code>{</code></td>
<td></td>
</tr>
<tr>
<td><code>18    setVal (x);</code></td>
<td></td>
</tr>
<tr>
<td><code>19    return *x;</code></td>
<td></td>
</tr>
<tr>
<td><code>}</code></td>
<td></td>
</tr>
<tr>
<td><code>}</code></td>
<td></td>
</tr>
</tbody>
</table>

Table 28. Checking in- and out parameters by the Splint.

The `/*@in@*/` annotation can be used to denote a parameter that must be completely defined, even if `impouts` option is on. Parameters without these annotations are by default set to `/*@in@*/`. Splint generates a warning every time a parameter is used within a function before it is assigned. The `/*@out@*/` annotation can be used to denote a parameter that may be undefined. Splint defines if a parameter should be defined or not before and after every function call. In this example Splint never generates a warning that “x” might be dereferenced in function `setVal(x)` because it is set to `/*@out@*/`. This means that the value of “x” can be NULL. Returning this pointer can lead to unwanted behaviour later in the application. This is the reason why developers always should check pointers before use.

### 7.3 Which methods are used at other companies

The research about static analysis tool has shown that the use of static analysis tools is very high at the ABB and Ericsson. Unfortunately we did not manage to find out what options were used when the code is checked by these tools. The experience we got during the evaluation of the static analysis tools has shown that there are only few changes needed to make the outputs of static analysis tools satisfying. LINT, SPLint, RSM and Coverity are some of the static analysis tools used at these companies. A short description of these tools is presented below.

#### 7.3.1 LINT

LINT is one of the static analysis tools in Lint family. Other tools are PC-Lint, FlexeLint and LintPlus. Some of the analysis provided by the LINT is source code analysis, value tracking,
and array indexing errors. Like SPLint, LINT suffers from high number of output results which can be customized with flags and code annotations. LINT is used at a unit level both at the ABB and Ericsson.

More details and examples of one of the LINT family tools, PC-Lint, are described in chapter 7.1.1 PC-Lint.

7.3.2 SPLint

More details and examples of SPLint code checking are described in chapter 7.3.2 SPLint.

7.3.3 Coverity

Coverity static analysis tool is one of the tools used at the Ericsson to uncover the issues within the source code. At the manufacture website, [Coverity1], the description of Coverity static analysis tool is as following:

"Coverity automates the detection of software defects and security vulnerabilities for complex software as developers write code. With Coverity, development teams identify critical software problems that could have a catastrophic impact. Our products help companies improve software quality and security, decrease time to market and optimize developer productivity. Coverity’s next-generation static analysis technology finds complex defects, scales to large code bases and does not waste developer time with false positives."

An evaluation of this tool was not made in this thesis because the price was not suitable for the Metrima.

7.3.4 RSM

This static analysis tool is used both at the ABB and Ericsson. The tool is little different than above mentioned tools. That is, the RSM is not only focusing on finding errors within the source code, it also provides methods to satisfy ISO9001, TickIT, CMM Levels 3 and 4 compliance. At the manufacture website, [RSM1], the description of RSM is as following:

"Resource Standard Metrics, or RSM, is a source code metrics and quality analysis tool unlike any other on the market. RSM provides a standard method for analyzing C, ANSI C++, C# and Java source code across operating systems. The unique ability of RSM to support virtually any operating system provides your enterprise with the ability to standardize the measurement of source code quality and metrics throughout your organization. RSM provides the fastest, most flexible and easy-to-use tool to assist in the measurement of code quality and metrics."

More details and examples of RSM are described in chapter 7.3.4 RSM.

7.4 New model suited for the C10v2

A model that is easy to maintain and do not involve extra costs for Metrima is to re-establish static testing of the C10v2 code with SPLint and complement it with PC-Lint. The RSM tool generated several warnings in C10v2 code and was very easy to use but most of these
warnings can be found by the tools Metrima already have, SPlint and PC-Lint, so further
investment is not necessary.

To run SPlint and PC-Lint some configurations are needed. Configuration file for the PC-Lint
is shown in Appendix B.
8 Dynamic analysis tools

Dynamic analysis is when testing is performed on code while it is executing. There are two general types of tests, structural and functional testes even called white- and black-box testing.

8.1 A study about dynamic testing

The main difference between structural and functional techniques are that in structural testing the focus is on internal implementation, i.e. a tester is able to see the context of code and in this way implement the tests on a lower level. The functional testing focuses on external behaviour. Here the output of the test is compared with the output expected.

Structural (White-Box) testing

The majority of the traditional testing techniques are based on program analyses such as white-box testing. Structural testing verifies the correct implementation of code units, such as program statements, data structures, algorithms etc.

The advantage of white-box testing is that once a problem is detected, it is also located. White-box testing is most commonly performed by the code developers. This is due to their high knowledge and understanding of code being tested. Once planned coverage has been achieved testing can stop.

Functional (Black-Box) testing

Functional testing involves analyzing the behaviour of the application according to the program specifications and generating test procedures for each function in the application. To be more specific, the main difference is that when doing functional tests the tester is not able to see the source code.

