THESIS ON INFORMATICS AND SYSTEM ENGINEERING C118

Embedded Software Solutions for Development of Marine Navigation Light Systems

ERKKI MOORITS

TUT
PRESS

TALLINN UNIVERSITY OF TECHNOLOGY
Faculty of Information Technology
Department of Computer Engineering

Dissertation was accepted for the defence of the degree of Doctor of Philosophy in Computer and System Engineering on May 20, 2016.

Supervisor: Prof. Gert Jervan
Department of Computer Engineering
Tallinn University of Technology, Estonia

Aivar Usk
Cybernetica AS, Estonia

Opponents: Prof. Jean Marc Thiriet
Université Grenoble Alpes, France

Prof. Peter Enoksson
Chalmers University of Technology, Sweden

Defence of the thesis: September 9, 2016

Declaration:
Hereby I declare that this doctoral thesis, my original investigation and achievement, submitted for the doctoral degree at Tallinn University of Technology has not been submitted for any academic degree.

/Erkki Moorits/

ISSN 1406-4723
ISBN 978-9949-83-011-4 (publication)
ISBN 978-9949-83-012-1 (PDF)

INFORMAATIKA JA SÜSTEEMITEHNIKA C118

Sardtarkvara lahendused valgusnavigatsiooni süsteemide arendusel

ERKKI MOORITS

TTÜ
KIRJASTUS

TABLE OF CONTENTS

LIST OF PUBLICATIONS

1. E. Moorits, G. Jervan, "Low resource demanding FOTA method for remote AtoN site equipment", Proceedings of the OCEANS '10 MTS/IEEE Seattle, 2010, pp. 1 – 5.
2. E. Moorits, A. Usk, "A Numerically Efficient Method for Calculation of the Angle of Heel of a Navigational Buoy", Proceedings of the 12th Biennial Baltic Electronic Conference BEC2010: 2010, pp. 357 – 360.
3. E. Moorits, A. Usk, T. Kõuts, "Wave Height Measurement as a Secondary Function of Navigational Buoys", Proceedings of the OCEANS '11 MTS/IEEE Kona, 2011, pp. 1 – 5.
4. E. Moorits, G. Jervan, "Profiling in Deeply Embedded Systems", Proceedings of the 13th Biennial Baltic Electronic Conference: 2012 13th Biennial Baltic Electronics Conference (BEC2012), 2012, pp. 127 – 130.
5. E. Moorits, A. Usk, "Buoy Collision Detection", Proceedings of the 54th International Symposium Electronics in Marine ELMAR-2012, 2012, pp. 109 – 112.

Author's contribution and objectives of the papers to the publications is as follows:

1. The objective of the paper was to develop a FOTA method suitable for AtoN devices, especially for flashers and telematics modules. The paper describes the FOTA method developed and tested by the author. Author proposed the idea to use an external buffer memory and also implemented new bootloader and supporting software for the telematics module. The author prepared the paper for publications and presented it at the conference.
2. The objective of the paper was to develop a heel angle measurement method suitable for AtoN devices, which is used on navigational buoys. This improvement gave valuable information for development of buoy onboard light sources and also some information for ships about decreased visibility range. The paper describes the method, developed and tested by the author that allows to measure the heel angle of a buoy. The author proposed mathematical simplifications and algorithms that allows to use trigonometric functions on 8-bit microcontrollers without significant overhead. This method is suitable for microcontrollers that require low

energy consumption. The author prepared the paper for publications and presented it at the conference.

3. The objective of the paper was to develop a wave height measurement method suitable for AtoN buoys. This method used on server side and is intended to inform ships about decreased visibility range of a buoy light. All input data is collected and transferred by using buoy onboard telematics module. The paper describes the algorithm used for navigational buoys for wave height measuring. Reference wave height data for tests obtained in collaboration with the Marine Systems Institute at TUT. The author's contribution in this paper was the development and implementation of the mathematical solutions and server side software for the wave height measuring in buoys-server systems. The author prepared the paper for publications and presented it at the conference.
4. The objective of the paper was to develop a profiling method that can be used in AtoN systems in software developement. This method was needed as aid for finding hot spots and bottlenecks in the software of the low-power Telematics Module. The paper describes solution for profiling embedded programs, which are running in memory constrained systems. In proposed solution, which involves slight modification of GNU compiler, is sent profiling data to external program that capture profiling data. The author's contribution is the development and testing of the solution, preparation the paper for publications and presentation it at the conference.
5. The objective of the paper was to develop a collision detection method for navigational buoys, this improvement allows to trigger an alarm about ship and buoy collisions. The paper describes the method that allows to detect collisions with a buoy and other objects. The author developed and tested the filters and algorithms that are suitable to detect collision by using acceleration measured by onboard acceleration sensor. The author also prepared the paper for publications and presented it at the conference.

LIST OF ABBREVIATIONS

3G	3rd generation of mobile telecommunications technology
AD	Analogue-to-Digital
AES	Advanced Encryption Standard
AIS	Automatic Identification System
AVR	Modified Harvard architecture 8-bit RISC single chip microcontroller, manufactured by Atmel
AtoN	Aid(s) to Navigation, in this thesis it refers to nautical navigation
BDD	Behaviour-Driven Development
CISC	Complex Instruction Set Computer
CPU	Central Processing Unit
DMA	Direct Memory Access
DSP	Digital Signal Processor
EEPROM	Electrically Erasable Programmable Read-Only Memory
EPROM	Erasable Programmable Read-Only Memory
FOTA	Firmware Over-the-Air
GCC	In this thesis it refers mostly to C compiler from GNU Compiler Collection
GPRS	General Packet Radio Service
GPS	Global Positioning System, in this thesis it refers to user side receivers
GSM	Global System for Mobile Communication
IC	Integrated Circuit
ICE	In-Circuit Emulator, in this thesis it refers to debugging hardware part, typically JTAG
IDE	Integrated Development Environment
IO	Input/Output
ISP	In-System Programming

JTAG	Joint Test Action Group – standard test access port
LAN	Local Area Network
LED	Light Emitting Diode
MCU	Microcontroller unit – small computer on a single integrated circuit
MISRA	Motor Industry Software Reliability Association
NMT	Nordic Mobile Telephony
OCD	On-Chip Debugger, in this thesis it refers to debugging software part, which is on the PC and uses ICE
PC	Personal computer, an general purpose computer
RAM	Random-Access Memory
RISC	Reduced Instruction Set Computer
ROM	Read-Only Memory
RS-485	Standard for defining the electrical characteristics of drivers and receivers for use in balanced digital multipoint systems
SPI	Serial Peripheral Interface Bus
SRAM	Static Random-Access Memory
TDD	Test-Driven Development
TCP/IP	Transmission Control Protocol/Internet Protocol
TM	Telematics Module
UML	Unified Modelling Language

1. INTRODUCTION

The progress in the semiconductor industry has been the main reason for replacing old mechanical devices with electronic analogues. In cars, for example, many mechanical parts have been replaced by electronic counterparts, and in aviation, fly-by-wire systems have been used instead mechanical ones for many decades [28]. In marine navigation light systems, old incandescent light bulbs that had complex mechanical light bulb replacement systems, heavy weight reflectors and bulky lenses, have been replaced with relatively reliable LED's (Light Emitting Diode) with small reflectors and electronic control systems [41, 42]. While initially electronic analogues were relatively simple circuits and in some cases it was not even a digital circuit, then in the end of 1970's, several different electronic control and monitoring systems were quite widely introduced [28]. Recently developed systems had at least one microprocessor unit. The success of microprocessors is due to the fact that it is possible and also sometimes much easier to implement several different functions in software than it is in hardware. This means that less hardware components has to be used and it reduces power consumption of the device. One good example of the hardware functionality replacement with software and then resulting in reduction of hardware complexity is the software-defined radio (SDR) [56]. All recent mobile phone base stations are based on SDR and it is possible to extend radio base station functionality mostly by software upgrade. However, replacing hardware functions with software rises the complexity of the software.

The amount of transistors in top end microprocessors and microcontrollers is increasing according to the Moore's law [64], i.e. doubling in every 18 months. With the increase of number of transistors, also computational power rises which sometimes, unfortunately, happens at the expense of reliability [16, 97]. Contrary to the rise in the amount of transistor and computational performance, power consumption and cost decreases. It is not uncommon that performance of relatively small microcontroller is comparable to or even higher than the 20-year old computer. Furthermore, it is possible to use microcontrollers, which have DSP (Digital Signal Processor) extensions, in complex signal processing applications where the small power consumption is essential, for example, in medical electronics. Since the mid 1990's, different hardware modules have been integrated with microcontrollers. For example, most microcontrollers have at least one configurable serial interface, while other microcontrollers have internal AD (analogue-to-digital) converters with several input channels. Other microcontrollers have a real-time clock, hardware for cryptographical encoding and decoding functions, PWM (Pulse-Width Modulation) modulators and even on chip voltage regulators. However, in terms of hardware functionality 8, 16 and lower end 32-bit microcontrollers, which are produced in the last decade, are not significantly different from each other. Lot of them have comparable

functionality, as well as power consumption, although the main difference is performance, mostly in 16 and 32-bit mathematical functions.

Compared to the similar microcontrollers from some decades ago, modern microcontrollers have more processing power and a lot of additional hardware. This additional hardware allows to use many different safety providing add-ons or add additional tasks to the device. It also allows to use much more powerful microcontrollers in places where it is essential that the device has minimal power consumption. Higher performance microcontrollers that have many additional features, in turn, may attract device manufacturers or software developers to add many different features into existing devices. However, additional functionality dramatically increases complexity of the microcontroller firmware and also significantly increases the time required for firmware testing, and it requires different development tools and testing methodology [82]. Inevitably, the growth of complexity reduces reliability of the firmware [21]. While microcontrollers allow to use relatively large and complex programs, it is very difficult to create such small microcontroller based systems that would be just as dependable as the large system with many internal protection mechanisms.

1.1. Motivation

All marine visual navigational aids are heavily dependent on the weather conditions, which have mostly negative influence to visibility range. Therefore it is wise to inform users and supporting staff about decreased visibility range, for this purpose buoys have heel angle [67] and wave heights [69] (by using server side support) measurement capability. Large number of buoys can also detect collisions [68]. In addition, several multifunctional modules exists which, for example, have an integrated TM (Telematics Module) and flasher. However, these additions and supporting applications increase significantly complexity of the program and are also potential cause of errors. Due to the increase of the TM complexity and flasher software, it is not very realistic to expect that tests that are carried out in the laboratory environment reveal all bugs. Therefore, TM and flashers have also remote software updating capability [65]. Also the majority of the AtoN devices are used in the places where power is limited, but there are requirement for minimal power consumption. Therefore, the best method to detect code section with high CPU or IO usage, and hence energy consumption is profiling [66]. Due to the complexity of the AtoN modules itself, environmental conditions and other limitations, it is necessary to use low power microcontrollers, different software developement approaches, software tools and solutions for the microcontroller software which is used in the AtoN systems.

Due to the limited capabilities of microcontrollers it is not feasible to use exactly the same programming languages, development tools and testing methods in embedded systems that are used for software development in conventional computer programs. Typically, tools and development methods

that are used in embedded software development, are in some way limited and have less functionality and, therefore, it is not possible to use all available testing methods. Embedded software development has some issues that are described below.

As any other software, embedded software may contain bugs, some of them are originating from coding, some are from task formulation, and some are related to underlying system, like hardware, kernel or libraries. In addition to hardware bugs or faults the remaining bugs are related with software. The most difficult bugs to detect and repair belong to the kernel or libraries. To this category belongs also memory corruption and fragmentation issues. While memory corruption bugs are relatively common in embedded systems, the fragmentation is not so common. Memory corruption can happen in two different ways. First one is caused by invalid pointers, and the second one is caused by stack overflow, which may happen even during normal program operation; the last one is quite common for larger embedded software. Pointer related bugs can be discovered by static code inspection tools, mostly by *lint* and its derivatives [31, 110]. Many latest compilers allow some pointer checking as well. Pointer related bugs can also be found by manual code inspection. Stack overflows can sometimes be detected by compilers, which have such capability, however, this is quite new addition to compilers [29]. It is possible to reduce significantly stack overflow caused effects by adding additional memory after each thread block [15]. Yet, this approach may mask stack overflows and work only with kernels that have such support. In multithreaded environment, it is possible to detect stack overflows before or after context switch by checking guard (also called “canary”) pattern in stack area end or by checking stack pointer value and comparing it with maximum stack address. Both approaches add some overhead to scheduler and require several bytes of free memory in thread structures [85], nevertheless, these methods are not usable in non-threaded programs. Without threading or if threading does not have stack overflow detection capabilities, the stack overflows are mostly detected manually by trial and error. Manual overflow detection usually has a great disadvantage, in order to find the exact location of the bug, some source code modification is needed, however, any source code modification may cause stack overflow bug to change its place.

Memory fragmentation issues in embedded software are related to non-regular memory allocations and deallocations. These allocations may even occur in normal program execution. In multitasking programs, it is difficult to foresee all fragmentation occasions [28, 71, 72, 82]. Therefore, to avoid memory fragmentations, it is preferred not to use any dynamic memory at all. Described issues are quite rare in desktop computers where the kernel is a protection layer between hardware and higher level software. This layer can also report and protect faulty memory access by using kernel based memory fault detection on special hardware such as MMU (Memory Management Unit). MMU and supporting kernel, however, is absent in smaller microcontrollers. The absence of such fault detection support causes major problems in embedded

software development process and this is the most noticeable shortage on all 8-bit microcontrollers. This in turn makes the program writing and testing for smaller microcontrollers quite complex task.

Software testing in embedded systems is also more complicated and time consuming than it is in desktop computers [28]. In desktop computer software development relies heavily to automated testing (unit tests) but large number of small embedded software is tested manually. Unfortunately, this is time consuming and error prone. The main reason for using manual testing is that it is difficult to create automated tests for hardware related code. In embedded systems, it is possible to carry out some testing by using emulators, simulators and OCDs (On-Chip-Debuggers), but none of them is capable of debugging non trivial multitasking programs and usually interfere with program real-time behaviour. On the other hand, debuggers that are used in desktop computer program development have much less above described side effects and have more functionality than debuggers that are used in embedded systems.

The differences between development and deployment platforms play a significant role of the embedded software development. The differences can be divided into two main categories – differences between the software platforms and differences between the hardware platforms.

In many cases, the difference between the software platforms is the difference between kernels, operating systems or libraries. In the best case, the differences are so small that it is not required to change any program code or are limited only to some missing functions or headers, which can be detected by compilers or linkers, and are relatively simple to fix. In the worst case, most functions are present but may behave slightly different or have different side effects. Bugs that are caused by lastly mentioned differences are much more difficult to find and fix. Also software platform differences become important when it is required to use automated tests; a code that is used by unit tests should compile on different hardware and software platforms.

Another significant difference between embedded systems and desktop computer is the hardware access. In embedded systems, it is relatively easy to access the lower level hardware but in desktop systems, such operations are limited to privileged users and special functions. This access also includes the use of hardware based watchdog timers which usually is different on embedded and on non embedded systems. While embedded systems access directly watchdog hardware, use non embedded systems special drivers or kernel functions to access to this hardware. Also embedded systems and desktop computers have available very different power saving profiles and program access to these profiles. While most embedded programs take into account energy consumption and selects most appropriate power profile, but in desktop computers and their programs, the power consumption is rarely a concern. Using profile that consume less energy, in turn, affects program structure and algorithms.

In between to the above mentioned hardware and software differences are such issues that are caused by hardware but play role in software. For example, when communicating with other devices or computers over the network is the main problem that difference architectures may have different word length and byte order (endianness). PC (Personal Computer) usually has 32 or 64-bit word length, but small embedded systems typically have 8 or 16-bit word length. This area also includes program optimisations, which depends on underlying architecture.

Despite the above mentioned issues that are mostly related to the IO (Input/Output), memory and debugging, the embedded system software development has several similarities to the desktop computer software development. These are mainly due to the similarity of standard shared libraries and programming languages. Most shared libraries have nearly the same base functionality, although same library in embedded system has usually fewer functions than in desktop systems, but it is possible to use majority of language features in embedded software development.

Above mentioned shortcomings and differences are the main sources of programming errors, and unfortunately most of them do not surface before final program release. Since most small embedded systems do not have any hardware or software mechanism to prevent fatal errors, the safest way is to use more powerful microcontrollers, which have certain fault protection mechanisms. However, many embedded systems have quite limited power budget and some of them are installed in remote sites. In these cases it is not reasonable to use more powerful microcontroller.

1.2. Problem Formulation

Issues described in the previous section are the main contributors for program errors or bugs. These issues may become critical in places where navigation light devices are mainly used, mostly on buoys and also in applications which are the main targets of the published works. Theoretically, it is possible to avoid large number of bugs by using simulations and static code checking. Although both significantly reduce overall amount of bugs, they are not capable of detecting all of them. It is also difficult to develop one unified testing method for all embedded systems, as the same scale embedded systems may be used in very different places and have different tasks and interfaces. Another problematic area is the peculiarities of writing embedded software code. Initially, small embedded systems were replacements for complicated digital logic and traditionally, electronic engineers wrote the embedded software but in most cases they did not have enough knowledge to write and test larger scale software [28]. Unfortunately, this also applies to quite big number of authors who write about embedded software and, therefore, there is very little literature available about larger-scale embedded software projects.