If the requirement for a program is to provide a specific result then a test case is generated to ensure this requirement is provided. Properly written program requirements are of high importance in order to create effective test cases. The tester must fully understand the details of these requirements. If not, the risk of making own analysis increases, making the test procedures less effective. [ED03a]

Another form of black-box testing is usage of specification checklists. These checklists contain information about input-output relationship of external functions which can be used to make tests cases.

Difference between White-box and Black-box

White-box and black-box testing differ in several ways. Some of those are: [Tian05d]

- **Perspective**
  The White-box technique tests the internal implementation, meaning that the code is presented for the tester while black-box testing focuses on external behaviour.

- **Objects**
The White-box technique is generally used for smaller projects or small units of large projects while the black-box technique is usually used in larger projects.

- **Timeline**
  The White-box technique is generally used in the beginning of software development while the black-box technique is more used in later phase of software development.

- **Defect detection and fixing**
  In white-box testing detected faults are easier to fix since the fault usually is located when it is detected. On the other hand white-box technique can miss faults related to the design. Black-box technique is therefore more effective in finding faults within the interface.

- **Tester**
  White-box technique is mostly performed by code developers while black-box technique can be performed by any professional.

### 8.2 Code Coverage

One strategy for improving code quality is by testing the code on its fundamental components, (units). To simplify dynamic testing code coverage is introduced. There are many different types of coverage techniques. These techniques are sometimes very similar or have more than one name. Below is a description of some common coverage methods. [EKCMa]

Code coverage can be perform with GCC’s utility “gcov”, however this is not the case for the C10v2 since the Ubicom SDK does not support it.

**Statement Coverage**

Statement coverage, also known as line coverage and segment coverage, records all of the lines executed during an execution.

**Branch Coverage**

Branch coverage, also known as decision coverage, covers all Boolean expressions. A branch coverage test has to cover both TRUE and FALSE evaluations at least once. Branches in C programs are created by:

- `if`
- `?`
- `for`
- `while`
- `do-while statements`
- `switch case`

The code below shows two possible scenarios, one where the “if” condition variable is true and one where it is false. Whether this code resulting in full branch coverage or not depends on the test case. If the variable “test” is tested for both true/false values a not referenced pointer error will be detected.

```c
int main(int argc, char* argv[])
{
    bool test = true;
```
int var = 5;
int *p = NULL;
if (test)
   p = &var; /*Functional code*/
else
   *p = 10; /*Error. Pointer not referenced*/
return 0;
}

Loop coverage

A do-while loop creates two conditions:
- one time execution
- more than one time execution

The for- and while-loop have a third condition and it is:
- executing the loop zero times
  i.e., the test expression is false when it is evaluated for the first time.

Some faults in loops only occur after a certain number of passes, this coverage technique is of great importance for this type of fault.

Multi-condition coverage

For nested expressions like “if(a && b)” or “if(a || b)” there is an extension of branch coverage method called multi-condition coverage. It requires both “a” and “b” to be TRUE and FALSE sometime during the test execution. In other words, nested expressions create four conditions:
- false/false
- true/false
- false/true
- true/true

A disadvantage of this coverage method is that it could be difficult to determine the minimum set of tests to cover all conditions.

Weak mutation coverage

Weak mutation coverage or relational operator coverage is a coverage technique that reports simple kinds of possible operator faults such as using “<=” instead of “<”. In C the relational operators are ”<”, ”<=”, ”>” and ”>=”. The faults occur when incorrect boundary values e.g. “a < 9” are set instead of ”a < 10”.

8.3 Test tools for dynamic analysis

In this thesis three different dynamic testing tools were examined, LabVIEW, Cantata and CTC. The reason these three programs were chosen is the following:

LabVIEW Already used at Metrima.
Cantata Used at many companies around the world.
CTC Big integration possibilities.

8.3.1 LabVIEW

LabVIEW is a graphical language mainly used for measuring data, data analyses, data presentation and data storage. Thanks to the design of icons and the graphical interface LabVIEW increases the user friendliness and writing applications becomes much easier.

Testing with LabVIEW

LabVIEW is used for Black box testing. A number of signals are generated from a DAQ-card (data acquisition card) in the testing computer. After this operation the C10v2 is read to verify that these signals have been received, see Figure 23.

![Figure 23. How the C10v2 is connected to the test environment](image)

The integration with C10v2 is minimal, since no source code changes are necessary. The test code have to work as a shell around the C10v2, controlling all the inputs and outputs.

8.3.2 CANTATA++

Cantata++ is developed by the IPL and is based on an earlier version Cantata. Cantata++ were developed 1997. This program is considered by many to be a complete software verification solution for C and C++ code. [Cantata1a]

Cantata++ can be integrated with several different IDE’s, but not with the IDE used for C10v2, more about this later in this chapter. This is why Microsoft Visual Studio is used for the example in Figure 24.

![Figure 24. Cantata integrated with Microsoft Visual Studio](image)

The easy integration with different IDEs makes learning stage faster for the developers.
**Dynamic Unit Testing**
Dynamic test cases are written in a Test Script Wizard. This wizard allows the user to create both white and black box tests and also handles exception checking. The wizard generates code that can be edited and when the test code is run the result is shown in the output window.