The thesis describes problems that have raised while developing a new Telematics Module for AtoN (marine Aid to Navigation) systems. It was not

possible to resolve the problems that are described in this thesis by the methods that are used in non-embedded software development. The following six peculiarities and limitations can be considered as the main contributing cause of the complexity of TM embedded software writing and testing:

- Large number of embedded systems use battery as the main energy source, e.g., most marine AtoN systems and therefore, it is essential that the energy consumption of the device is as small as possible. The main problem with such low-power and constrained systems, is that programs which are used in these systems can use only relatively simple algorithms, also available memory size is limited and in several cases system responsiveness is limited as well. Due to this, all program parts should be rather simple, and all complicated data processing should be done externally on a more powerful computer such as AtoN monitoring server or on other dedicated computer.
- The main programming languages for embedded software are C and C++, but initially both were developed for much more powerful computers (especially C++), and have such language constructs or contain libraries that are too resource consuming for small embedded systems, such as AtoN systems. Unit tests also depend on programming languages; some languages allow to write unit tests more easily. The main problem is to find such program structures, which allows to write more easily larger programs, without consuming significant amount of microcontroller's resources and allowing to use automated testing.
- One of the major problems in embedded software development is software testing; nearly all embedded programs interact directly with hardware, but unit tests, which reduce significantly overall testing efforts, need to run on different hardware, hence needed to emulate or mock target hardware [33]. Another issue with automated tests is that these tests require that programs and functions have certain ending or exit points. Large number of embedded programs are created as super-loop programs, which are inherently endless programs, and it makes it difficult to write unit tests for these programs. Normally, in this case, the test runs forever. In embedded software development large extent OCD is used for debugging, but OCD might have great impact for software real-time behaviour. The AtoN devices, which are the main target of this thesis, have quite high complexity and it is not feasible to use only OCD. The main problem is to find or create such program structures that take small amount of microcontroller resource and allows to create fixtures for automated testing.
- In embedded systems watchdog is used nearly in every program. It is trivial to use watchdog in super-loop programs. However, in multithreaded programs where watchdog should monitor several threads simultaneously, it must also take into account states of all

threads and this is not achievable by using common practices. Therefore the main problem with watchdogs timers, is that no universal method for using watchdog timers with multitasking programs exists.

- Differences between device registers widths and endianness. The embedded systems control and configurations software runs mostly on 32- or 64-bit computers and these computers may also have different endianness. Therefore the main problem is to effectively convert data between different endianness and different register width. In general, these conversions are rather simple, but so far no compiler can do it efficiently enough; these functions are subject to manual optimisation. Similar issues arise with cryptographical functions.
- In larger systems, such as Linux computers, kernel with special hardware is responsible for avoiding memory fragmentation. But in some rare cases, and depending on the application, it is possible that without memory fragmentation protection, embedded system may exhaust free memory. Therefore, the main problem is to develop a mechanism that reduce memory fragmentation as much as possible and at the same time to be suitable for use in smaller embedded systems.

Due to the above listed peculiarities, software development for embedded systems is significantly different than for desktop computers. Currently there is no known specific recommendations or other work for this field, especially for low power AtoN systems.

1.3. Contributions of the Thesis

The main contributions of this thesis are the methods, improvements and solutions suitable for development of new generation low-power AtoN systems, mainly the Telematics Module, which is important component in the navigation light systems that is used on Estonian costal areas. Developed module has also low power consumption, which consequently limit memory size and computational power, but on the other hand provide an capability for long-term autonomous work. To create firmware for such module, software development methods that are significantly different from methods that are used for regular software development have to be used. Software that is based on developed methods is reused also in other devices, mainly in new generation flashers. Additionally, the methods that are presented in this thesis allow to add different functions to TM, like wave height measuring [69], buoy heel angle calculation [67] and collision detection [68] or when server supports measuring vibration in fixed navigational structures.

In order to achieve the above described design goals, several new techniques had to be researched and developed. The main contribution of the thesis are as follows:

- Discussion on how to use effectively program structures that are well known from C++ (however, not very memory and CPU efficient) but absent in C language. The main focus in presented examples are on smaller microcontrollers, which have separated data and program memories. The presented methods can lead to significant memory savings, if wisely used. As demonstrated in development of the TM module.
- Testing methods suitable for software testing in low power embedded systems. Embedded software for smaller systems is often developed without any automated tests. Only on larger embedded systems and desktop computers such tests are used. This thesis proposes program structures and functions, which consume small amount of processor and memory resource, and allows to use automated tests. Also workarounds for code that are not automatically testable are described. Described methods simplify use of CI (continuous integration) servers for testing and regression detection. Presented solution also allows to use lightweight unit tests to test hardware. It is also possible to use such lightweight hardware test programs with FOTA (Firmware Over-The-Air) [65] in order to remotely test deployed TM hardware.
- Effective methods for resetting watchdog in non trivial multithreaded programs. It is possible to reset watchdog timer only by one thread or process. However, in multi-threaded programs watchdog has to be shared between tasks and when at least one thread locks, it causes watchdog to reset. The current thesis describes two different approaches that can be used with multithreaded programs. Pros and cons about different schedulers for multithreaded programs that utilise watchdog timer are also given. Presented methods allows effectively use watchdog in multithreaded programs, which has least one long running task. This method was used for wave height measurements [69].
- Code optimisation methods for deeply embedded systems. Although, most optimisation is carried out by the compiler, still some functions which are quite often used for simple data manipulation, are not optimised even by latest compilers. In this thesis several ways of how to optimise simple data manipulation routines are shown. Also two optimisation algorithms for AES (Advanced Encryption Standard) cryptographical functions, which give significant memory savings or increase processing speed, are described. It is possible to use the described solutions in such cases when minimal processor or memory resource consumption is required. At developing the TM, described methods were used for buoy heel angle [67] and buoy collision detection [68] and it also gave opportunity to encrypt communication channel.

- Alternative approach for standard dynamic memory handling routines. This solution uses memory pool and allows to check memory overruns (writing beyond the end of an allocated block). Although this improvement is not related to any publication included to this thesis, it is used at developments of the Telematics Module.

1.4. Organisation of the Thesis

This thesis is organised into 5 chapters. The proposed methods and developed systems are described and in the publications attached to the thesis.

An introductory Chapter 2 gives a brief overview of microcontrollers and their history and a short overview of programming languages and other development tools, which are used in embedded systems design.

Chapter 3 gives an overview about the main usage area of described improvements and how the methods have been used in development of low power embedded AtoN systems. The chapter also gives a small historical overview about AtoN systems, which are used in Estonia, and development of the AtoN systems; it gives a brief overview of suitable standards and coding guidelines, which were used at development, and reasons and implementations of presented improvements.

The main part of this thesis is Chapter 4 that describes the improvements in development methodologies of embedded systems, discusses suitable programming languages, highlights different program structures, discusses about software testing and usage of dynamic memory, and gives a small optimisation about substitution table (*S-Box*) calculation in AES cryptographical algorithm.

In Chapter 5, concluding remarks and summary of the thesis are presented.

2. BACKGROUND

The central subjects of this thesis, embedded systems and microcontrollers, which are used in AtoN devices, are very different from regular computers. As the AtoN systems have grown steadily into much more complex systems, simple logic or analog circuits sufficient to control and monitor these devices. To control such complicated systems, most appropriate is to use programmable devices, mostly microcontrollers. As the AtoN systems usually operate in remote places, where minimal power consumption is essential, are also used microcontrollers that have minimal power consumption and therefore have quite simple architecture and low performance.

This chapter provides a definition of embedded systems and an overview of the most widespread microcontroller families, their history and development tools. While microcontrollers that are used in AtoN systems have also significantly different development tools, and use different programming language constructs, then background information of the microcontrollers, programming languages and development is provided as well.

2.1. Embedded Systems

An embedded system is a computer system designed to perform dedicated tasks [37]. In many cases embedded system is a part of a larger system, which is intended to communicate directly with other computers or other embedded systems [40]. Many embedded systems have also real-time computing constraints. Physically, embedded systems range from small and power efficient portable devices like watches, to large stationary installations like factory controllers. The complexity of embedded systems varies from very low – with a single microcontroller chip – to high – with multiple computational units. The common denominator of embedded systems is presence of processing units that are either microcontrollers, DSPs or general-purpose processors. Typical embedded system functions as a standalone system with long-term operation.

Current thesis focuses on standalone embedded systems, which have one low power microcontroller for several concurrent tasks. All specific tasks that may require a lot of resources, like network communication, have a dedicated MCU (microcontroller unit); this task partitioning is quite widely spread in low power systems such as car alarms, telematics systems and also in marine navigation light systems.

2.2. Microcontrollers

Unlike general processors, microcontrollers contain most of the required hardware in one IC (Integrated Circuit). Microcontroller has at least a CPU (Central-Processing-Unit) that includes ALU (Arithmetic and Logic Unit), a

control unit, registers, program-memory (generally a flash memory), RAM (Random-Access Memory), I/O devices and a clock system (Figure 2.1).

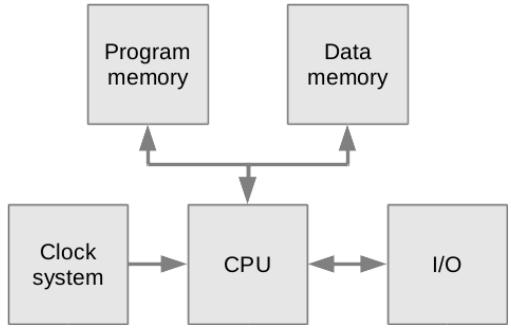

Figure 2.1: Minimal MCU

Two different instruction sets exist, which are used in microcontrollers. While the earlier microcontrollers tend to have a CISC (Complex Instruction Set Computer) instruction set, the recent microcontrollers have more likely a RISC (Reduced Instruction Set Computer) instruction set. Microcontrollers with CISC instruction set have usually more sophisticated architecture; it has microprograms that allow to execute several base instructions in one instruction. Due to the more intuitive instructions and to the microprograms, it is much easier for the programmers to work with CISCs, and the executable programs tend to be smaller. While the assembly language being the main programming language in the 1970s, the CISCs had a clear advantage as majority of the programs were written in this language. Contrary, the RISC instruction set processor has much simpler architecture, mainly because of the lack of microprograms and control unit being less complex. For programmers, programs for RISC instructions tend to be more complex and 30% larger [48]. However, in many cases these programs are a little bit faster than similar programs for CISC instruction set. The downside of the RISC processors is the program length; as most processors execute programs from RAM, and for executing program that has the same functionality as program for CISC processor it requires more RAM and the same also applies to cache. The RISC instruction set is quite complicated and therefore it makes it difficult to directly write assembler programs. However, the instruction set allows to write efficient compilers, and most RISC instruction set microcontrollers have relatively good compiler support. The main advantage of the RISC processors is simplicity; these processors contain less transistors and take less silicon die area, which makes its power consumption lower than in similar size CISC processors, and thus makes this instruction set more appropriate for microcontrollers [13, 39].

In addition to different instruction sets, microcontrollers may also differ by architecture. There are two different architectures available – Harvard architecture, (Figure 2.2) and von Neumann architecture (Figure 2.3).

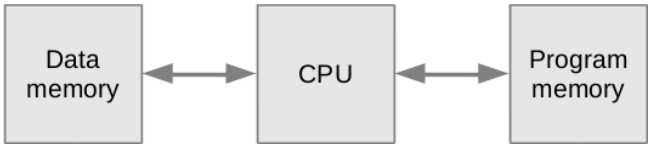

Figure 2.2: Harvard architecture in programmer's view

Harvard architecture has separate data and instruction busses, and memories, allowing transfers to be performed simultaneously from both busses. This architecture does not allow to use simple unified memory. In fact, some processors and microcontrollers that are Harvard machines by the most rigorous definition, may have operations to read and/or write program memory as data. For example, AVR microcontrollers by Atmel have load-program-memory (LPM instruction) and store-program-memory (SPM instruction) instructions. Having separate address spaces create certain difficulties for high-level language compiler and library developers, as most compilers do not support the notion that read-only data might be in a different address space from normal writeable data and thus need to be read using different instructions.

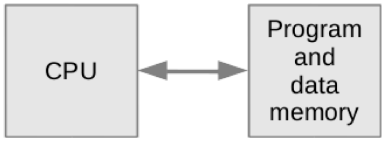

Figure 2.3: von Neumann architecture in programmer's view

On the other hand, von Neumann architecture has only one bus that is used for both data transfers and instruction fetches. Therefore, on von Neumann architecture data transfers and instruction fetches must be scheduled; they cannot be performed at the same time. The von Neumann architecture is a design model that keeps program and data in same memory and is accessible by using the same instructions. It has inability to operate simultaneously on both busses. This inability may slow down microcontroller overall program execution speed, but it can be compensated by using different caches for data and program. For this architecture it is relatively simple to make self-modifying programs or bootloaders[1].

At the present time, the majority of smaller microcontrollers have the Harvard architecture, for example, all the AVR and PIC microcontrollers. The only exception is the Texas Instruments MSP430, which has the von Neumann architecture [93]. However, larger ARM based microcontrollers have a modified Harvard architecture, which in programmer's view is very similar to the von Neumann architecture. In programmer's view, the major difference between both architectures is that the Harvard architecture microcontrollers have

[1] This is used widely in larger computers where main program is loaded from disk to RAM.

separated program and data memories, and need different instructions for accessing these memories. As most of the modern compilers and languages are initially designed for the von Neumann architecture, the compilers usually do not support access to both Harvard memories. Program parts which need access to program memory should use separate functions, which are usually not related to the compiler.

2.2.1. Brief History of Different Microcontrollers Families

The rapid development of microcontrollers began in the early 1970s with the increased IC integration and development of first 4-bit microcontrollers. Due to the high degree of integration of microchips it was natural that besides the microprocessor also additional hardware was added into single crystal. First, the RAM was added, then timers and input-output ports, and eventually other peripherals.

In the early 1990's, microcontrollers with electrically erasable memories such as flash and EEPROM (Electrically Erasable Programmable Read-Only Memory) became available. These microcontrollers could be erased and programmed by using only the electrical signals. Prior to the electrically erasable and programmable memories, microcontrollers often required specialised erasing and programming hardware – typically ultraviolet light (UV) source for erasing and higher voltage for programming. Therefore most microcontrollers before the 1990's had two different variants – one had an UV erasable EPROM (Erasable Programmable Read-Only Memory) for program memory, which had a transparent quartz window on the top of the IC package, and the other was PROM (Programmable Read-Only Memory) variant. These microcontrollers were one-time programmable (OTP) microcontrollers. Technically, however, both were the same microcontrollers. It was possible to reprogram UV erasable microcontrollers twenty to forty times. Due to the IC packages, UV erasable microcontrollers were much more expensive than their traditional OTP versions. Microcontrollers with EPROM and OTP program-memories are not being produced any longer, mostly Flash and FRAM (Ferroelectric RAM) versions are in production.

4 – bit Microcontrollers: The first microcontroller was developed by Texas Instruments in 1971: TMS1000 [92]. This microcontroller went to production three years later, in 1974. Unlike Intel 4004 microprocessor TMS1000 has all supportive parts in the same silicon die such as RAM, ROM (Read-Only Memory), counters, timers and I/O interfaces. This microcontroller had very simple design, it had only two 4-bit general registers, 1-level deep stack, no interrupts. TMS1000/TMS1200 had 43 instructions and TMS1100/TMS1300 54 instructions. Initially, this microcontroller family had only 6 different microcontrollers – 28 pin TMS1000, 40 pin TMS1200, TMS1070 and TMS1270 microcontrollers, which had direct interface for high voltage displays, and TMS1100 and TMS1300, which had twice more RAM.

Besides Texas Instruments, other different manufactures have also produced 4-bit microcontrollers for over 30 years: National – COP400, NEC – μPD75xx, OKI – MSM84xx, Fujitsu – MB884xx, Panasonic – MN14xx/MN15xx, Toshiba – TMP/TCP 43xx/46xx/47xx/47Pxxx, Hitachi – HD/HMCS 4x/4xxxx and Atmel – MACH4.

8 – bit Microcontrollers: In this subsection are listed some 8-bit microcontroller families, which are widely used or is substantially influenced the development of microcontrollers.

F8 microprocessor was the predecessor to 8-bit microcontrollers. This microprocessor was developed in 1975 by Fairchild and it required at least one external microchip for program storage (F3851 or F3856). In 1977, Mostek released an MK3870 [70] microcontroller, which was an F8 microcontroller with integrated memory.

The first widely spread 8-bit microcontroller family was MCS-48 (8048 microcontrollers), it was developed and released by Intel in 1976. This CISC microcontroller had 96 instructions, 1 kB of program memory, 64 bytes of RAM, 8-bit timer and 3 I/O ports [46]. MCS-48 was quite widely used in desktop computers for supportive tasks, for example, IBM used it in the PC keyboard controller [43]; modern computers have integrated the same chip into super I/O device.

Another well-known microcontroller, PIC1650, was developed in 1977 by General Instrument Corporation. This simple RISC microcontroller had 56 instructions, 32 8-bit registers, 512x12-bit program ROM, four I/O ports and internal clock generator [30]. In 1993, Microchip (spin off from General Instrument Corporation) introduced PIC16C84. This microcontroller had on-chip EEPROM for program-memory.

In 1981, Intel introduced new Harvard architecture microcontroller MCS-51, commonly referred as 8051. This new microcontroller differed significantly from its predecessor 8048. It has different architecture and instructions. This MCU has 111 base instructions, 6-source/5-vector interrupt structure, 128 bytes of RAM, 4 kB of ROM, dual 16-bit address bus, four 8-bit bi-directional I/O ports, one full duplex serial port, two 16-bit counter/timers and a on chip oscillator [47]. While Intel no longer manufactures the MCS-51, binary compatible derivatives are still produced from various manufactures. In addition, several companies offer MCS-51 derivatives as IP (Semiconductor Intellectual Property) cores for the use in FPGA (Field-Programmable Gate Array) or in ASIC (Application-Specific Integrated Circuit) designs.

In the early 1970s, Motorola (now Freescale Semiconductor) started a project that in 1975 developed their first microprocessor, the MC6800, which was a base for all MC68XX/MC68HCXX microcontrollers [73]. The MC6800 was a CISC microprocessor with the von Neumann architecture [48]. This microprocessor has a 16-bit address bus, which could directly access 64 kB memory, and an 8-bit bi-directional data bus. It has 72 variable length

instructions with seven addressing modes for a total of 197 instructions. It has four interrupt vectors – restart vector, separate no-maskable interrupt (NMI), software interrupt and hardware interrupts. In 1979, a MC6800 based 8-bit microcontroller MC6801 [75] and MC6805 [74] were developed. Both had on-chip RAM, ROM and I/O on a single die. The MC68HCXX was a successors to MC68XX microcontrollers, this family has several improvements like lower power consumption, higher performance (MC68HC11) and additional hardware (MC68HC08 [26, 77]). The MC68XX and MC68HCXX series microcontrollers were popular in automotive applications.

Atmel developed its first 8-bit microcontroller in 1996. This microcontroller was AT90S1200; a 8-bit RISC microcontroller, which has slightly modified Harvard architecture. It has 89 instructions, 32 general purpose registers, 1 kB of program memory, 64 bytes of EEPROM, one timer, analogue comparator, on-chip oscillator and 15 programmable I/O lines [2]. Unfortunately, the first Atmel microcontroller did not have any SRAM (Static Random-Access Memory), and it had very limited C compiler support[2]. Soon after the release of the AT90S1200, a series of different microcontrollers were also released – AT90S2313, AT90S2323, AT90S2343, AT90S4414, AT90S4434, AT90S8515 and AT90S8535; all of them had SRAM that allows to call virtually unlimited number of sub functions. Among the other AVR microcontrollers the AT90S8515 [1] was produced, which was intended to replace 8051 microcontroller. This microcontrollers has a 40-pin DIP (Dual In-line Package) package with the same pinout as the 8051 microcontrollers, including the external multiplexed address and data bus. The only difference was the reset line polarity. In 2008, Atmel released family of new 8/16-bit AVR XMEGA microcontrollers. These microcontrollers had more memory, DMA (Direct Memory Access) controllers, event system, cryptographical engine and high speed AD and DA (Digital-to-Analogue) converter, and also some 16-bit instructions. All experiments that have been made in the context of this thesis are carried out on the 8-bit AVR microcontrollers.

16 – bit microcontrollers: The 16-bit microcontrollers are not so widely used as the 8 and 32-bit microcontrollers, only some 16-bit microcontrollers have been spread more widely.

In 1982, Intel released its first MC-96 family of microcontrollers that were widely used in car industry. Another well-known 16-bit microcontroller family is the Texas Instruments MSP430 [93]. This RISC microcontrollers have the von Neumann architecture and are designed as measurement controllers and work on batteries. Besides Intel and Texas Instruments 16-bit microcontroller families, several other 16-bit microcontrollers families exist: STMicroelectronics ST10 families, Infineon (former Siemens) C166 family microcontrollers and Freescale HC12 [25] and HC16 [76].

[2] Subroutines use stack to pass parameters and return addresses, therefore, it is quite complicated to call any subroutine without using RAM.

32 – bit Microcontrollers: The best known 32-bit microcontrollers are the ARM architecture based RISC microcontrollers. This architecture was first developed in the mid 1980's for personal computers. The ARM uses quite simple instruction set and therefore these processors have relatively low transistor count and quite low power dissipation; this makes ARM architecture well suited in power constrained devices. As of 2016, in terms of quantity, ARM architecture microprocessors are globally the most widely produced 32-bit instruction set architecture. The first ARM processor was produced in 1985, and ever since the ARM has released many different 32-bit processors, from ARM1 to ARM11. In 2004, ARM launched Cortex-M3 core processors. This Cortex-M family was intended to replace 8- and 16-bit microcontrollers but these are still not so widely spread as 8- and 16-bit microcontrollers. In 2005, ARM launched Cortex-A series microprocessors, which were intended to be used in high performance applications such as tablets and mobile phones. As ARM Holdings itself does not produce processors, it licenses the processor architecture to chip manufactures. Many microcontroller manufactures have some ARM versions in their product portfolio.

Atmel developed its 32-bit microcontrollers in 2006, the AVR32 microcontroller family. This microcontroller family has completely different architecture than the 8-bit AVR microcontrollers. The AVR32 architecture consists of two different micro-architectures: the AVR32A and AVR32B. Both of the microarchitectures provide different performance, have different registers, peripherals, instruction set, and different power consumption [5]. The AVR32A microarchitecture targets cost-sensitive, lower-end applications. All AVR32UC microcontrollers have this microarchitecture. The AVR32A microarchitecture saves chip area at the expense of slower interrupt handling. AVR32B, on the other hand, targets applications where more processing power is needed like ethernet switches. AVR32B microcontrollers had mostly the same functionality and application areas as ARM microcontrollers, however, starting from 2013, the whole microcontroller family of AVR32B is not produced any longer.

Microchip introduced 32-bit microcontroller family in the end of 2007 – PIC32MX microcontrollers. The initial device line-up is based on the MIPS32 M4K core [62]. The PIC32MX family is pin-compatible with most of the 16-bit Microchip PIC24/dsPIC microcontrollers. This microcontroller family has quite similar functionality as the ARM microcontrollers, and therefore PIC32 microcontrollers are not very widely spread.

64 – bit Microcontrollers: Unlike the 64-bit processors, only a few 64-bit microcontrollers have been developed. The main argument against the 64-bit microcontrollers is the high power-consumption. In 2011, ARM Holdings announced the release of new 64-bit architecture [34] processor's family: the ARMv8. This family has two different processors: Cortex-A53 and Cortex-A57. Both are targeted to tablets, smartphones and other mobile devices.

Toshiba also developed MIPS based 64-bit microcontroller TX4927 [95] in 2001. This 64-bit microcontroller has 200 MHz clock and PCI (Peripheral Component Interconnect) interface, SDRAM (Synchronous Dynamic Random Access Memory) memory controller, DMA controller, interrupt controller with 18 sources, 2 channel UART (Universal Asynchronous Receiver/Transmitter), 3 channel 32-bit timer/counter, and 16-bit bi-directional I/O ports. TX4927 has relatively low power requirements, it operates at 200 MHz and consumes only 1.5 W [96]. It has the same performance as typical desktop computer.

A brief overview about different microcontroller families was given in the above sections. While this thesis is focused to embedded software, are also outlined most significant properties from the programmers point of view. In programmer's view, the most important properties of microcontrollers are as follows:

1. Memory protection – the presence or absence of memory protection unit determines the complexity of a program development.
2. The size of the memory – mostly, the size of the RAM sets the upper limit for the size and complexity of a program.
3. Registers – having more CPU registers allows to write more efficient program.
4. CPU clock – for non signal processing or time critical application, in most cases, CPU frequency does not play significant role. However, since lower clock frequency gives significant power saving, it is used in several embedded systems.
5. CPU endianness – it plays role when embedded system need to communicate with other systems.

2.3. Programming Languages, Debugging and Development Tools

The following section gives a short overview about programming languages, supporting programs, debugging tools, hardware for program memory uploading and hardware for simplification of embedded software development.

2.3.1. Programming Languages in Embedded Systems

This section gives a general overview about programming languages, which are used in embedded software development, together with some programming languages that had importance in history.

Machine Code

It was quite natural that in the first microcomputers a machine code was used for programming [79]. Since machine code programs can be written without using computers, this programming method was also used in the very beginning of a computer era when there were no computers to write programs. The major shortcoming of machine code programming was that the program code had to

be entered by hand to a program memory or to a device that held a program, which was time consuming and prone to errors. In 1970's when rapid development of microcontrollers began, relatively powerful computers were available, which allowed to use translators or at least had possibilities to write them. At the present time it is not known that machine code is being used. It is used only for teaching purposes.

Assembly Language

Assembler was the next step from machine code to higher level programming languages [38]. The creation of assembler language was greatly motivated by computers that were able to run translators and larger and more complex programs.

Unlike the higher level languages, one assembler instruction is also one machine instruction and translator does not change the order of instructions. Some translators are able to use preprocessors, it makes possible to use macros such as GCC (C compiler from GNU Compiler Collection) uses. Assembly language allows to translate symbolic memory addresses into relative or absolute addresses. For example, Listing 2.1 presents two instruction infinite loop, which always jumps one instruction backwards. In this example translator changes addresses L1 and L2 into real memory addresses, which may be 0x100 for L1 and 0x101 for L2.

```
1:L1: nop       ; no operation
2:L2: jmp L1    ; jump back to nop instruction
```

Listing 2.1: Example of symbolic addresses.

Depending on the *jmp* instruction and architecture, the parameter L1 may be symbolic or absolute address.

Possibility to create functions that are not feasible in higher level languages is the main advantage of the assembly language. For example, when it is needed to take maximum performance from a computer, when the compiler does not support some specific instructions or it needs to create extremely small programs. In embedded systems use of assembly language several places is not uncommon. For example, functions that access the AVR program memory use special program memory read instructions, which are available only in assembler. Assembler can also be used when it is needed to call no-operation *nop*[3] instruction.

Another advantage of assembly language is the possibility to access directly the registers, which in higher level languages is more complicated if possible at all. Such flexibility gives to a programmer more control over the hardware.

There is a common misconception that assembler programs are always faster than programs in higher level languages but this is true only for smaller

[3] Higher level language usually does not have *nop* instruction, this instruction is most likely required for delay loops.

programs. Lot of programs that are written in C have similar performance as similar assembly programs. The argument whether the programs in higher level language are faster or smaller is true with non-trivial programs – a program should have at least 500 to 1000 effective lines of code in C, and the compiler should also use maximum optimisation, and programming language should allow quite low level hardware access. If all above mentioned conditions are met, the C program can be as fast as the same program in an assembly language. This is mainly due to the fact that code reuse and optimisation in a large assembly language program is more complex, and it is much harder to use microcontroller registers as effectively as higher level languages do; this is mainly limited due to human capabilities. Of course, this argument can be relatively easy to refute but doing so would take a lot of time even for an experienced programmer.

Due to the availability of higher level programming languages the only argument for assembly language would be a need to use instructions that are not supported by a compiler or to create small and extremely fast programs. However, it is still valuable to know assembly language at some level; debugging of embedded software is not possible without knowing it.

C, Ada and Other Procedural Higher Level Languages

Higher level languages (procedural languages) were introduced mainly for achieving the following goals: to speed up the programming process, to reduce the coding errors, and to get more readable programs. Programs in procedural languages can be as fast as programs that are written in assembler. However, developing, debugging and porting are much easier. Some higher level languages, such as C, have several features that can have side effects or are implementation defined. Therefore, it is relatively easy to make coding errors, which are difficult to find, for example, pointer related bugs. Furthermore, program speed, size and memory footprint are highly dependent on compiler and optimisation level. Debugging from disassembled code may be difficult: compilers may eliminate some portions of code, which do not have any visible effect, like badly written delay loops.

The most commonly used programming language for embedded systems is C. This language was is created by Dennis Ritchie between 1969 and 1973 at AT&T Bell Laboratories [35]. One of the first uses of this language was to rewrite the UNIX operating system, which had previously been written in assembly language. C language is quite different from other languages. Unlike many higher level languages it has quite low level access to hardware, but it is not dependent on underlying hardware, like assembly. The C language design provides constructs that map efficiently machine instructions to higher level languages and therefore the language is used in several different applications that were formerly coded in assembly. It makes C relatively easy to use in embedded systems, but the downside is that it is quite complicated to create larger programs, which are not directly related to hardware. However, the C language allows to use such constructs that have undefined or implementation

defined behaviour and it is also possible to use code constructs that are hardly understandable [35]. Therefore it is difficult, but not impossible, to use the C programming language in safety critical programs [45]. The main advantage of C languages is that almost every platform has a C compiler and most microcontroller vendors have tools and supporting documentation.

For embedded software development, safer programming languages, like Ada, are used. Ada was originally designed by a team led by Jean Ichbiah of CII Honeywell Bull under the contract of the United States Department of Defence (DoD) from 1977 to 1983 in order to supersede many programming languages used by the DoD. It had built-in language support for explicit concurrency, offering tasks, synchronous message passing and protected objects. This language was originally meant for embedded and real-time systems. Ada is not so widely spread in 8-bit microcontrollers, mostly because of popularity of C and compiler support limitations. The first Ada port for AVR GCC and its runtime was released in the mid 2000's. Due to the small number of users who use Ada programming language on AVR microcontrollers, the development of Ada compiler was quite slow and currently many features are still missing. It is quite likely that many features will never be implemented for AVR or similar microcontrollers. The main reason for missed features is the small memory and low computational power. Therefore it is not reasonable to use Ada's GCC port on 8-bit microcontrollers; and using Ada in the context of this thesis was not even considered.

For smaller, mainly for the 8-bit microcontrollers, the C programming language is one of the most frequently used languages; and in some extent the assembler is used. All the other languages, except C++, can be considered as experimental or too resource demanding.

C++, Ada and Other Translated Object Oriented Languages

Increasing microcontroller performance allows to create more complex programs, that consequently leads to a need of object-oriented languages. The most commonly used object-oriented programming language for microcontrollers is C++; 20% of projects use it [86]. The second widely used object-oriented language is Ada, which falls within the scope of a several percent. In larger systems with more powerful microcontrollers, object-oriented languages are used in much greater extent. Below the functionality of the C++ and Ada, and the usage of those languages in embedded systems are briefly discussed.

C++ was developed between 1979 and 1983 by Bjarne Stroustrup as an object oriented improvement to C [89]. While C++ is an object-oriented improvement to C and contains quite resource demanding functions, it is still possible to write relatively complex programs that are as fast as C programs, even by using inheritance. C++ has two features that may cause problems when using these in embedded systems: dynamic memory allocation and virtual functions. When using frequent dynamic memory allocation and deallocation,

the *new* and *delete* operators, it is possible that the memory may fragment, which consequently introduces quite difficultly detectable bugs, and therefore using dynamic memory in embedded systems is not recommended [72]. The second possible source of problems are virtual functions. These functions can be overridden during program execution. For calling these functions, indirect calls are used that read function addresses from a table in RAM. As microcontrollers have limited amount of memory and virtual function table, which is placed into RAM, makes using this feature in embedded systems problematic. This shortcoming may not be actual when using newer compilers as it is able to devirtualize functions. The good side of C++ is that this language has lot of supporting tools like many different UML (Unified Modelling Language) tools and unit testing frameworks, which ease the programming significantly.

In embedded systems it is possible to use Ada alongside the C++. The major downside of Ada is that this language is used in very limited areas and only few compilers support it. The AVR GCC has also unofficial Ada port [23] but this has quite limited functionality.

Using object-oriented languages in smaller microcontrollers may not give significant benefit. The main advantage is in the complex programs, especially where it is necessary to model some external process or use unit tests. It is necessary to keep an eye on virtual functions and other resource demanding functions.

Java and other Interpreted Object Oriented Languages

The interpreted languages have quite big advantage over translated languages. One program is able to run without re-compilation on different architectures. All interpreted languages have an interpreter as a middle layer between program and hardware, and underlying OS. This layer is responsible for program execution. Besides universal program, the interpreter gives one layer security between underlying OS and programs; program errors are caught by interpreter. It also allows to create and test program in one architecture and run on other. Most widely known interpreted languages are Java and Python and both are used on larger embedded systems.

Java was developed by James Gosling at Sun Microsystems and first released in 1995. Unlike many other interpreted languages, Java programs are compiled to architecture independent Java bytecode and executed by Java virtual machine (*jvm*). The main Java drawbacks are that this language requires quite powerful processor to run and initially it was not intended to be used in real-time systems. As Java is not designed to perform real-time tasks it has several functions that have unpredictable timings. The main reason of unpredictable timings is the stochastic delays, which are caused by garbage collector. The garbage collector is responsible for unused memory management; it might start its tasks at an unpredictable time [87]. Some real-time virtual machines also exist that do not have such drawbacks [32, 44], but these virtual machines are not very widely used. In soft real-time and non-real-time

embedded systems, Java is used in Javelin Stamp microcontrollers and on some Lego NXT bricks, and also in Android OS.

Another well-known interpreted language is Python [83], which was developed by Guido van Rossum in 1991. Unlike Java, the Python has a weak type system; the declared data or variables do not have distinct type. Due to the weak type system this language is not best suitable for embedded systems. Weak typing system might create hard-to-detect bugs, for example, it might change numerical variable to string variable. The compiler is not able to determine type conflicts during compilation and therefore it is possible bugs, which are caused by invalid conversion, surface during an exploitation phase; probably the most well known type conversion error was an Arianne 5 accident [54], although it was not related to weak types. The only known devices where Python is used are Telit GSM/GPRS/3G (Global System for Mobile Communication/General Packet Radio Service/3rd generation of mobile telecommunications technology) modems, like GM862 [91].