**Coverage testing**

Many different coverage rule sets are available in Cantata. Own defined rule sets can also be included. By rebuilding and running the program coverage information is generated for all the files specified in the wizard.

**Contact with IPL**

IPL have been very supportive and clear in their answers to our questions. The main question was if Cantata++ could be integrated with Ubicom ip2022 and Unity. By answering a series of questions about the IDE, the target, memory space, communication ways, etc. IPL finally gave us this answer:

”…I am sorry to report that IPL have finally concluded that Cantata++ cannot be made to viably work on your Ubicom target environment.”

This was given to us by Ross Terry and Matt Davis, Software Test Tools Consultants at IPL.

**8.3.3 CTC++ and CMT++**

CTC++ and CMT++ are Testwells solutions for code coverage measurement and dynamic analysis. Execution of CTC++ is done in following three steps. These steps are meant for command line mode of use.

1. **CTC++ Pre-processor**, ctc, is used for compiling the source files and for linking the instrumented program with the CTC++ Library during runtime.
2. When the code is executed coverage and timing history is saved and at the end of the program this information is written to a file.[testwelletc]
3. Finally this file is used to create a human readable report which can be seen either in Excel (tab separated values) or in a HTML-browser (HTML-code).

CTC++ is configured by a textual configuration file for compilers unknown to the program.

Tanks to CTC++ design and structure they can be a future candidate for Metrima's next testing program. The reason why these programs were not tested with C10v2 is that changes in the evaluation copies in these programs are needed to integrate with the C10v2 environment.
9 Compilers

In this chapter we will start by looking at the compiler and decide if this is the best suited compiler for this project. The ip2k-elf-gcc is tested with different option flags and the results achieved by these changes are analysed.

9.1 Is GCC the only solution?

When this question was given to one of the software producers at Ubicom we got the following answer:

“The only compiler(s) that I’m aware of is the one provide with SDK which is GNU based and the GNU compiler. I would not recommend building a project without the SDK. It is your only legitimate sources for Ethernet drivers, and other complex drivers needed to use the on chip I/O devices…”

In this project a lot of modules beside the Ethernet are used, among this modules are an Operating System (IpOS), a TCP/IP Core (ipStack) and an Inter IC communication bus. We kept searching for an alternative compiler but could not find a free compiler better suited for our project. This is why we are concentrating on the GCC provided by Ubicom and trying to use the tool available in the IDE and other testing tools for quality improvement.

9.2 GNU Compiler Collection – GCC

GCC development is a part of the GNU Project, which is directed by the Free Software Foundation [Gcc1a]. When GCC is used, it normally does pre-processing, compilation, assembly and linking. With different option commands we are allowed to change the behaviour for these stages. These commands can be used to display information about the compilation and control overall behaviour. For example, the ‘-E’ option says that nothing is done except the pre-processing. Some of the options control the pre-processor and others the compiler itself. Other options control the assembler and linker. Table 29 shows some of the topics where changes are possible [Gcc1b].

<table>
<thead>
<tr>
<th>Topic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overall Options:</td>
<td>Controlling the output files; executable, object files, assembly files and pre-processed source.</td>
</tr>
<tr>
<td>C Dialect Options:</td>
<td>Controlling the variant of C language compiled.</td>
</tr>
<tr>
<td>Warning Options:</td>
<td>How picky should the compiler be?</td>
</tr>
<tr>
<td>Debugging Options:</td>
<td>Symbol tables, measurements, and debugging dumps.</td>
</tr>
<tr>
<td>Optimize Options:</td>
<td>How much optimization?</td>
</tr>
<tr>
<td>Pre-processor Options:</td>
<td>Controlling header files and macro definitions. Also, getting dependency information for Make.</td>
</tr>
<tr>
<td>Assembler Options:</td>
<td>Passing options to the assembler.</td>
</tr>
<tr>
<td>Link Options:</td>
<td>Specifying libraries and so on.</td>
</tr>
<tr>
<td>Directory Options:</td>
<td>Where to find header files and libraries. Where to find the compiler executable files.</td>
</tr>
<tr>
<td>Target Options:</td>
<td>Change the default options to be able to run a cross-compiler, or an old version of GCC.</td>
</tr>
</tbody>
</table>
Code Gen Options: Specifying conventions for function calls, data layout and register usage.

Table 29. GCC option command categories.

9.2.1 Code Coverage using GCC, gcov

GCC provides built-in Basic Block Coverage. In order to use the gcov test coverage tool extra option flags must be added to produce coverage information:

- **-fprofile-arcs**, when an application is run with -fprofile-arcs option a file by the same name as the original source file is created but with the .da extension in the directory where the source file is located. The .da file contains the data necessary for gcov.

- **-ftest-coverage**, this option causes GCC to identify and track each basic block within the source file. This information is saved in a file with the same name as the source file but with the other extensions, .bb and .bbg. The .bb files contain a list of source files and each function within that file. The .bbg file contains a list of possible branches taken from one basic block to another. Together this information makes it possible for gcov to reconstruct program flow.