Both interpreted languages have similar independence of the underlying platform. However, if the program was developed on a different architecture, it might be necessary to carry out additional hardware related testing on a target hardware. Programs in both languages are not well suitable for hardware control; programs in Java and Python need some intermediate layer which has access to hardware. The layer makes hardware access resource consuming.

Other Programming Languages

In addition to the above listed programming languages, there are very few alternative programming languages for embedded systems. For Programmable Logic Controllers (PLC), ladder diagrams are used most often, which mostly describe relay logic and therefore are not suitable for generic programs. Another alternative for embedded devices is LabVIEW. This is also not very common among programmers, mostly because of the high price and it uses graphical dataflow programming language “G” instead of regular text based programming languages.

2.3.2. Supportive Programs

In order to create executable programs, linkers are needed. These programs take all compiler generated object files, find missing functions from libraries and put them together into one executable file. When the linker creates final executable file, the compiler usually calls it automatically; in most cases, programmers do not see when linker is called. Even when the user needs to call linker during a final compilation step, it is still mostly called through compiler. The compiler has more information about the target system and it recognises the types of microcontrollers. The linker however needs to know only some of the microcontroller's family information.

For PC programs the linking is the final step in program compilation. Yet microcontrollers need one additional step to generate program image from an

executable image. A memory image loading is needed because microcontrollers do not have such resource or operational system to load executable programs, resolve libraries, and place final executable into the right memory. Therefore it is required that these steps should be carried out before loading a program into microcontroller memory. Memory image creation software is typically distributed within same package with linker. With GNU tools is this program is part of binutils package.

2.3.3. Standard Libraries

Typically, embedded programs use at least one shared library, but unlike desktop computers that use dynamic libraries, only static libraries are used.

Programs, which are written in C, most likely use a C standard library: *libc*. In embedded systems, this library has quite limited functionality; it has few resource consuming functions and many string manipulation functions. Quite often various functions that are related to floating point mathematics or threads are missing. Typically, every smaller (8 or 16-bit) microcontroller family has its own architecture specific *libc*, AVR has *avr-libc* [99], MSP430 has *msp-libc* [103] or *newlib* [104]. Several different *libc* implementations are available For 32-bit microcontrollers, for example, *newlib*, *uClibc* [106], *dietlibc* [100] and *EGLIBC* [102]. *Libc* implementations, which are for smaller microcontrollers, contain usually some additional functions like delay functions, checksum calculations and EEPROM access functions.

In addition to *libc*, some reusable code is included with the compilers. Every compiler version has its own set of functions and with the release of every new compiler, additional library functions seem to be added.

2.3.4. Debugging

Testing and debugging in desktop computers usually takes same effort as coding. In embedded systems, however, testing and debugging can take twice as much time as coding. This difference is mostly due to the limited development tools and target hardware. Embedded systems present special problems for programmers as it usually lacks user interfaces and storage media, which is available in desktop computers. These shortcomings make simulators, emulators and in-circuit software debugging tools essential for many common development tasks. The following section outlines some of the most commonly used debugging tools.

Simulators and Emulators

For debugging smaller embedded programs, it is possible to use special programs and hardware that emulates target microcontroller.

One possible option to imitate microcontroller is to use special simulation software. The simulator uses only software to simulate target hardware and therefore it is not possible to use it for real-time task simulations. Many

simulators are developed by microcontroller manufacturers and are included in their official IDE (Integrated Development Environment), for example, Atmel has the AVR simulator in their *Atmel Studio* (former *AVR Studio*).

Many open source simulators exist: *mspsim* for the MSP430 microcontroller and *SimulAVR* for AVR microcontrollers. Unfortunately simulators are not very widely used. The main reason for this is a relatively complex simulation process: to examine one specific code fragment it might be necessary to input a lot of different input signals in order to reach the desired state. In most cases it is done by setting or clearing some graphical interface input fields. This makes the use of simulators quite ineffective for large programs. It is reasonable to use simulator only when no real target hardware is available or a test program has such input values that are impossible in test environment.

More accurate method for debugging a program is to use an emulator: a device that is connected to PCB (Printed Circuit Board) instead of a microcontroller. Traditionally, emulator has a plug that inserts into the socket where the microcontroller chip would normally be placed. Unlike the simulator, which is pure software, the emulator is a device that imitates very precisely real hardware. Emulators are usually capable of storing full call trace and therefore it is possible to retrieve a command sequence that was executed before an error occurred. An emulator control software is usually integrated into IDE; it is similar to simulator control software.

As emulators are relatively complicated devices, most of them are produced by microcontroller manufacture's and have quite high price. Despite the good properties of the emulators, the recent microcontrollers do not have any supporting emulators. This is most likely caused by the fact that most recent microcontrollers have too high clock frequency or have packaging which contains too many IO lines. This makes it technically difficult to design such emulator that acts like a real hardware. In newer microcontrollers, which do not have emulators, it is possible to use an ICE (in-circuit emulator).

JTAG, ICE and OCD

An ICE (in-circuit emulator) is a hardware device that is used to debug a software of an embedded system by using its onboard microcontroller. This term covers all hardware debuggers, including debuggers that provide access using JTAG (Joint Test Action Group – standard test access port) connection to on-chip debugging hardware on standard production chips.

In most cases, ICE uses JTAG connection, mainly because that JTAG hardware interface uses only four or five electrical signals and it is able to access large amount of microcontroller hardware. In addition to debugging, it is possible to use JTAG as a software uploading tool. Although the JTAG is the most popular connection type, but microcontrollers may have an alternative to JTAG; for example, Atmel uses *debugWIRE* interface, which uses only one wire, and this alternative interface has the same functionality as the JTAG. The ICE, which is connected to MCU, are sometimes called in-circuit debuggers

(ICD), to distinguish the fact that they do not replicate the functionality of the MCU but instead control already existing MCU.

In the context of this thesis the OCD[4] is the whole debugging system that includes ICE as the hardware part, and also has debugger as software part. With the OCD it is possible to control and monitor all microcontroller interfaces, change values in registers, use microcontroller outputs for controlling real hardware and execute program step by step. It is also possible to pause program execution and notify developer in some predefined condition; for example, when a program counter points to some specific memory address. More advanced OCDs [84] allow to watch thread states but architecture used in this thesis does not have such support. Typically IDE's also have quite good support for OCD; for example, many modern IDE's are capable of binding an ICE sent data to a source code, and it is possible to watch program behaviour in higher level language like C, which simplifies debugging significantly.

Other Debugging Methods

In addition to above mentioned debugging tools, two additionally alternative methods exist. One is to use the microcontroller’s serial port for output of program states, and another is to toggle general purpose IO pin when some parameter has changed. Both methods are also described in the Section 4.3.5 – Debugging and Testing.

Sending microcontrollers output states through serial port is the easiest debugging method. When the microcontroller has other interfaces for data outputting, then it is possible to use any other serial interface, like JTAG [7]. In most cases only one way communication from microcontroller to developer's computer is used. This testing method uses some text or binary data outputting command (like *printf* in C), which may take quite long time and may use some kernel functions like interrupt handlers. Data outputting is the main bottleneck of this method and therefore it cannot be used in places that are related to print functions itself, scheduler, interrupt routines and bootloader. Also, it may have a big impact to real-time tasks and most likely change inspected memory function requirements. Most often it is used as ad-hoc debugging tool; print calls are inserted to find and to remove a bug. Should there be no other more effective methods for monitoring program behaviour then this method is the most preferred. This is also used widely in hobby projects.

Second method for monitoring program behaviour is to use one free microcontroller output pin. The main idea of this method is to change pin output state when a microcontroller program's internal state changes, for example, a program executes true branch from if-else sentence. Unlike the other debugging approaches, this approach has very low overhead; it needs very few instructions to complete, and mostly does not influence any real-time tasks. However, this method is usable only for simple programs. Using a proper measuring equipment, it is possible to monitor some time critical functions like measuring

[4] OCD may also refer to software part, which is between debugger and ICE.

some process duration or synchronisation [28]. Due to the limitations of this simple method it is usable only for very simple programs.

2.3.5. Microcontroller Memory Programming

Before using a device, which is fitted with an integrated microcontroller, it needs to have a software programmed into its program memory. For smaller microcontrollers, it can be programmed by using four different methods: SPI (Serial Peripheral Interface Bus), JTAG or similar serial programming, using a bootloader or a special memory chip programmer.

Most of the smaller microcontrollers have at least one interface to access program memory, typically, these are SPI interfaces. But it is not uncommon that some microcontrollers use JTAG interface only, like MSP430. The main benefit in using serial programming is that it needs little hardware: only three to five wires to connect from programming adapter to microcontroller. This method allows to access program memory without needing to physically remove any memory chip. The only drawback is the requirement for special programming software and adapter. This programming method is also known as In-System Programming (ISP).

Another relatively common programming method is using a bootloader. A bootloader is a small program that is programmed into microcontroller special memory section to enable reprogramming microcontroller program memory. In ISP programming mode, the microcontroller acts as an external memory, which is connected to programmer, but when using bootloader, the microcontroller acts like ISP programmer. It receives a program from a communication interface like a serial port, or from an external memory like SD card, and loads it into microcontroller program memory. In most cases the bootloader can use microcontroller’s full functionality. The only limitation is the size of the bootloader's program, which should be some few kilobytes. In some embedded systems the bootloader supports program loading from encrypted images [3, 4]. In order to use a bootloader, one should have write access to microcontroller program memory. Before using a bootloader, it is required that a bootloader is loaded into program memory with methods like SPI or JTAG.

Older microcontrollers, which have external program memory chips, were programmed by using special programmes for memory chips. For programming an external memory chip, a memory chip which contained a program, had to be taken out and then inserted into a programmer. Similar methods are used in larger embedded systems for transferring program from an external memory card like SD card or CompactFlash. Lot of smaller microcontrollers have an internal or external program memory, which is programmable through programming interface and therefore this method is not used widely on microcontrollers.

2.3.6. Development Boards

During embedded system development, some systems have very limited IO functionality or do not have suitable hardware for development process. Some systems do not have required debugging interfaces and some older microcontrollers have only one time programmable memories. To cope with these shortcomings, most of the microcontrollers have development or evaluation boards, which have the interfaces required for debugging and additional memory. The main difference between a development board and an end product is that the development boards are intended to be used in laboratory environment. These boards usually do not have such enclosure that could be used in outdoor environment and many of them do not have a power supply.

Two different types of development boards exist: generic, and for a specific product. The generic development boards have less hardware. Generic boards have typically lots of different onboard IO connections, but have less supportive hardware. The main drawback of this kind of boards is the difficulty to connect external high speed hardware; most of the interfaces are unprotected and are quite sensible to external electrical interferences. These boards are suitable in the beginning of the product development phase for experimenting with some isolated function or just for an engineer or for a student who is interested to be acquainted with the targeted microprocessor and learn how to program it. The best examples are Atmel 8-bit AVR development boards STK500 [9] and STK600 [10]. Both boards support most of the AVR microcontrollers.

Another kind of development boards and modules are product specific boards. These contain most required hardware for specific tasks, and also some additional hardware for debugging and interfaces for experimenting with other electronics. In some cases these can be used as prototyping but usually are not usable outside laboratory environment. The best examples in this category are the Texas Instruments EZ430, which is a wireless development board, and Atmel Butterfly, which is mainly an LCD development board. Both have one microcontroller soldered and the main purpose is to demonstrate one specific function.

In addition to above listed development boards, third party development boards exist. These are intended to be used with some simple products, which do not have very strict environmental requirements, for example, Ethernut board [51].

2.4. Conclusions

The Chapter 2 gave background information for the current thesis. The first part described embedded systems in general: systems that has one single purpose. Different microcontrollers with word lengths of 4, 8, 16, 32 and 64 bits exist. The 4-bit microcontrollers are mostly historical. A 64-bit microcontroller is still quite new and not widely used. Currently the most popular microcontroller is the 32-bit ARM architecture microcontroller, which is used largely in mobile

phones and tablets and as well as other electronics. From the energy consumption view, the 8 and 16-bit microcontrollers have very low power consumption; however, some later 32-bit microcontrollers have similar characteristics. Developments and researches that were made within current thesis are mainly targeting 8-bit microcontrollers.

In embedded systems the most frequently used programming language is C. The next popular programming language is C++ and all other languages are not used so often. The C language was initially intended for rewriting UNIX operational system but it also allows to write quite effectively hardware related programs. The C++ was created as an improvement for C; this language is much more complicated but it allows to write object oriented programs for embedded systems that are as effective as similar programs in C.

Additional programs and hardware such as linkers, debuggers, OCDs and programmers are also required for embedded software development. Linkers are programs that take object code and produce the final executable program. Memory image generation programs are within the same software collection with linker. It is possible to use OCD, emulators and simulators for debugging embedded software. Finally, a program and hardware that is needed in embedded software development is a programmer and its software that allow to upload final program image into microcontroller’s program memory.

3. CASE STUDY

In following sections a brief overview about usage area of developed methods, improvements and publications, which are related to this thesis are given. Primarily, initial development stage and later improvements of new generation AtoN on site Telematics Module (in the following texts are used abbreviation TM), are described. Many problems, which raised during the TM firmware development, were not solvable by using commonly known software development methods. The majority of problems were related to FOTA [65], buoy onboard heel angle calculations [67] and buoy collision detection [68]. Some issues occurred during the development of wave height measurement application [69]. To ease the TM software development a lightweight profiling application [66] was developed. Also, the methods that were used in similar situations on larger computers did not solve the raised problems. Hence, the new methods were required, which were quite specific for TM, but fortunately quite universal in order to use in other similar systems.

3.1. Marine Navigation Light Systems

In this section an overview of microcontroller usage in marine navigation light systems is given. This section gives a description of Telematics Module. All improvements in embedded software development described in Chapter 4 are used for development of this module.

3.1.1. Aid to Navigation and Remote Monitoring Systems

In maritime safety, visual navigation light systems – typically buoys and lighthouses - play an important role. Several different ship based systems that utilise GPS (Global Positioning System) and AIS (Automatic Identification System) are also used. Despite the GPS and AIS based systems, lights based navigation is very useful in places where ship speed is quite high, in some cases even over 35 knots. It is also required that all such AtoN devices have higher reliability characteristics than consumer electronics.

Although AtoN navigation light systems have quite robust hard- and software, it is not very rare that some devices may have failures or damages from the environment. Most failures are caused by component failure and can be repaired by supporting staff. Damages may have several different causes – damages where humans are involved, like ship and buoy collision or damages, damages which are caused by animals or birds, like cormorants or seagulls who can damage electronic equipment, and damages that are caused by natural phenomena, like storms. All of them are handled like failures and require supporting staff intervention. Therefore it is essential that such devices have remote monitoring and control capabilities – AtoN telematics that allows to inform ships and supporting staff about AtoN device state or faulty AtoN

devices. Remote monitoring and control mechanisms also allow supporting staff to retrieve detailed device information, like battery voltage, buoy heel angle, or reconfigure device functionality and even update device firmware.

Also other natural phenomena like wind and waves may have quite big impact to marine safety. Both have similar effect on navigational buoys – they decrease navigation light visibility range. Decrease of visibility range is related to the physical dimensions of light source, it has vertically very narrow beam (mostly because of low energy consumption), and even quite small waves may have noticeable effect to visible distance. In installation sites where sea is icing in winter times, it is possible that ice pushes buoy to very high heel angle, or push it under the ice or drag to another location. Therefore it may be beneficial to inform supporting staff about sea or buoy conditions that are caused by high waves or ice.

In AtoN devices GSM based solutions, like SMS (Short Message Service) messages and GPRS/3G data connections are used for remote monitoring and controlling. Also, some systems that are too far from shore use satellite or radio communication. In receiver side there is a monitoring server, which has typically a database for storing device operational history and have device controlling capabilities. Depending on devices and communication interfaces server may have additional tasks, like collecting measurement data from buoys, calculate wave heights, or update AtoN device firmware. Monitoring server software is typically quite complicated set of different programs, it has several isolated programs – one for communicating to AtoN devices (front end part), another for user interface (back-end part), for database and for complicated calculations (like wave height processing).

3.1.2. Estonian AtoN System

Typical visual AtoN systems consist of light sources like buoys or lighthouses, telematics modules, servers, databases and user interfaces. In Figure 3.1 an AtoN system, which is used in Estonia is described. This system contains also synthetic AIS radio network. In buoy a light source, flasher, GSM/GPRS telematics module, GPS and battery packs are installed. Similar systems are installed to lighthouses, but there instead of a battery pack some other power source, like solar panel, wind generator or connected to power network is used. The main tasks of TM in this system is to monitor other devices, compute distance from installation location to last known location and transfer monitoring and distance information over GSM/GPRS network to server. Server collects all messages from different buoys and lighthouses, stores important information to database and sends buoy operational information to AIS server, which sends buoy info to AIS transmitter network. The server stores also acceleration data and forwards it to wave height calculation submodules, which calculate wave heights and store results to the database.