For achieving the best result all optimization options should be avoided. These options could modify the execution order or grouping of the code which can make the coverage difficult or impossible. [GCC04a]

This tool can not be used with C10v2 since code optimization in necessary for the SDK compilation, see chapter 9.4.2 Optimization, for more details. Coverage is not supported by ip2k-elf-gcc.

9.2.2 Code Profiling using gprof

To produce the best code profile the profiling module requires that no code optimization is made. Since the code is highly optimized, this option can not be used in our project, see chapter 9.4.2 Optimization. Profiling is not supported by ip2k-elf-gcc.

The gprof application provides several forms of profiling:

- Flat profiling, shows the time spent in each function.
- Call graph, provides a relationship graph, between the functions in the application.
- Annotated source code listing gives each line of the code a number which holds the number of times it has been executed.

Gprof is unlike gcov not part of standard GCC distribution. Gprof is a part of the `binutils package` (a collection of GNU binary tools such as linker(ld) and assembler(as)). In order to cause the application to produce profiling information the following options are needed. [GCC04b]

- **-gp** is used to activate and link profiling libraries.
- **-g** is added for line-by-line profiling. This is a standard option that inserts debugging information. This information is used by gprof to map lines in the executable program to lines in the source file.
- **-finstrument-functions** is used when you want your own written profiling functions to be used. This option makes sure your profiling function is run just after entering and before returning from a function.
- **-ax** generates the extra code needed to perform basic-block profiling. This option is not supported by ip2k-elf-gcc.
9.3 Default Compiler options

The system we are examining is compiled with base options set by Ubicom. These base options are usually enough for finding errors. But for testing and improving the quality of the code “overall options” are needed. These base options are specified in the Makefile.inc, see Figure 25.

```plaintext
.....
# Set the global C flags to pass to gcc.
CFLAGS += -Wall -fno-builtin \ 
${GLOBAL_CFLAGS} \ 
-ffunction-sections \ 
-frename-registers \ 
.....
```

Figure 25. Base compilation options in Makefile.inc

Base Option Descriptions

For a detailed description of the options below see [Gcc1b].

- **-Wall**
  Contains a collection of warning-options.

- **-fno-builtin**
  Makes the compiler to not recognize builtin functions that do not begin with `__builtin_` as prefix. Some of the functions effected include abort, exit, sin, cos and strlen.

- **-Os**
  Code size optimization. The –Os should always be enabled, the SDK will not go through the compilation if there is no optimization.*

- **-g**
  Produces debugging information that is made for debugging.

- **-ffunction-sections**
  Places relative functions and data items into specific sections in the output file. This option is used for optimizations.*

- **-frename-registers**
  Refers to a technique used to avoid unnecessary serialization of program operations. This will benefit processors with many registers.* (ip2022 has 15 registers)

* These options make debugging harder.

9.4 New compiler options suited for the C10v2

The options below are our recommendations suited for C10v2 environment. By generating extra warning messages the programmers are more aware of what could be wrong in the code. This gives them more control over the code and leads to fewer faults.
9.4.1 Control options

Wextra

-Wextra, is the new name for –W. The old name is still supported, but the newer name is more descriptive. This option turns on some extra warnings not included in -Wall, such as missing return values as in the code below.

```c
int func (int x)
{
    if (x < 0)
        return x;
}
```

When a signed and an unsigned value are compared an incorrect result can be produced if the signed value is converted to unsigned. If an unsigned value is compared against zero with ‘<’ or ‘<=’ a warning message is shown.

A warning is also generated if the parameters in the function header do not have a type.

```c
void foo(bar) {} 
```

When an “if” or “else” statement have an empty body a warning is generated.

When compiling with –Wextra following warnings were generated:

- Unused parameter ’idi’.

```c
udp_ping.c line 27
u8_t *ipdatalink_get_hwaddr(struct ipdatalink_instance *idi)
{
    return NULL;
}
```

- Comparison between different types.
  File TcpApp.c, line 371

```c
u16_t n;
u8_t *in_buffer
for( n=0; n<in_buffer[1]+2; n++) {
...
```

- Empty body in an else-statement
  File alarm_stack.c, line 349

```c
for(i = 0; i < 6 && older == 0; i++) {
    if(date_and_time[i] < lowest_priority_time[i])
        older = 1;
    else if(date_and_time[i] < lowest_priority_time[i]) i = 10;
        else;
}
```

Werror
When many files are compiled the `-Werror` options can be used to change all warnings into errors so that no warnings are missed.

**Pedantic**

`-pedantic`, warns for code that are not following a strict ISO standard. At Ubicom the native standard is ISO C89. Pedantic warnings are disabled in the expression that follows `__extension__`. When this option was added to the compiler the compilation did not succeed. This is caused by the assembler files included in the project. Below are some of the warnings generated from the files compiled before the compiler read the included assembler files. This is only warnings from code written by the developers at Metrima.

```plaintext
realtime.h:25:1: warning: "STATUS" redefined, this is the location of the previous definition
/cygdrive/C/Ubicom/sdk/pkg/ipOS/include/ip2k/ip2022.h:74:1:
main.c:36:1: warning: C++ style comments are not allowed in ISO C89 (this will be reported only once per input file)
```

**ISO code standards**

`-std=iso9899:199409` or `-std=iso9899:1999`, used to select the language standard. This option can not be used since then interface code for the ipModules can not pass the compilation. The pedantic flag and the different standards are some of GCC’s most significant options for testing and should be supported by Ubicom.