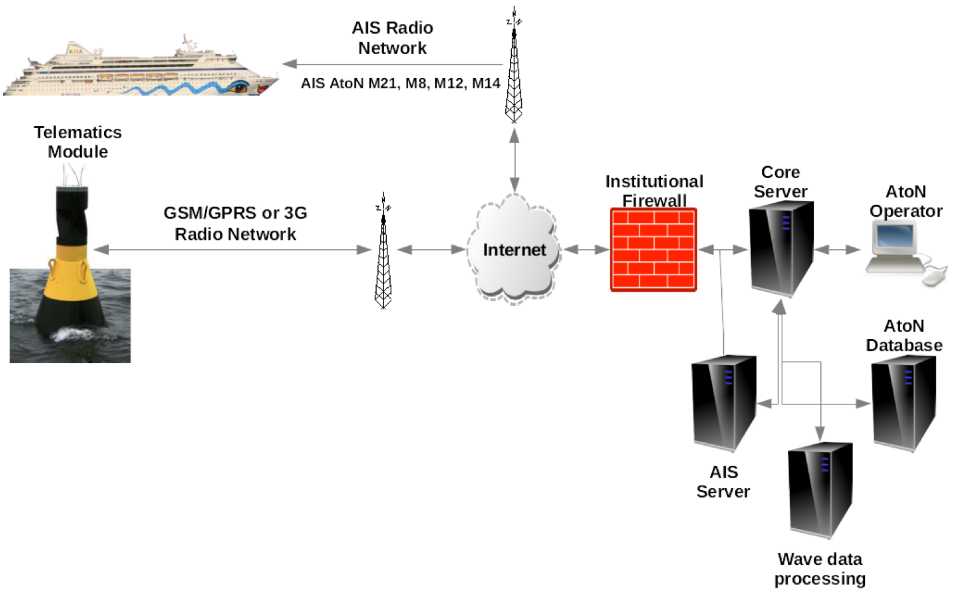

Figure 3.1: Typical synthetic AIS AtoN system setup

Similar AtoN solutions with synthetic AIS are developed by several manufactures, like Sealite [109], but solutions from different manufactures are not compatible with each other.

Also several buoy AtoN electronics manufactures (like Sabik [108], Tideland [112], Pharos Marine [107], Zeni Lite Buoy [113] and SRT Marine Technology [111]) have AIS transmitters, which can be installed on a buoy. Systems that use such transmitter do not need complicated server part, but if something goes wrong, it is much more difficult to analyse what has happened with buoy, and most cases it is required to send maintenance staff to check the buoy. Also such transmitters tend to consume more energy than synthetic AIS buoys.

3.2. Telematics Module

The most complex part in the described AtoN system is TM (Telematics Module). This module is responsible for most communications and synchronisation tasks. As all described improvements in this thesis are related directly with TM, in following section short overview about this module evolvement is given.

3.2.1. History of Telematics Module

Estonian made remote control and monitoring capabilities AtoN telematics was introduced in 1994. First telematics modules had NMT (Nordic Mobile Telephony) modems and in order to report AtoN status to server, it was needed to take data call to central phone number. In 1999 NMT network was closed and

all modems, which were installed to AtoN devices, were replaced by GSM modems, and as this change involved only modems, most of the software remained the same.

Telematics modules with GSM modems were used until 2005. After 2005 telematics modules were updated with GSM/GPRS modems, and 3G modem support was added in 2011. New GPRS modems are able to communicate by using a TCP/IP (Transmission Control Protocol/Internet Protocol) connection instead of using GSM data calls. In new generation TM also all other electronics were replaced. Most notably new microcontrollers were added that had lower energy consumption, more computational power and more additional hardware. Unlike the old microcontrollers, which were programmed only in assembly language, the new microcontrollers had C and C++ compiler support (GCC compiler support), therefore, all telematics module software was rewritten, and also a small kernel was added. Improvements in programming language and kernel made feasible to add several additional functions such as onboard buoy heel angle calculation [67], buoy collision detection [68] and also the FOTA [65] capability. In 2006, a new front-end server was introduced, which was able to communicate over TCP/IP network with new GPRS modems. The new server also supported receiving raw acceleration data from telematics module, which gave it a possibility to calculate wave heights in special server side program. A wave height calculation software [69] was developed in 2010. It is a separate server side program that also exports wave height data to a public web server[5].

To solve the problems that were raised during the new TM firmware development, significant research on embedded software development and testing methods was required. The results of this research are presented in the following chapter.

3.2.2. Architecture of an AtoN System

Embedded systems, which are used in above described AtoN system, are composed of several controllers that are connected to the local area network of the AtoN site, while one of the controllers is responsible of communication tasks. This is also a gateway to the Remote Control and Monitoring Systems (RCMS) central server.

[5] Wave heights are computed by the developed method is used by METOC portal, which is operated by Marine Systems Institute of Tallinn University of Technology, http://on-line.msi.ttu.ee/metoc/.

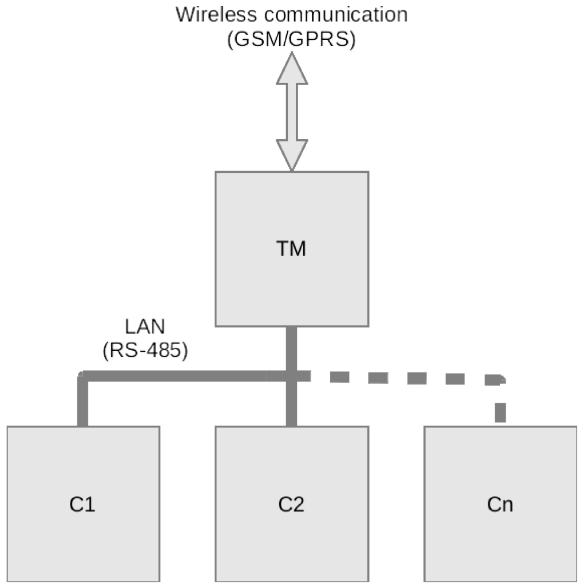

Figure 3.2: AtoN internal network

A typical marine AtoN system interconnection is presented in Figure 3.2 where TM has several communication tasks and acts like a network gateway, and C1, C2, …, Cn are internal controllers in charge for the AtoN site's mission. The main task of a TM is packet forwarding between wireless and local area interfaces; a TM may also be configured to fulfil some additional tasks like time synchronisation with GPS, certain measurement tasks or even flashing. Internal controllers C1, C2, …, Cn may be navigational lantern flashers, smart power supply system controllers, or measurement controllers.

3.2.3. Hardware Design Considerations of Telematics Module

In the beginning of the TM development it was known a priori that the new module should at have least the same functionality as previous NMT or GSM Data based modules. Previous TMs had capability for tracking buoy position by using GPS, sending module status to central server, communicate to other devices over LAN (Local Area Network), measure analogue input voltages and detecting state change on digital inputs. First improvement was using a new communication channel – previous TM uses GSM data call for transmitting data to server, new module uses GPRS connection, which has much higher transfer rate and was much cheaper. The rest of the functionality remains largely the same – lower levels of LAN communication was not changed, only some communication command parameters on LAN and GPRS link (formerly GSM data) were added. In the planning stage it was decided that microcontroller should have least three serial ports – one for GPS, one for LAN, and one for GPRS. Therefore it was required also to replace the old HC11 microcontroller

with a new AVR family microcontroller, this allows also to use effectively C programming language as the main language.

As Estonian buoys use batteries as a power source, it was essential that onboard electronics have low power consumption. Therefore it is required to use such microcontrollers that have lowest possible power consumption, even if this means less computational power and hardware capabilities. Typically are 8-bit microcontrollers suitable for AtoN devices, but also some recent 32-bit microcontrollers have similar parameters such as ARM Cortex-M0 and smaller AVR32 microcontrollers. In AtoN system 8-bit AVR microcontrollers are used – ATMega1280 [7]. The main reason for selection of this microcontroller was that most of the AtoN devices are developed at least a decade ago, and in that moment no 32-bit microcontrollers that had comparable power efficiency was available. Selected microcontroller have 8-kB of internal SRAM, 128 kB of program memory, 86 programmable input/output lines, 4 programmable serial ports, 5 timers/counters and 16-channel analogue-to-digital converter.

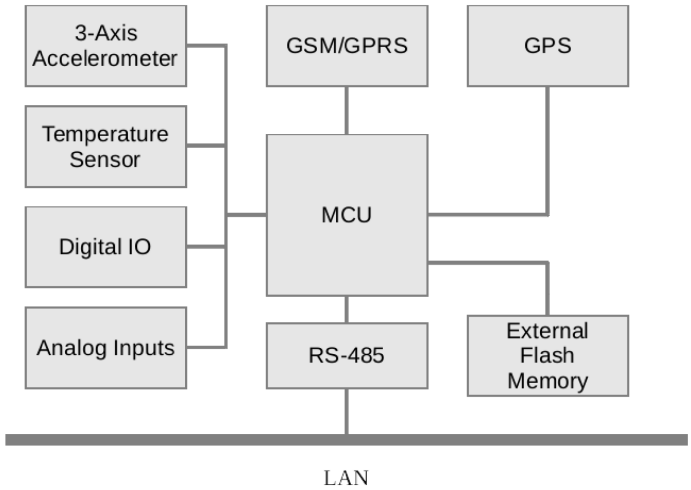

Figure 3.3: Telematics Module block diagram

Figure 3.3 shows relevant subsystems of a TM utilised for acceleration measurement, inclination angle calculation, digital input, LAN monitoring, and status/alarm communication tasks, leaving out all parts that are not relevant for programmers point of view, i.e. power supply. The central part of TM is MCU, which is ATMega1280. TM has also GSM/GPRS modem, GPS, RS-485[6] interface for LAN communication and 4-Mbit external flash memory for storing firmware images. MCU analogue inputs are connected to 3-axis accelerometer, temperature sensor and voltage input. This device has also several digital inputs and outputs. All analogue-to-digital conversions have 10-bit measurement values representing voltage levels, and all samples are acquired typically with a

[6] Standard for defining the electrical characteristics of drivers and receivers for use in balanced digital multipoint systems.

20 ms to 100 ms interval, with sequential delays of 0.2 ms between readings. In the current system implementation, the GSM/GPRS modem has integrated TCP/IP stack. 4-Mbit flash memory is needed for firmware updating – data communication is the slowest and most failure prone phase of the firmware updating process, and this external memory allows to buffer new firmware during update.

3.2.4. Telematics Module Software Design Considerations

In complex systems like AtoN systems, tasks are divided in-between submodules [28]. For this purpose, special message passing methods (different devices on one network) and shared memory resources (multiple tasks in one processor) are used. In AtoN systems, both message passing methods are used – there is at least one flasher and TM in buoys, wich is connected through local RS-485 network (Figure 3.2). The TM has multiple tasks that communicate by using shared memory (Figure 3.4). Local devices are connected through TM and the Internet to front-end server. This network topology is most optimal for AtoN devices. Only one device has connection to the Internet; this reduces complexity of flashers and other local area devices.

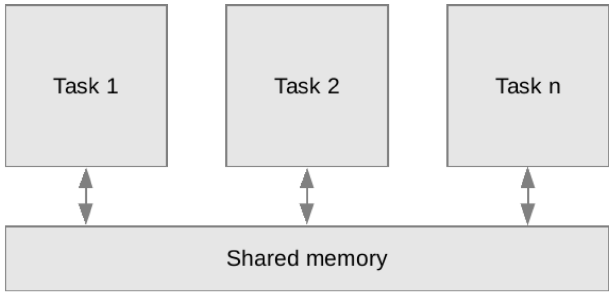

Figure 3.4: Communication between tasks by using shared memory

Every AtoN device, which is connected to local area network, has at least two different software modules, one for network and another for AtoN specific tasks. Although two modules are bare minimum, typically at least six modules are used. For example flasher, which is one of the simplest modules, has one additional software module for external flash memory and configuration memory, one for AD converter, one for flasher hardware, one for local area interface and one for control logic and shared memory (Figure 3.5). All described modules are connected through control logic modules and all message passing is realised with shared memory areas. TM has the same architecture but it has additional GPS and GPRS modules and does not have flasher module. This modular system is relatively easy to develop and to maintain. Basically all larger programs have similar architecture. All this communication and task slicing is possible with kernel, which has hardware abstraction and separation and has special message passing mechanisms.

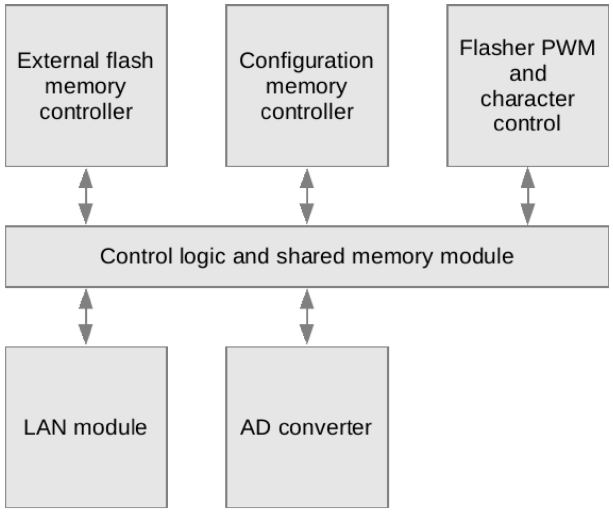

Figure 3.5: Flasher internal software modules

By using real-time kernel to share processor resource between tasks on a new TM, it was required to decide which programming language to use. Most reasonable language for TM was C at that moment. The main reason for C was that GNU C compiler has quite mature support for AVR microcontrollers. Assembly language was rejected by experience gained with the older TM; the old module had nearly same functionality but all software was written in assembler, and assembler does not give any significant advantage over other programming languages in such systems. Other programming languages like C+ + and Ada were not seriously considered at this time – C++ compiler was not able to compile larger program as effectively as C compiler and Ada did not have support for chosen microcontroller (GNU Ada compiler and runtime library for AVR was released several years later, and is still quite experimental). However, as C++ is grown out from C, it was possible to switch from C to C++ with small efforts. Issues that surfaced by using different programming languages or migrating from one language to another are described in Section 4.2 – Programming Languages – C and C++.

Due to the experience gained by previous TM development it was known that TM should perform several different tasks, and all were required to execute in parallel, nearly in real-time. To accomplish that, a complicated super-loop program or scheduler was needed, which could share processor resource between different tasks. The idea of using a super-loop program was not considered, instead it was reasonable to use a real-time kernel or scheduler. In public domain several different kernels exist and for described TM several kernels were nearly suitable. Writing our own kernel from scratch was considered too time consuming, and this option was ruled out. Several kernels found in the public domain had quite small memory footprint (FreeRTOS [85]) but had quite limited or no hardware driver layer. In our systems it was required

that hardware part is strictly separated from other code. The separation was needed for testing, because otherwise it is relatively difficult to create test program either manually or automatically; such programs are discussed in Section 4.3 – Program Structures and Improvements on Testing. Other kernels (like eCos [101]) had a well separated driver layer and lot of supporting kernel functions, but also required relatively powerful microcontrollers – such microcontrollers have quite high power consumption and relatively long wakeup time from sleep mode, therefore more powerful microcontrollers or kernels were not considered for our systems. Some kernels were also certified for IEC61508 SIL 3 level (SafeRTOS [98]), but are quite similar to FreeRTOS and target mostly 32-bit microcontrollers. Some microcontroller manufactures have also their own kernels, like Texas Instruments, which have TIRTOS [94], but those kernels support mostly their own microcontrollers. If kernel supported microcontrollers go out of stock then using same source code on other microcontrollers is difficult as it may have lot of microcontroller specific functions and certainly need porting (if this is allowed by licence policy).

Kernels that were nearly suitable for AtoN systems had two major drawbacks – they did not have any automatic power saving support and no thread safe watchdog handling. As AtoN systems are mostly battery powered and should consume minimal amount of energy, it is required that microcontroller enters sleep mode when it has no tasks to perform. But all events that occurred during sleep mode should be completely processed. For example, when the serial interface receives any data during sleep mode, it should be completely received and processed[7]. Therefore, the only one possible way to return from low power mode was by disabling it from interrupt handler. Another problem was entering to the sleep mode – decisions that allows to enter sleep mode should be based on hardware states; sleep modes are disallowed when any of the microcontroller communication interfaces is active, for example when the serial port is sending or receiving data. Therefore, a function that set microcontroller to sleep mode should be aware about all peripheral hardware states; when some device is active, it is not allowed to enter sleep mode.

Another issue with kernels from public domain is lack of support of shared watchdog control. A watchdog timer is required for resetting microcontroller when program stays in a state that is not desired, i.e. enters a dead loop after encountering a program bug. As a microcontroller has only one watchdog and to reset the watchdog it is required to write into special register or execute watchdog resetting instruction, it is possible to reset watchdog only in one

[7] For this a different communication protocol was required – serial interrupt is triggered after one byte is received; this first byte is usually received while microcontroller is in sleep mode, and therefore at least the first byte is lost. To receive data, the controller should be woken up from sleep mode, therefore at least one dummy byte should be sent from transmitting side, which then will wake up a controller.

thread, and this thread should take into account other thread states. Suitable solutions for watchdogs are discussed in Section 4.4.2 – Multitasking Programs and Watchdog. Above mentioned difficulties made it impossible to use all known kernels in AtoN devices.