**9.4.2 Optimization**

By adding optimization options the compiler tries to improve performance and/or code size. The price for this improvement can be lost of ability to use coverage and profiling testing and debugging. This was the case in our project. There are different levels of optimization, see Table 30. To get best result from `-gprof` and `–gcov` the recommended level is, -O0, no optimization. The optimization options contain a number of flags. The flags turned on by the controls vary for different targets. Visit [Gcc1c] for more details.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-O0</td>
<td>Do not optimize. If multiple <code>-O</code> options are used, with or without level numbers, the last such option is the one that is effective.</td>
</tr>
<tr>
<td>-O / -O1</td>
<td>Optimize. The compiler turns on some of the options supported by the target.</td>
</tr>
<tr>
<td>-O2</td>
<td>Optimize even more. Nearly all supported flags are added to the compiler.</td>
</tr>
<tr>
<td>-O3</td>
<td>By adding <code>-O3</code> all optimizations specified by <code>-O2</code> are included and it also turns on the <code>-finline-functions</code> and <code>-frename-registers</code> options.</td>
</tr>
<tr>
<td>-Os</td>
<td>Size optimization. All optimizations specified by <code>-O2</code> are included beside optimizations that typically increases the size. Other optimizations designed to minimize the code size is also done.</td>
</tr>
</tbody>
</table>

Table 30. Optimization option levels.
Since there are code quantity problems for further development we helped the developers to find the options needed for size optimization. Most of these options make debugging harder and sometimes impossible. As mentioned earlier the SDK can not be compiled without optimization, i.e. -O0 can not be used.

-**O/ -O1**

The option -Os was replaced with -O and the following errors were generated by the linker.

```
ip2k-elf-ld: section .reset [0201ffe0 -> 0201fff7] overlaps section .text [02010404 -> 02020a13]
ip2k-elf-ld: section .config [02020000 -> 0202007f] overlaps section .text [02010404 -> 02020a13]
made: *** [next.elf] Error 1
```

This means that total size of the executable file is larger than the available memory in the target.

-**O2 and -O3**

When the code is compiled with -O2 and -O3 no errors are generated. But when the options necessary for –gcov and –prof are added we get the same result as in “-O/ -O1”, the size of the executable file is larger than the available memory in the target.

-**fomit-frame-pointer**

-fomit-frame-pointer do not keep the frame pointer in a register for functions that do not need one. This avoids the instructions to save, set up and restore frame pointers. By using this option the code size decreases by 7.5 %, see Table 31.

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Section</td>
<td>size</td>
</tr>
<tr>
<td>.text*</td>
<td>57630</td>
</tr>
<tr>
<td>.data*</td>
<td>758</td>
</tr>
<tr>
<td>.bss*</td>
<td>570</td>
</tr>
<tr>
<td>Total</td>
<td>58958</td>
</tr>
</tbody>
</table>

Table 31. C10v2 memory usage.

* .text, the code size, in decimal.
* .data, the initialized data section size.
* .bss, the uninitialized data section size.

Other options for code size optimization:
- By switching heap memory allocation algorithm for the ipOS from best-fit to first-fit the size of the code decreases.
- By adding –relax, redundant (duplicated) page instructions such as branches and calls will be removed during the linking.
- There are other options for code optimization like -Os and -frename-registers. These options are set by default, and no configurations are necessary.

**Future investments**
The space freed by using the recommended options was used for developing new functionality for power-line communication. The size problem can be solved by changing chip from the ip2022 to ip3023 which has a 4 Mb Flash memory. Ip3023 also supports code coverage and code profiling needed for application testing. This has not been done for several reasons. The main reasons are:

- Metrima’s factory in Karlskrona does not have the machines necessary to integrate the circuit to the platform. The new circuit has a BGA (Ball Grid Array) connection which requires new machinery.
- The ip3023 is not a one chip solution, meaning that the whole platform have to be redesigned to suit the ip3023.
10 Conclusions and Future work

During this thesis we run into several issues regarding software application development in embedded systems. The largest issue was the memory available in the IP2K chip. After downloading the code into the chip, approximately 7 Kbytes of available memory were left to our disposal. With this in mind we had to find appropriate methods and tools to perform different tests that would improve the quality of C10v2 code.

Results of our research showed that C10v2 environment could not easily be integrated with dynamic analysis tools. To make this possible the development companies of the dynamic analysis tools we examined had to make changes in their evaluation copies. This, unfortunately, was a cost we were not able to take and which made us to discard this solution for the C10v2.

The issues with dynamic analysis tools made us focus on static analysis methods/tools and the IP2K compiler. This research resulted in detailed discussion about Code standards, Code inspections, Static analysis tools and the GCC Compiler. By applying some of the static analysis methods mentioned in these chapters we were able to find inconsistencies within the bNet protocol and C10v2 code. Errors like this can be very difficult to find since the code seems to be flawless. Other errors that were found by code reading techniques were: not responding messages (missing else-statement), variables declared twice (#define statement), pointers that were not checked for NULL before use, error-prone code like “=” within an if-statement, incorrect code comments etc, see Appendix A. By complying with the rules in chapter 5 Code Standards, readability, testability and portability in new C10v2 code will be increased. The changes made in IP2K compiler resulted in code size reduction by more then seven percent and new flags making the compilation more strict.

Early in this thesis we realized that complying with the rules and standards was not always the case at Metrima. As a future work this have to be followed by everyone working in the C10v2 project. Further, static analysis tools should be used early in the code development, preferably at unit level. By performing code inspections method several serious errors were found in another project at Metrima. This method has never been used in C10v2 project. The results should be enough to encourage the use of this method in the C10v2 project.
11 References

11.1 ARTICLES


[CÅAB04a] Coding and documentation standard, Cons T Åhs, Alexander Bottema, 2004, Uppsala University

[CÅAB04b] Coding and documentation standard, Cons T Åhs, Alexander Bottema, 2004, Uppsala University, rule 2.4.2


[Ubi04a] Ubicom, StreamEngine Router Gateway, A new class of product for family and small office applications, IP3K-DPB-RGW-11 -04
11.2 BOOKS


11.3 INTERNET


[Ubi05c] IP2K-DUG-ADVWLESS-12, IP2000 Network Processor Advanced Wireless

[Ubi05d] IP2K-DPB-ADVWLESS-13, IP2000 Series Advanced Wireless Kit,


12 Appendix

Appendix A – Code inspection results using the check-list and ad-hoc method

Execution

The code inspection was performed using Metrimas check-list and ad-hoc method on following files:

- Msg_handler.c
- TaskManager.c
- Alarm_stack.c
- MEM_map.h
- BNet-specifikation

The inspection results

**Msg_handler.c**

Line 977  According to the bNet-specification all bNet messages must have a return statement. Following bNet-message does not have a return statement.

```c
    case IP_READ_VENDOR:
    if (data[1] == 2) {
        EEPROM_readByte(SCENIX_DHCP_BASE, &data[1]);
        EEPROM_readString(SCENIX_DHCP_BASE+1, &data[3], data[1]);
    }  
    break;
```

Line 159  According to the bNet-specification container data[1] have a length of 30. In the code below this number is 29.

```c
    case SCRATCHPAD_WRITE:
    if ( (data[1] >= 4) && (data[1] <= 29) ){ // Check for valid length
        //Checks that address + data len is within allowed boundary
        if( ( *(u16_t*) &data[3]) + data[1]-3 ) <= SCRATCH_PAD_DATA_SIZE ) {
            EEPROM_writeString(SCRATCH_PAD_DATA + *(u16_t*) &data[3], &data[5], data[1]-3);
            data[1] = 1;
        }
        else{   // Tried to write outside scratchpad, prepare a error msg
            data[2] = 255;
            data[1] = 1;
        }   
    }
```

Line 228  According to the code comment the reboot of the system should be done after three seconds (TICK_RATE). In the code the reboot time is one second.

```c
    case RESET_MSG:
    if(data[1] == 2 & data[3] == 0){
```
data[10] = SOFT_RESET;
writeLastMainFrameDownData(SOFT_RESET); // Skriver orsak till omstart, logdata, tid/datum
EEPROM_writeByte(FORCED_BOOT_BY, &data[10]); // Vilken funktion som orsakade reseten
oneshot_init(&icp_timer);
// Reboot after three seconds
oneshot_attach(&icp_timer,TICK_RATE*1,(void(*)(void)) &timer_reboot,0);
}
i = 0xFF;                              // Sätt att noden ska göra factory reset efter omboot
EEPROM_writeByte(FACTORY_DEFAULT, &i); // skriv att noden behöver göra factory default nästa gång
data[10] = HARD_RESET;
writeLastMainFrameDownData(HARD_RESET); // Skriver ner orsak till omstart, logdata, tid/datum
EEPROM_writeByte(FORCED_BOOT_BY, &data[10]); // Vilken funktion som orsakade reseten
oneshot_init(&icp_timer);
// Reboot after three seconds
oneshot_attach(&icp_timer,TICK_RATE*1,(void(*)(void)) &timer_reboot,0);
}
data[10] = EXTERNAL_RESET;
writeLastMainFrameDownData(EXTERNAL_RESET); // Skriver ner orsak till omstart, logdata, tid/datum
EEPROM_writeByte(FORCED_BOOT_BY, &data[10]); // Vilken funktion som orsakade reseten
oneshot_init(&icp_timer);
// Reboot after three seconds
oneshot_attach(&icp_timer,TICK_RATE*1,(void(*)(void)) &ext_timer_reboot,0);
}
else{
data[2] = 255;
}
data[1] = 1;
writeData(com_module, data);
break;

Line 447 According to Metrimas code convention no hard-typed address values are allowed. Last parameter in this function call is hard-typed.