As writing of our own kernel was too time consuming task, the only choice was to take a kernel that satisfies most of the requirements and write missing functions. Kernel that satisfies most of our needs was NutOS [51]. This kernel has all required features and it was relatively small, but it does not support very well different sleep modes and had only basic watchdog handling functions. It has also TCP/IP stack and web server and functions for AT command parsing. Chosen kernel requires only minimal modifications – it was needed only to add automatic power saving modes, to improve watchdog handling and RS-485 based LAN driver.

In the next step it was required to choose coding standards and development methodology. While TM is a mission critical device[8], it is beneficial to follow as much as possible best practices in that software development field, including using coding standards or guidelines. Unfortunately, any coding or any other software related standards or guidelines for devices that are used in AtoN hardware part were not available at the beginning of the TM development. But, in later stages of the initial development it turns out that MISRA C (Motor Industry Software Reliability Association) coding guideline is nearly suitable for this product. In Section 3.3 – Standards are given overview about different standards on TM software.

During the first weeks of new TM development, cowboy coding was used. After a few weeks agile methodology like development was taken in use, however, instead using automated testing tools, all testing was done by hand. This testing methodology largely dominates the entire TM software development cycle, but later when this product was improved by some additions, TDD (Test Driven Development) was used in some extent and in one TM related product BDD (Behaviour Driven Development) was used. Some extensions were added by using waterfall model. In Section 4.1 – Embedded Software Development Processes different methods and their suitability for AtoN device software development are discussed.

3.3. Standards

This section highlights relevant standards for embedded software development, especially standards that are useful for AtoN device development.

In embedded software development mainly two type of standards are used: coding style standards (guidelines) and standards that are related to specific programming languages and their capabilities. Safety critical systems have additional safety related standards like IEC61508 [45], and in some software

[8] By the usage area it may be also safety critical, but similar devices are currently declared as non safety critical devices.

developments, it is beneficial to follow ISO9001 quality management standard [35]. In the context of this thesis, the IEC61508 standard is not relevant. Currently it is not required that embedded AtoN systems corresponds to any safety integrity levels.

3.3.1. Style Guidelines

Coding style guidelines are documents or standards that are used mostly in larger projects where several developers are involved. These guidelines are typically related to organisation, developer group or some product. However, no official coding standard, i.e. ISO standard, exists. The main aim of the coding standards is to specify coding format so that all developers would write using the same style. Same coding style is mainly required to ease the software maintenance – a significant amount (50%-90% [11, 24, 57]) of the lifetime cost of a software goes to application maintenance, and most of the software is maintained by several different developers during its lifespan. One of the best written coding guideline for C is the NASA C coding style guideline [49], which has a quite detailed description of C source code layout. As embedded software development is quite a resource consuming task, it is reasonable to ease this process and use such guidelines when there is more than one developer.

3.3.2. Coding and Programming Language Standards and Guidelines

Most programming languages and libraries, which are used in embedded systems, contain several insecure functions or have possibilities to construct such functions that may have unpredictable side effects [35, 71, 72]. In safety and mission critical systems or programs where testing and maintenance have quite large proportion in program's overall development, it is beneficial to disallow to use such functions. Therefore, industry-wide best practices are published, written as coding guidelines. But it does not mean that when following these coding guidelines the resulting program is free of bugs. There is even no clear evidence that directly following the guidelines reduces the bug rate significantly [14, 36]. However, it will allow to write more maintainable programs in less time.

While most AtoN systems have many mission and safety critical characteristics, there are no known specific guidelines meant for this software segment. The most suitable guidelines for AtoN software are MISRA C [71] and MISRA C++ [72]. Both are used in automotive industry. The JSF C++ coding standard [55] is used in Joint Strike Fighter F-35 program and in some extent, the JPL C coding standard [50] is also used. All listed guidelines discourage the use of code constructs that produce hard-to-maintain code, specify naming conventions and commenting style, have rules for complexity limits. According to MISRA and other guidelines it is not recommended to use such functions that are able to fail stochastically. One of the most notable functions of this kind is *malloc*; it may fail very unexpectedly due to the

unavailability of continuous memory. Since MISRA and similar guidelines have relatively simple rules, there are several programs that are capable to check the violations of these rules. These programs are mostly *lint* and its derivatives i.e., *PCLint*. Alternatively, it is possible to detect large number of violations by setting appropriate compiler flags or even by using simple scripts. The following section describes several programming methods and also note their compliance with listed guidelines.

All the programming languages have typically their own standards. For example, the C language has an ISO/IEC 9899:2011 standard, which is called C11. There are also C89 and C99 standards. The C++ has ISO/IEC 14882:2011 standard, which is called C++11. These standards describe compiler and several library functions. Unfortunately, most embedded compilers do not fully support official language standards and may have their own implementations; it is not uncommon that some non-significant part from 20 years old standard is not completely supported. However, above mentioned standards are most effective in the following cases: when there is a need to create portable program, or when a program has to meet some other standard like IEC61508. It is elementary that a portable program source code should meet some common standards; it is quite rare that two different compilers for different architecture have exactly the same functionality and types, even the different versions of compilers may have quite different features. Determination of a language standard increase probability to detect possible bugs by compiler. In cases when embedded software testing is carried out on a development computer and the source code contains such library functions that are available on both architectures, it is required to specify at least a language standard. Meeting the requirements of a language standard is also necessary as the program may need to meet some other standards as well. This requirement arises in certification process, as certification body needs to use the same environment that was used during the development. In rare cases it is possible to avoid this requirement but in such cases the decision should be justified. For example, when some CPU has only one compiler but this compiler does not meet the standards.

To conclude this subsection we can say that in embedded software development, it is highly recommended to specify a language standard and to follow the coding guidelines.

3.4. Challenges in Telematics Module Software Development

This section describes problems, which raised during TM development, and improvements, which were done during the development of a new TM. Also a short background information about each improvement is given. The Chapter 4 presents solutions for every problem.

In the beginning of the TM software development process it was expected that bugs may show up during development process, but it was not clear how much and how these bugs will affect overall development and program behaviour. The primary concern was how watchdog behaves when bugs are

encountered. It is quite well known how watchdog behaves in super-loop programs, but we did not have any information on how it behaves and how it can be controlled in multitasking programs. In first program versions watchdog was reset in *idle* loop, and as expected, when TM software freezes by the result of some bug and program has least one sleep functions in frozen code, then watchdog was still reset in *idle* loop and reset did not occur. During the TM software development it turned out, that in multithreaded programs such watchdog resetting mechanism was needed which take into account states of all threads. During development process two different watchdog resetting mechanism were designed, both are described in Section 4.4.2 – Multitasking Programs and Watchdog.

In desktop computer program debugging a wide range of different debugging tools are available. However in majority of cases in embedded software development, mainly simple OCD is used. It was complicated to use only OCD based methods in TM software development, the OCD shortcomings show up in the very beginning of TM software development. Using OCD for program debugging in software where interrupts are used only few times and running program is a super-loop type program, makes the use of OCD quite straightforward and it reduces significantly development time. However, for more complex programs like multithreaded programs that use periodically at least one asynchronous interrupt, the OCD debugging becomes quite complicated or in some situations even impossible. The main reason for it is that significant amount of bugs show up some time after error occurs, such as, most stack overflow bugs. When bug shows up some time after it had happened, it is not possible to detect the cause of the bug by using simply OCD. In order to find out which process was involved when the last error occurred, it is required to track writes to specific memory areas. When at least one asynchronous interrupt is allowed, then after firing it is program counter set to current interrupt handler. While program counter value does not point to observed program area, then this in turn it does not allow to use program step-by-step walk through. Therefore finding a cause of bug in such situations is quite complicated task. In TM development several above described situations were encountered. In above described reasons the OCD is used occasionally for debugging. It is used only in rare situations where it is known that any asynchronous interrupt can not show up and no threads are started.

To cope with above mentioned OCD shortcomings, two well known methods for debugging were introduced – first, predefined debugging information outputting over serial line and second, changing microcontroller output state depending on program state. Debugging information outputting over serial line was preferred approach, but when this approach is unsuitable, output state change monitoring can be used as a backup debugging method. Both are quite robust and require a lot of developer interactions, like program recompilation and uploading to microcontroller memory, but unlike OCD it is possible to watch program states in very different situations. As both debugging aids require fast communication with hardware, methods were developed that are

suitable for this purpose; these are described in Section 4.3.5 – Debugging and Testing.

The new TM was intended to replace the old HC11 based telematics module that has different endianness. Therefore it was also required to change byte order in several places. Mostly this was needed for communication with monitoring server or configuration software, also byte order change was also required for FOTA. Changing byte order effectively is described in Section 4.5.4 – Byte Order Manipulation.

As noted in the beginning of this chapter, TM has a lot of different functions and quite wide variety of supported hardware – this module allows to use three different modems and two different GPS receivers. When major difference between GPS receivers was serial interface baud rate, then modems had completely different AT commands for TCP/IP connection, for entering the power save mode and for network selection. Program that copes with such different hardware is most reliable when written using C++. In the beginning of the development of TM a reliable C++ compiler for AVR was not available and therefore the only choice was to write all software in C using structures and functions pointers, which is discussed in Section 4.2 – Programming Languages – C and C++. Described method is quite well known in non deeply embedded systems, but in deeply embedded systems it is used quite seldom. Within this thesis one possible way to use such solution effectively in deeply embedded systems is given; in this solution, function pointers are placed to different memory, which allows to reduce microcontroller RAM consumption.

An important improvement in TM software was *malloc* like function, which was backed by memory pool. This improvement was required to cope with the drawbacks of standard *malloc* – during compilation it is not possible to know how much free memory the microcontroller has left, and in several cases it can be discovered when a program was loaded to microcontroller memory and started. Developed *malloc* replacement function uses predefined memory pool; this enhancement allows the compiler, linker and diagnostic tools to analyse memory requirements before a program is loaded to microcontroller memory. This improvement is described on Section 4.6 – Dynamic Memory.

During the last development stages when most TM functionality was implemented, it was important that new functions take minimal amount of microcontroller RAM and stack. One method for limiting memory consumption was to use inline functions. While compiler has quite good support for it, this support is more likely targeted for larger systems – while inlining large functions, compiler always warns about program growth. As it is not wise blindly ignore the compiler warnings, small research was carried out about inline code and its peculiarities. Results of this small research are presented on Section 4.5.2 – Program Code Inlining.

In later TM versions it turns out that in some TM usage areas it was essential to encrypt data. As TM has only 8-bit microcontroller, then only feasible and

still enough strong encryption is AES. In this thesis two optimisation methods are presented, which can be used with AES encryption when it is used in small microcontroller. This work is presented in Section 4.5.5 – Optimisation of AES Cryptographic Functions.

In TM development it was decided to write TM software as modular as possible – at least one module is related directly to hardware and another to control logic; for this case it is reasonable to use different programming languages for different tasks or modules. Also initial TM software was written completely in C, but lot of larger improvements were done by using C++. In this thesis some methods are given that allow to use different programming languages more effectively in one embedded program; this is described in Section 4.2 – Programming Languages – C and C++ and in Section 4.3 – Program Structures and Improvements on Testing.

3.4.1. Later Improvements

Although the initial TM software was quite reliable, still some functions should have been implemented in different way, or revealed some bugs. To support bug fixes and improvements on fielded devices, it was required to develop FOTA capability. To use FOTA with TM it was required to create a new bootloader, either which allows to directly load software from server or copying program form external memory to microcontroller program memory. In TM it was practical to use the second solution – TM has integrated additional flash memory that stores several memory images and the bootloader uses an image that has been pointed by configuration [65]. This solution allows to store at least one working software version and to use it when new version does not work on this controller. Software rollback was needed in cases when new software version does not work on some specific controller, or a controller has such configuration that does not allow to run with new software version. In addition to bootloader improvements, small research for testing super-loop programs (Section 4.3.1 – Super Loop Programs) and using watchdog with multithreaded programs (Section 4.4.2 – Multitasking Programs and Watchdog) was carried out.

As FOTA is used for firmware updating, it allows to use programs that contain unit tests to test hardware; this feature turned out to be very useful in situations where TM had some component failure. To use unit tests on microcontroller it was required to develop a lightweight unit test framework. Although the developed framework can be used for testing PC programs, there are other frameworks, which can be found in public domain that are better suited for this purpose. Using unit tests in embedded systems development is discussed in Section 4.3 – Program Structures and Improvements on Testing.

3.5. Heel Angle Calculation and Buoy Collision Detection

Another area where it was possible to use acceleration sensor was heel angle calculation [67] and buoy collision detection [68] both operations carried out on

microcontroller. In collision events, buoy onboard electronics may become inoperational quite fast and therefore collision detection and reporting should be as fast as possible.

To detect collisions with a navigational buoy, must continuously monitor acceleration signals from all three axes. Assuming that this method is used only with navigational buoys, the sampling period can be set quite low but not below 20 ms. In a typical buoy installation can be assumed that a collision may appear from any direction and therefore must take into account signals from all three acceleration axes. To detect a collision event in case of unlimited computational resources available, one would calculate the acceleration vector length, taking into account all acceleration values, and base the decision on that vector length. In our case, the system has rather limited amount of memory and computational capability, therefore it is not practical to calculate the vector length; instead, it is possible to achieve almost same results by adding up the acceleration values. Using only summation, must take into account the fact that during collisions is returned much higher resulting acceleration than in case of using acceleration vectors; this is usually the case when an impact comes in between two or three axes. Due to the specifics of installed TM, it is possible to tolerate errors that are introduced by higher acceleration values, also it does not need to get very exact collision values, but it is only required to know when acceleration value exceeds certain threshold level. Therefore in order to detect collision from total acceleration the static (DC) component from obtained signal is filtered out. To filter out the DC component an IIR filter is used. After DC level removal, the second stage of filtering is applied, which plays a major role in collision detection system. A collision detection filter should be rather fast, with acceptable filter delay. In our case of less than 1 second. This filter must be also quite robust in order to avoid false collision reports. Therefore, it was required that this filter is partly a pure averaging filter and also a low-pass filter (LP filter).

Figure 3.6 presents a simulation of buoy collision event. For simplification, all three acceleration vectors are summed up, and the Earth's gravitational acceleration is subtracted. Resulting acceleration signal indicates whether a collision of the buoy with a ship has been encountered. As is shown in Figure 3.6, with sufficient acceleration it is possible to get collision events in reasonable time. In that simulation, first event may be recorded 50 ms after the first acceleration peak and the second one 400 ms after the first acceleration peak. In addition, several tests were carried out where the TM was mounted to a heavy object and the test object was hit with another object, i.e. an artificial collision was created, and the results were comparable to the simulation in Figure 3.6.

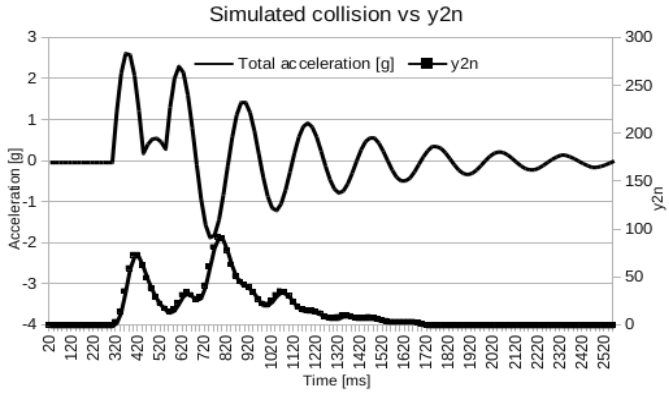

Figure 3.6: Simulated collision [68]

Developed solutions utilises an onboard acceleration sensor; for detecting collision events acceleration from all axes is time filtered, and when the filtered value exceeds a threshold value, a corresponding alarm is issued.

For heel angle detection autonomous angle calculation was required; when a buoy is submerged below the ice (Figure 3.7), then flashing of navigational light is no longer required and it can be stopped, but decision for this can be done only onboard (when buoy is submerged then is not possible to communicate with a server).

Figure 3.7: Buoy in Ice [67]

Calculation of the buoy inclination angle based on digitised real-time acceleration data can be performed by using simple trigonometric functions like sine or tangent. For systems that have hardware floating point support, the most elegant and easiest way would be to use tangent. But in 8-bit embedded systems where all numbers have quite small range, the only feasible option is to use the sine function. Inclination angle was calculated in two stages: first controller calculated the intermediate heel angle value, which then was sent to server that calculated the remaining angle value. This intermediate value can also be used

for alarm triggering in TM software. As buoy movement may have great influence to calculation outcome, it was required to take this also into account, in this case was used acceleration value averaging.

By developing above mentioned features some research for limiting functions parameters was required (Section 4.5.1 – Limiting Function Arguments), unit tests and limiting program dependencies (Section 4.3.2 – Minimising Relations Between Submodules). Large number of function argument turns out problems on both calculations – all acceleration calculations had real-time constraints, these calculations should be carried out in less than 1 ms. As the used microcontroller had only 8 kB of RAM and every thread had 256 to 512 bytes of stack, it was not reasonable to pass acceleration values to calculation functions by parameter coping; instead structures and pointer passing were used. Another issue was with testing; it was not possible to make real buoy collisions or measure real buoy heel angle for testing the developed methods. Therefore it was required to test in simulated environment, but for simulation was required to separate hardware functions and limit all other program dependencies.

3.6. Wave Height Calculation by Using Navigational Buoys

Another research grew out from the additional TM hardware – this module has onboard triaxial accelerometer, and TM was able to send acceleration values to central monitoring server. It was quite easy to notice that acquired acceleration data from buoys have very large periodical component, and this component has nearly the same period as typical sea waves have. From this observation a research for wave height detection by using navigational buoys was initiated [69].

Proposed method allows to use typical steel spar buoys with the with of roughly 5 tons (Figure 3.8). These buoys are deployed for around the year operation, capable to withstand ice conditions. Chain moorings are used as standard, increasing overall buoy weight by 0.5 to 1.5 tons, and also keeping the buoys from riding the waves freely. Since the primary task of these buoys is to serve as a source of a navigation light signal, they are designed in a way allowing only limited wave following.

Figure 3.8: Steel spar buoy [69]

To obtain wave height values, a buoy acceleration amplitude is used which is corrected by peak frequency, taken from FFT (Fast Fourier Transform). Developed approach is similar to NDBC (National Data Buoy Center) [22] published approach, except NDBC method uses maximum amplitude and frequency to find wave height, but the developed method uses average of acceleration peaks and corrects it by buoy own movement period.

In order to validate the obtained wave height data, the Estonian Maritime Administration, Cybernetica AS and the Marine Systems Institute at Tallinn University of Technology have performed trials since late 2008 to establish feasibility of such wave height measurement network based on navigational buoys. Even if navigation buoys are not ideal wave following platforms, it is still possible to calculate a rather close approximation of the actual wave height based on their acceleration. Tests and validation of the wave height estimation method were performed in five reference measurement sessions in three different locations, each lasting at least two weeks. In all cases the reference sensor was deployed at a distance less than 3 nautical miles from the buoys under testing (Figure 3.9). Pressure based wave gauge was used for reference measurements performed by the Marine Systems Institute at Tallinn University of Technology.

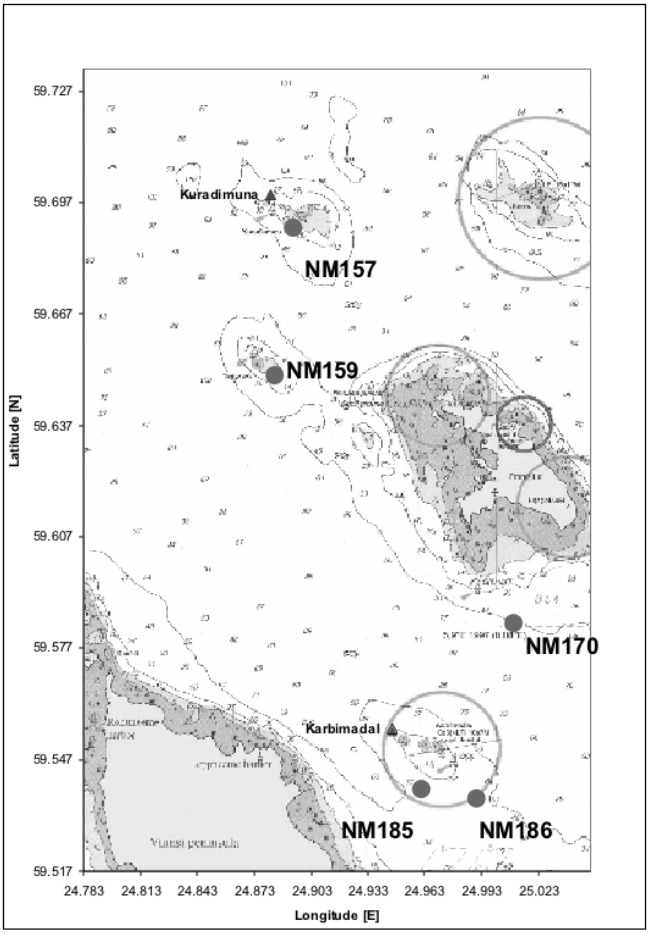

Figure 3.9: Location of navigational buoys hosting the acceleration sensors used in the wave parameter measurement experiment, and pressure based wave measurement equipment used for reference measurements shown with triangles [69].

Results of the comparison of reference and calculated wave heights are good. Measurement periods captured different wind conditions and wave field realisations. Two datasets fit with each other very well for waves below 2 m, 95% of the resulting wave heights differed from the reference wave heights by less than 41 cm. In case of wave heights of over 2 m, the maximum difference was 86 cm (Table 3.1 and Figures 3.10-3.13), although the number of such larger wave heights was probably not sufficient for drawing a proper statistical conclusion. Certain errors can be at least partly attributed to the different measurement and reporting intervals and sometimes short data acquisition periods, with both due to the fact that the primary task of a navigational buoy is AtoN signalling. Nevertheless, both errors have almost negligible impact on measured wave heights. Another issue is natural variability on wave field,

which play role if there is distance between navigational buoy and reference measurement site and always it is.

Percentage of calculation results within the maximum difference	**Maximum difference in calculated significant wave height [m]**	
	Range: 0.0 m to 2.0 m (21794 reference points)	***Range: 2.25 m to 5.0 m (401 reference points)***
68.27%	0.29	0.63
90.00%	0.37	0.78
95.00%	0.41	0.86
95.45%	0.41	0.87
99.73%	0.53	1.10

Table 3.1: Differences between wave height pressure based reference measurement and calculated results [69]

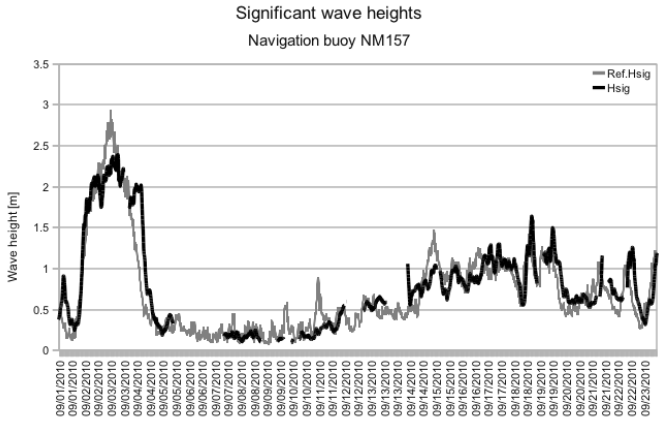

Figure 3.10: Results of the first test period on buoy NM157 (Sept. 2010) [69]

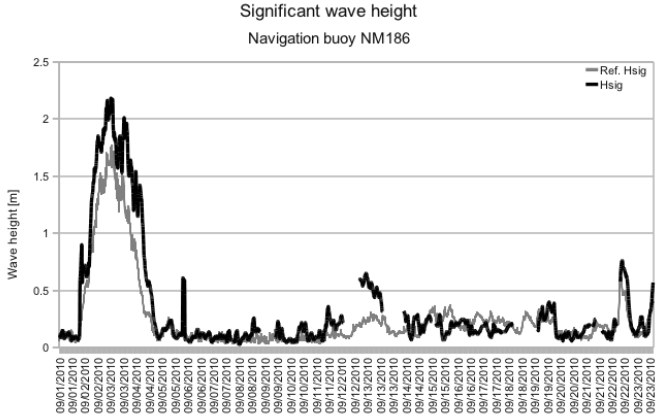

Figure 3.11: Results of the first test period on buoy NM186 (Sept. 2010) [69]

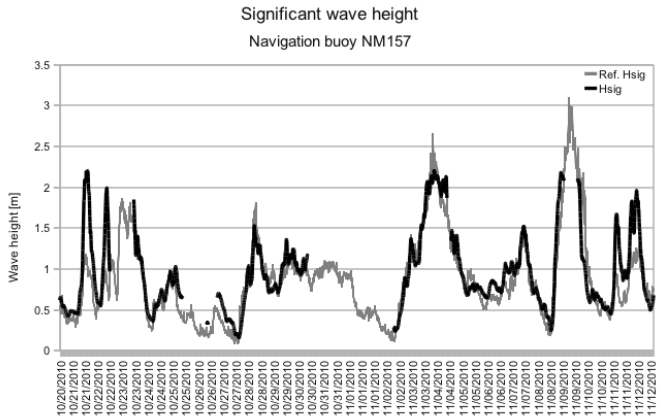

Figure 3.12: Results of the second test period on buoy NM157 (Oct.-Nov 2010) [69]

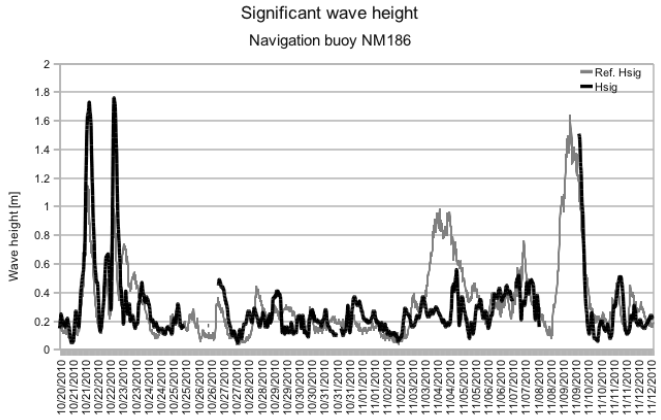

Figure 3.13: Results of the second test period on buoy NM186 (Oct.-Nov. 2010) [69]

One outcome of this research was wave height calculation program, which take input data from buoys and send processed wave height to a public METOC web portal. Wave height calculation program utilises TM as acceleration data source – TM that is installed to navigational buoy sends measured acceleration data to central monitoring server with predefined interval. The monitoring server stores acceleration data and notifies the wave height calculation software about new data. After receiving message about new acceleration data, the wave height calculation program retrieves new acceleration data, calculates wave heights, binds calculated values to geographical location, and sends results to database and to a third party server.

3.7. Conclusions

This chapter provided background information on developed improvements – it describes systems that are used for marine visual navigation (AtoN systems). The chapter described mainly problems that raised during development of new TM. In order to solve these problems, it was required to develop new methods in embedded software development. This chapter gave a short descriptions about developed improvements to achieve design goal. The technical descriptions are in the next chapter. The new module replaces the old communication module, also a new communication channel was taken in use and several improvements like buoy heel angle calculation, collision detection and wave height measurement were implemented.

AtoN systems, which are used in Estonia, contain buoys, lighthouses, central server and synthetic AIS network. In lighthouses and buoys TM is the main component; this module is required for transmitting data from AtoN internal devices to the monitoring server and it is also capable of transmitting some monitoring information, like acceleration data. Main part of TM is a microcontroller that is responsible for LAN and GPRS communication; it has also several analogue inputs and GPS. While TM has several interfaces and concurrent tasks, a simplest way to control those interfaces and share CPU resource between tasks is to use a kernel. In TM a heavily modified version of NutOS is used.

At development of above described module it was required to carry out research in embedded software development. Therefore, using watchdogs and OCD in multitasking programs, automated tests, programming languages, dynamic memory, inline code and optimisation were researched. In addition to research, methods were developed for buoy collision detection, heel angle calculation and raw acceleration data transmission to central server for wave height calculation. All results of described developments and research are presented in the next chapter.

4. THE ADVANCES IN EMBEDDED SOFTWARE DEVELOPMENT

Chapter 3 outlined problems that surfaced when developing a new TM firmware. This chapter gives solutions for previously mentioned problems, lists relevant standards, discusses about the use of different programming languages and testing methods, and discusses how testing methods depend on a program structure. The chapter also outlines peculiarities about multithreaded programs, places where it is possible use such optimisations that are not achievable by compiler, and finally, the use of dynamic memory. The following section presents software development processes that are suitable for the use in low-power microcontrollers. All the presented improvements are developed during TM and other AtoN devices software development by researching optimal solutions for the real-time kernel and supporting libraries. All the published papers use the described solutions as supportive methods and required improvements. The main field where the described methods best suit are the 8-bit and smaller 16-bit microcontrollers; although same applies for larger processors as well.

4.1. Embedded Software Development Processes

This section describes different development processes and outlines the most appropriate solutions for embedded systems. For desktop computer programs several different software development processes exist but lot of them have such properties that make them unsuitable for small embedded systems. The following sections give an overview about some software development processes, which are used in AtoN onboard embedded software development. In addition to description of software processes, a short overview of the use of UML in small embedded systems is also given. Examples of using main parts of agile processes are given – the unit or automated tests on small embedded systems. Presented automated tests require several improvements that are described in following chapters – unit testable programs should have minimal amount of relations with other code, test programs are usually written in C++ and unit testable programs tend to require little more resource than programs that do not support unit tests.

4.1.1. Code and Fix – Cowboy Coding

Code and Fix, also known as Cowboy Coding (as used in the context of this thesis), is a software development philosophy where programmers have autonomy over the whole development process – control of the project's schedule, languages, algorithms, tools, frameworks and coding style [60]. A cowboy coder is usually a single developer who has very little or no participation in end-user or management. As this development philosophy has

no formal management, it is quite complicated to use it in a commercial embedded project. However it has also several advantages. In some cases, mostly in smaller programs, absence of formality significantly reduces the efforts to develop a program. Therefore it is the most optimal solution for experimenting with new hardware and for prototyping purposes. However, most of the prototypes are later rewritten by using some other development philosophy.

In embedded systems, the Cowboy Coding is used quite widely. It is used mostly in smaller systems, which are not safety related like hobby or student projects or even in smaller commercial projects such as small programmable components that have quite limited functionality like a simple voltage regulator. It is quite common that this development philosophy is used in early stages of large software development. In general, programs that are written by using this philosophy are quite small, less than 500 code lines in C or less than five function points[9].

4.1.2. Unified Modelling Language

UML is a standardised (ISO/IEC 19501:2005), software modelling language, it provides a set of graphic notation techniques to create visual model of object-oriented systems. The UML combines several levels of modelling techniques, it has business modelling, object modelling and component modelling, therefore it can be used with most processes throughout the software development life cycle. Although UML is quite universal modelling language, it is not intended for embedded and real-time domain, but it has several extensions for this.

While UML has several extensions, which allow it to be used in embedded or real-time systems, using UML in small embedded systems is quite problematic. The main reason for this is that most UML tools use object-oriented languages like C++, and such program constructs require a lot of memory or CPU resources. Therefore, microcontrollers where UML is usable should have significant amount of RAM and program memory – at least 1 kB of RAM and 20 kB of program memory. Also programs that are created by using UML, tend to be little slower than other programs where UML was not used. Due to the high demand of resources, the UML tools are not widely used in 8-bit microcontroller software development.

4.1.3. Agile Practices – Test Driven Development and Behaviour Driven Development

This subsection provides an example for using unit tests in embedded software. This example shows testing of input and output functionality; to use this

[9] Function points [40] give relatively accurate estimates for business type applications but not for scientific or mathematics applications [90]. Embedded AtoN systems do not contain complicated mathematics and are more like business type applications.

example in real program, it is also required to use similar methods that are presented in Section 4.3 – Program Structures and Improvements on Testing.

TDD (Test Driven Development) is a software development process where tests are written before writing the real code. This process relies on the repetition of a very short development cycle: first the developer writes an initially failing test case, which defines a desired improvement, then he produces the minimum amount of code to pass that test, and finally refactors the new code. This methodology ensures that the source code is thoroughly unit tested and eventually leads to modularised, flexible and extensible code. BDD (Behaviour Driven Development) is similar to software development process like TDD, but it combines the general techniques and principles of TDD with ideas from domain-driven design and object-oriented analysis and design. The principal difference of both methods is the idea who write tests or specifications, in TDD is developer responsibility to write tests, but in ideal BDD somebody else should write the specifications like business analysts. In the following code listing is an example describing one software module, which counts input changes and after third change toggles the output. In a TDD case, the following unit test is needed for this described requirement:

```
 1:/* Initialization of the test, reset internal registers. */
 2:void FnTest::setUp ()
 3:{
 4:    reset_regs ();
 5:}
 6:
 7:/* Test for six consequentive input changes */
 8:void FnTest::six_changes ()
 9:{
10:    /* Initial state test. */
11:    CPPUNIT_ASSERT_EQUAL (0, (int)get_output ());
12:
13:    /* Set input value to 1, check output, it should be 0. */
14:    update_input_val (0x01);
15:    CPPUNIT_ASSERT_EQUAL (0, (int)get_output ());
16:
17:    /* Same input condition, output should have same values
18:     * as in previous test. */
19:    update_input_val (0x01);
20:    CPPUNIT_ASSERT_EQUAL (0, (int)get_output ());
21:
22:    /* Same input conditon, output shold have value 0. */
23:    update_input_val (0x01);
24:    CPPUNIT_ASSERT_EQUAL (1, (int)get_output ());
25:
26:    /* Same as previous test. */
27:    update_input_val (0x01);
28:    CPPUNIT_ASSERT_EQUAL (1, (int)get_output ());
29:
30:    /* Same as previous test. */
31:    update_input_val (0x01);
32:    CPPUNIT_ASSERT_EQUAL (1, (int)get_output ());
33:
```

```
34:    /* After setting input value to 1 should output reset
35:     * to 0. */
36:    update_input_val (0x01);
37:    CPPUNIT_ASSERT_EQUAL (0, (int)get_output ());
38:}
```

Listing 4.1: Simple TDD example.

In the Listing 4.1, the lines 2 to 5 are the internal registers initialisation, which are required to ensure that all tests start in the same state. When some other tests have been run before, the register states may be nearly unpredictable. The rest of the program is test itself (lines 8 to 38): in line 11, initial output state is tested, after that, the input register value is updated (lines 14, 19, 23, 27, 31 and 36) and lastly an outcome after each change is tested (lines 15, 20, 24, 28, 32 and 37).