EEPROM_writeString(S0_MAP_LAST_PERIOD_TARIFF_OFFSET + (data[3] * 2), &data[4], 2);

Line 498 The constants below does not have same name in the bNet-specification and C10v2 code.

Code: INCLUDED_USE_READ_MSG
bNet-spec: INCLUDED_USE_READ

TaskManager.c

Line 30-32 Following include files are included twice.

#include "hwUart.h"
#include "S0.h"
#include "tcpApp.h"

MEM_map.h

Line 117 Parenthesis not used when declaring a macro.

#define TEMPERATURE_HOUR_SIZE NO_OF_TEMPERATURE_SENSORS * TEMPERATURE_SIZE

Line 216 Parenthesis not used when declaring a macro.
#define S0_RESET_BILL_VALUE_END
(u16_t)S0_RESET_BILL_VALUE_OFFSET +
S0_RESET_BILL_VALUE_TOT_SIZE

## Alarm_stack.h, Alarm_stack.c

Following functions are declared but never used.

**Line 30** Alarm_stack.h

```c
void initAlarm(void);
```

**Line 42** Alarm_stack.h and line 280 Alarm_stack.c

```c
void saveAlarmStack(void);
```

**Line 31** Alarm_stack.h and 254 Alarm_stack.c

```c
void recoverAlarmStack(void);
```

## bNet-specifikation

**Page 16**

Sub ID: S0_AND_CONSUMPTION_SET_MSG, # 83, V20.1
Length can only have value 3 and 18. In the bNet-specification length is described as 3-18. It should be 3, 18.

**Page 24**

Sub ID: SCRATCHPAD_READ, # 165.
Field is described as 0, 1, 2, 0, 4.
It should be 0, 1, 2, 3, 4.

**Page 25**

Sub ID: INCLUDED_USE_READ, # 168.
In the request message the variable LEN has value range 1.
This value should be 2.

**Page 30**

Sub ID: START_TEMP_SENSOR_SEARCH_FROM_D80, # 175.

The bNet-specification for message 175 is
START_TEMP_SENSOR_SEARCH_FROM_D80. In the C10v2 code the message 175 is referred as AD_POWER. This can lead to misunderstandings.
Appendix B - Configuration file for PC-Lint

Following configuration file contains the configurations needed to be able to lint the C10v2 code.

//**********************************************************
//                begin included file
//**********************************************************
// Compiler definitions

// Header file locations
-i"C:\Amir\StaticTools\C20V2_KOD\PROJECTS\config"
-i"C:\Amir\StaticTools\C20V2_KOD\PROJECTS\app"
-i"C:\Amir\StaticTools\C20V2_KOD\PKG\upOS\include\ip2k"
-i"C:\Amir\StaticTools\C20V2_KOD\PKG\upOS\src\ip2k"
-i"C:\Amir\StaticTools\C20V2_KOD\PKG"
-i"C:\Amir\StaticTools\C20V2_KOD\PKG\upOS\include"
-i"C:\Amir\StaticTools\C20V2_KOD\PKG\upConfigure\include"
-i"C:\Amir\StaticTools\C20V2_KOD\PKG\upFile\include"
-i"C:\Amir\StaticTools\C20V2_KOD\PKG\upFile\src"
-i"C:\Amir\StaticTools\C20V2_KOD\PKG\upDial\include"
-i"C:\Amir\StaticTools\C20V2_KOD\PKG\upDial\src"
-i"C:\Amir\StaticTools\C20V2_KOD\PKG\upDNSClient\include"
-i"C:\Amir\StaticTools\C20V2_KOD\PKG\upEthernet\include"
-i"C:\Amir\StaticTools\C20V2_KOD\PKG\upEthernet\src"
-i"C:\Amir\StaticTools\C20V2_KOD\PKG\upI2C\include"
-i"C:\Amir\StaticTools\C20V2_KOD\PKG\upI2C\src"
-i"C:\Amir\StaticTools\C20V2_KOD\PKG\upManage\include"
-i"C:\Amir\StaticTools\C20V2_KOD\PKG\upNE2000"
-i"C:\Amir\StaticTools\C20V2_KOD\PKG\upNE2000\src"
-i"C:\Amir\StaticTools\C20V2_KOD\PKG\upPPP\include"
-i"C:\Amir\StaticTools\C20V2_KOD\PKG\upPPP\src"
-i"C:\Amir\StaticTools\C20V2_KOD\PKG\upSecurity\include"
-i"C:\Amir\StaticTools\C20V2_KOD\PKG\upSMTPClient\include"
-i"C:\Amir\StaticTools\C20V2_KOD\PKG\upSPI\include"
-i"C:\Amir\StaticTools\C20V2_KOD\PKG\upSPI\src"
-i"C:\Amir\StaticTools\C20V2_KOD\PKG\upStack\include"
-i"C:\Amir\StaticTools\C20V2_KOD\PKG\upStack\src\ip2k"
-i"C:\Amir\StaticTools\C20V2_KOD\PKG\upStorage\include"
-i"C:\Amir\StaticTools\C20V2_KOD\PKG\upStorage\src"
-i"C:\Amir\StaticTools\C20V2_KOD\PKG\upTFTP\include"
-i"C:\Amir\StaticTools\C20V2_KOD\PKG\upTFTP\src"
-i"C:\Amir\StaticTools\C20V2_KOD\PKG\upTime\include"
-i"C:\Amir\StaticTools\C20V2_KOD\PKG\upTime\src"
-i"C:\Amir\StaticTools\C20V2_KOD\PKG\upUART\include"
-i"C:\Amir\StaticTools\C20V2_KOD\PKG\upUART\src"
-i"C:\Amir\StaticTools\C20V2_KOD\PKG\upUSB\include"
-i"C:\Amir\StaticTools\C20V2_KOD\PKG\upUSB\src"
-i"C:\Amir\StaticTools\C20V2_KOD\PKG\upWeb\include"
-i"C:\Amir\StaticTools\C20V2_KOD\PKG\upWeb\src"
+libdir(C:\Amir\StaticTools\C20V2_KOD\PROJECTS\config);
+libdir(C:\Amir\StaticTools\C20V2_KOD\PKG\upOS\include\ip2k);
+libdir(C:\Amir\StaticTools\C20V2_KOD\PKG\upOS\src\ip2k);
+libdir(C:\Amir\StaticTools\C20V2_KOD\PKG);
+libdir(C:\Amir\StaticTools\C20V2_KOD\PKG\upInclude);
+libdir(C:\Amir\StaticTools\C20V2_KOD\PKG\upConfigure\include);
+libdir(C:\Amir\StaticTools\C20V2\KOD\PKG\ipFile\include);
+libdir(C:\Amir\StaticTools\C20V2\KOD\PKG\ipFile\src);
+libdir(C:\Amir\StaticTools\C20V2\KOD\PKG\ipDial\include);
+libdir(C:\Amir\StaticTools\C20V2\KOD\PKG\ipDial\src);
+libdir(C:\Amir\StaticTools\C20V2\KOD\PKG\ipDNSClient\include);
+libdir(C:\Amir\StaticTools\C20V2\KOD\PKG\ipEthernet\include);
+libdir(C:\Amir\StaticTools\C20V2\KOD\PKG\ipEthernet\src);
+libdir(C:\Amir\StaticTools\C20V2\KOD\PKG\ipI2C\include);
+libdir(C:\Amir\StaticTools\C20V2\KOD\PKG\ipI2C\src);
+libdir(C:\Amir\StaticTools\C20V2\KOD\PKG\ipManage\include);
+libdir(C:\Amir\StaticTools\C20V2\KOD\PKG\ipNE2000);
+libdir(C:\Amir\StaticTools\C20V2\KOD\PKG\ipNE2000\src);
+libdir(C:\Amir\StaticTools\C20V2\KOD\PKG\ipPPP\include);
+libdir(C:\Amir\StaticTools\C20V2\KOD\PKG\ipPPP\src);
+libdir(C:\Amir\StaticTools\C20V2\KOD\PKG\ipSecurity\include);
+libdir(C:\Amir\StaticTools\C20V2\KOD\PKG\ipSMTPClient\include);
+libdir(C:\Amir\StaticTools\C20V2\KOD\PKG\ipSPI\include);
+libdir(C:\Amir\StaticTools\C20V2\KOD\PKG\ipSPI\src);
+libdir(C:\Amir\StaticTools\C20V2\KOD\PKG\ipStack\include);
+libdir(C:\Amir\StaticTools\C20V2\KOD\PKG\ipStack\src\ip2k);
+libdir(C:\Amir\StaticTools\C20V2\KOD\PKG\ipStorage\include);
+libdir(C:\Amir\StaticTools\C20V2\KOD\PKG\ipStorage\src);
+libdir(C:\Amir\StaticTools\C20V2\KOD\PKG\ipTFTP\include);
+libdir(C:\Amir\StaticTools\C20V2\KOD\PKG\ipTFTP\src);
+libdir(C:\Amir\StaticTools\C20V2\KOD\PKG\ipTime\include);
+libdir(C:\Amir\StaticTools\C20V2\KOD\PKG\ipTime\src);
+libdir(C:\Amir\StaticTools\C20V2\KOD\PKG\ipUART\include);
+libdir(C:\Amir\StaticTools\C20V2\KOD\PKG\ipUART\src);
+libdir(C:\Amir\StaticTools\C20V2\KOD\PKG\ipUSB\include);
+libdir(C:\Amir\StaticTools\C20V2\KOD\PKG\ipUSB\src);
+libdir(C:\Amir\StaticTools\C20V2\KOD\PKG\ipWeb\include);
+libdir(C:\Amir\StaticTools\C20V2\KOD\PKG\ipWeb\src);

//**********************************************************
// Project definitions
//**********************************************************
-#dARCH IP2K // Compiler does this from MCU definition
-#dIP2K

//**********************************************************
// Error reporting suspension
//**********************************************************
-#wlib(1)

//**********************************************************
//                end included file
//**********************************************************