While using BDD the following behavioural description needs to be written:

```
 1:/* Behavioural description for imaginary hardware controller. */
 2:DESCRIBE(update_input_val, "6 calls to "
 3:                           "update_input_val")
 4:    /* Reset internal registers, and verify initial
 5:     * conditions. */
 6:    reset_regs ();
 7:    IT ("returns 0 on initial state")
 8:        SHOULD_EQUAL ((int)get_output (), 0)
 9:    END_IT
10:
11:    /* Do first input update and verify output, least 6
12:     * test cycles are required. */
13:    update_input_val (0x01);
14:    IT ("returns 0 after first input update")
15:        SHOULD_EQUAL ((int)get_output (), 0)
16:    END_IT
17:
18:    update_input_val (0x01);
19:    IT ("returns 0 after second input update")
20:        SHOULD_EQUAL ((int)get_output (), 0)
21:    END_IT
22:
23:    update_input_val (0x01);
24:    IT ("returns 1 after third input update")
25:        SHOULD_EQUAL ((int)get_output (), 1)
26:    END_IT
27:
28:    update_input_val (0x01);
29:    IT ("returns 1 after fourth input update")
30:        SHOULD_EQUAL ((int)get_output (), 1)
31:    END_IT
32:
33:    update_input_val (0x01);
34:    IT ("returns 1 after fifth input update")
35:        SHOULD_EQUAL ((int)get_output (), 1)
36:    END_IT
```

```
37:    update_input_val (0x01);
38:    IT ("returns 0 after sixth input update")
39:        SHOULD_EQUAL ((int)get_output (), 0)
40:    END_IT
41:END_DESCRIBE
```

Listing 4.2: Simple BDD example.

In Listing 4.2, in lines 2 and 3, the test name and short description of this specification is described. The specification body is between the lines 4 and 41, and the line 41 shows the end of the specification. The line 6 shows register initialisations, and in the lines 7 to 9 initial conditions are verified. The line 7 describes what will be done, the line 8 tests results to given value, and the line 9 is the last line of this test. Same applies to all other lines which are described between IT and END_IT lines. The only difference between TDD is that before tested function result are updated input values. The input updating is shown in the lines 13, 18, 23, 28, 23 and 37.

The above described test and specification corresponds to the following C source code:

```
 1:/* Variable for output port states. */
 2:static uint8_t port_state;
 3:
 4:/* Counter for tracking input switches. */
 5:static uint8_t sw_cnt = 1;
 6:
 7:/* Reset all internal registers, this should correspond to
 8: * microcontroller reset. */
 9:void reset_regs (void)
10:{
11:    port_state = 0;
12:    sw_cnt = 0;
13:}
14:
15:/* Update output and internal counter accordingly
16: * microcontroller input value (function parameter val). */
17:void update_input_val (const uint8_t val)
18:{
19:    /* Change output only when microcontroller input pin
20:     * has logical high. */
21:    if (val == 0x01)
22:    {
23:        /* Output should toggled after three positive
24:         * input tests. */
25:        if (sw_cnt >= 2)
26:        {
27:            /* Toggle output port bit 0, and set switch counter
28:             * to 0 (initial state) */
29:            port_state ^= 0x01;
30:            sw_cnt = 0;
31:        }
32:        else
33:        {
34:            /* Increase counter that holds number of positive
35:             * input tests. */
```

```
36:             sw_cnt++;
37:         }
38:     }
39:     /* Else not needed.. */
40:}
41:
42:/* Return state of the microcontroller output port. */
43:uint8_t get_output (void)
44:{
45:     return port_state;
46:}
```

Listing 4.3: Program code which corresponds to the previous TDD and BDD examples.

The Listing 4.3 defines in the lines 2 and 5 two static variables, where the first holds port state and the second is switch counter. The lines 9 to 13 show the reset function, which set *port_state* and *sw_cnt* to initial state and also corresponds to microcontroller reset. In the lines 17 to 40 are functions, which were described by test on Listing 4.1 and specifications on Listing 4.2, these functions update output register states as described on tests or behaviour description. The lines 43 to 46 show the defined function, which returns port state to what was changed by *update_input_val* function.

Unit tests and behavioural descriptions are mainly intended for testing business or control logic, also both allow to lock down the program behaviour according to specifications. Unfortunately, both are relatively difficult to be used in hardware dependent code. Code that is written to be tested automatically (unit or behaviour tests) should be hardware independent; this allows to run tests on development computers with different architecture. Above described tests and testable code can be added to continuous integration server (CI) task list. This eases significantly finding of regressions, which may be introduced during the development or bug fix process.

As behaviour descriptions are technically same as unit tests then these descriptions are not discussed in the following sections.

4.1.4. Sequential Development Processes or Agile Practices

The most effective software development processes in embedded software development is the sequential processes where requirements are defined before coding such as Waterfall or V-model [40]. Both are sequential processes where typically the following phases are followed: the requirements are defined, then software architecture is designed, then the software is implemented, and after that follows a verification stage, and finally maintenance. In the V-model every step has own testing and verification phase. Above described processes allow using UML diagrams in specification or design stage, and in coding stage the automated tests can be used to test and lock down requirements.

Above listed sequential processes are well suited for embedded software development, especially in mission or safety critical cases, but for non-critical software, it is possible to use agile practices [19] including TDD and BDD. In

agile practices the most complicated task is the software functionality description and documentation writing. A large number of different software versions may add more maintenance work. However, while using sequential processes, full specification is required at the beginning of the project. This approach is also easier for a customer to understand, as it does not require too frequent communication with a developer and it has less different versions of programs like agile practices have. The major benefit for using agile practices is short development time, which is required to complete a software project.

In embedded software development, all above described models and agile practices can be used. If embedded system is not safety or mission critical, it is useful to consider the use of agile practices, as in mission or safety critical systems the software project should be described in high detail. Therefore, it is difficult to use only agile practices. In mission and safety critical software development, program behaviour should be modelled and maximally detailed description should be provided before the real program coding starts. This is typical for V and waterfall processes. Also, in order to avoid software regression bugs and decrease the overall development effort, the code should support fully automated testing (unit tests).

4.2. Programming Languages – C and C++

In every software project it is required to decide which programming language is going to be used. In this aspect, the embedded systems are not different. Although it is possible to use low level programming languages like assembler in embedded systems, but mostly the higher level languages like C or C++ are used. This section gives the reasons why C or C++ is used and outlines some problems that might rise from switching from C to C++.

4.2.1. Main Differences Between C and C++

The following section outlines the main differences of C and C++, which are significant factors in several cases when choosing a programming language for an embedded system.

In embedded systems development, the C programming language has been dominating for decades as the main programming language. Nevertheless in last decade the C++ has been gaining popularity as an alternative language to C. While C allows to write programs with low level hardware access, then C++ also allows to write low level hardware access programs but adds a possibility to write object-oriented programs as well. This makes it usable even in small embedded projects [86]. Back in 2005, the use of C++ compiler was relatively difficult due to the compiler bugs and inefficient code generation, then by 2016, there are no significant differences between C and C++ compilers. The C++ compiler from GCC 4.9 package, for example, is capable of producing programs with nearly the same memory requirements as C compiler. The only remaining issue with C++ compiler in embedded programs are the virtual functions – compilers and linkers put these tables to RAM (theoretically it is

possible to place these tables into program memory). On the other hand, most of the embedded programs are quite simple and therefore it is relatively easy to write programs without virtual functions. With the exception of virtual functions, programs written in C and C++ have similar memory requirements; the difference in most of the cases is less than five percent. To illustrate this, two simple examples in Listing 4.4 and 4.5 are given. A program size without any optimisation differs roughly 10%; a program that is written in C is 176 bytes and a program that is written in C++ is 196 bytes. The same programs that are compiled with maximum optimisation, which is typically used in embedded systems, have exactly the same size: 148 bytes.

```
 1:#include <stdio.h>
 2:
 3:/* Function that prints only "Hello World!!!" */
 4:static void hello (void)
 5:{
 6:    (void)printf ("Hello World!!!\n");
 7:}
 8:
 9:int main (void)
10:{
11:    /* Call to function that prints "Hello World!!!" */
12:    hello ();
13:
14:    return 0;
15:}
```

Listing 4.4: Simple "Hello World" in C – this program has only one function call (line 12).

```
 1:#include <stdio.h>
 2:
 3:/* Simple test class, which is compatible with similar program
 4: * written in C. */
 5:class HelloWorld
 6:{
 7:public:
 8:    /* Method that prints only "Hello World!!!" */
 9:    void hello ()
10:    {
11:         (void)printf ("Hello World!!!\n");
12:    }
13:};
```

```
14:
15:int main ()
16:{
17:    /* Declare and initalize test class. */
18:    HelloWorld hw;
19:
20:    /* Call to method that prints "Hello World!!!" */
21:    hw.hello ();
22:
23:    return 0;
24:}
```

Listing 4.5: Simple "Hello World" in C++; this program has one class and one method call (line 21). In this example, printf is used for compatibility with C program, in C++ mostly insertion operator "<<" are used.

As lot of development tools (most notably UML tools) and libraries (unit testing libraries) have support for C++, however, for C there is no support at all or it is limited, therefore it is reasonable to write most control logic in C++. It is reasonable to write program parts that are related to hardware in C, as this language has several useful additions for embedded programs that lack in C++. In several cases, it is not possible to write hardware related programs by using object oriented approach [82], therefore it makes no sense to use C++. For example in Listing 4.6 there are partial structure initialisations, all structure elements have value 0xFF, except the elements 1, 3, and 6, which have respective values of 1, 3, and 0.

```
 1:#define ARRAY_SIZE 8
 2:
 3:__extension__ uint8_t array_example[ARRAY_SIZE] =
 4:{
 5:    /* Set all array elements to default value 0xff. */
 6:    [0 ... (ARRAY_SIZE - 1)] = 0xFF,
 7:
 8:    /* Set second element to 0x01. */
 9:    [0x01] = 0x01,
10:
11:    /* Set fourth element to 0x03. */
12:    [0x03] = 0x03,
13:
14:    /* Set seventh element to 0x00. */
15:    [0x06] = 0x00
16:
17:    /* All reamining elements have value 0xff. */
18:};
```

Listing 4.6: Partial structure initialisation.

4.2.2. Using Different Programming Languages In One Software Project

In larger embedded programs software parts that have different responsibilities are separated from each other: hardware related code, networking code, control logic and other submodules are in separated source or packages. The main

reason for separation is the reuse of the code and to simplify testing. It also simplifies using different programming languages within same program.

Currently the best working practice for hardware or kernel related code is to use the C programming language. For the higher level code or code that is visible for the user such as program logic, the best solution is to use the C++ programming language. The main reason for doing so is that program parts that are written in C can be used with C and C++, but doing this in an opposite is usually much more difficult. This approach makes lower level program code little more flexible than it would be when written purely in C++. As C is a more mature programming language, in some circumstances these functions that are written in C, can be a little faster than C++ counterparts. In situations where it is needed to use C functions in programs that are written in C++, it is possible to write wrapper classes for C functions, which can be placed in C++ header files. The Listing 4.7 shows the wrapper class for C functions, lines 10 and 11 declare two functions: *function_1* and *function_2* which are both written in C; both functions are related and should be used as one C++ class. C++ specific code is between the lines 17 and 41; those lines define class *Functions* which have two methods, *cFunction1* (lines 27 to 30) and *cFunction2* (lines 33 to 36); both are inlined methods and contain only calls to corresponding C functions. When this file is included in C source code the C preprocessor is able to use code between the lines 10 and 11 and use this file as a regular C header file. When it is included to C++ source code the preprocessor and compiler is able to use all definitions, including C functions *function_1* and *function_2*.

```
 1:#ifndef __HEADER_H_
 2:#define __HEADER_H_
 3:
 4:#ifdef __cplusplus
 5:extern "C" {
 6:#endif
 7:
 8:/* Functions function_1 and function_2 are C functions that
 9: * are realated with hardware or kernel (for example). */
10:extern void function_1 (void);
11:extern void function_2 (void);
12:
13:#ifdef __cplusplus
14:}
15:#endif
16:
17:#ifdef __cplusplus
18:
19:namespace ClassNamespace
20:{
21:
22:/* Class that contains wrappers for C functions. */
23:class Functions
24:{
25:public:
26:    /* C++ wrapper for function_1 */
27:    void cFunction1 ()
28:    {
```

```
29:         function_1 ();
30:     }
31:
32:     /* C++ wrapper for function_2 */
33:     void cFunction2 ()
34:     {
35:         function_2 ();
36:     }
37:};
38:
39:}
40:
41:#endif
42:
43:#endif
```

Listing 4.7: Mixing C and C++.

Similar language mixing techniques work with other programming languages like Ada. Generally, mixing programming languages does not increase significantly program size, and therefore it is well suitable in embedded systems. This technique is useful in situations where some specific task is written in another language than the project's main programming language [82]. But when such language mixing involves assembler, one should encapsulate and isolate assembler blocks from other source codes (this is also a rule 2.1 of MISRA C [71] and MISRA C++ rule 7-4-1 to rule 7-4-3 [72]).

4.2.3. Alternative Approach for C++ Virtual Function Table

Virtual functions are functions in C++ whose behaviour can be overridden within an inheriting class by a function with the same signature. These functions allow reducing significantly relations between classes and writing less complex code, which is also much easier to be tested and maintained. The virtual functions use a special table: a virtual function table (V-table) that stores function calling addresses. The C language does not have such table but in several cases, such table would significantly reduce program's complexity. A similar approach is used in device drivers. In the following Listing 4.8 a simple example of the use of virtual functions for accessing an imaginary hardware is shown. A similar approach may be used to write hardware drivers in kernels.

```
 1:/* (Virtual)class that contains skeleton for implementation.
 2: * All implementations DevXFunction classes should extend to
 3: * this virtual class.*/
 4:class Functions
 5:{
 6:public:
 7:    /* Variable for holding some device related state. */
 8:    uint8_t state;
 9:
10:    /* Pure virtual class should have virtual destructor
11:     * also. */
12:    virtual ~Functions () {};
13:
14:    /* Do device specific initialisations. */
```

```
15:    virtual uint8_t init () = 0;
16:
17:    /* Operations that are common for all devices but each
18:     * device require different implementation. */
19:    virtual uint8_t doSomething () = 0;
20:};
21:
22:/* Functions for device nr. 1. This class implements all virtual
23: * functions that are defined in Function base class. */
24:class Dev1Functions : public Functions
25:{
26:public:
27:    /* Initialise device 1. */
28:    uint8_t init ()
29:    {
30:        /* ... */
31:        return 0;
32:    }
33:
34:    /* Do device specific operations. */
35:    uint8_t doSomething ()
36:    {
37:        /* ... */
38:        return 0;
39:    }
40:};
41:
42:/* Functions for device nr. 2. This class implements all virtual
43: * functions that are defined in Function base class. */
44:class Dev2Functions : public Functions
45:{
46:public:
47:    /* Initialise device 2. */
48:    uint8_t init ()
49:    {
50:        /* ... */
51:        return 0;
52:    }
53:
54:    /* Do device specific operations. */
55:    uint8_t doSomething ()
56:    {
57:        /* ... */
58:        return 0;
59:    }
60:};
61:
62:/* Load device handler class by given device number (dev_nr). */
63:Functions *loadFunctions (const uint8_t dev_nr)
64:{
65:    Functions *fn;
66:
67:    if (dev_nr == 1)
68:    {
69:        /* Device with index 1 should use device nr 2
70:         * functions. */
71:        fn = new Dev2Functions ();
72:    }
73:    else
```

```
 74:    {
 75:        /* All other devices uses device nr 1 functions. */
 76:        fn = new Dev1Functions ();
 77:    }
 78:
 79:    return fn;
 80:}
 81:
 82:int main ()
 83:{
 84:    /* Retrive device information from external function,
 85:     * (device information is stored in database, for
 86:     * example) */
 87:    const uint8_t dev_nr = getDevNr ();
 88:
 89:    /* Load device dependent functions, for that purpose is used
 90:     * factory function. */
 91:    Functions *dev = loadFunctions (dev_nr);
 92:
 93:    /* Initalize device and call device specific functions. */
 94:    dev->init ();
 95:    dev->state = 1;
 96:    dev->doSomething ();
 97:    dev->state = 2;
 98:
 99:    /* Finally release memory that holds device structure. */
100:    delete dev;
101:
102:    return 0;
103:}
```

Listing 4.8: Original C++ code.

In the example above, in the lines 82 to 103 is the main function, which retrieves device identification by calling *getDevNr* (line 87) function. Identification number is passed to factory function *loadFunctions* (call on line 91 and function implementation is on line 63 to 80), which returns corresponding class: *Dev1Functions* or *Dev2Functions*; the rest of main function uses one of the device classes. Classes *Dev1Functions* (line 24 to 40) and *Dev2Functions* (from the line 44 to 60) are derived from abstract base class *Functions* (from the line 4 to 20). This base class has two abstract methods, *init* and *doSomething*, and it also has one variable name *state*.

In the C language it is possible to create a program with the same functionality by using function pointers and structures: while C does not have C++ like V-tables, it allows to write similar tables. As function structures do not change during a program execution, it is possible to place this constant table to program memory. The following example in the Listing 4.9 illustrates same program as in the Listing 4.8, but this program is written entirely in C, and uses function pointers.

```
 1:/* Structure that hold device state and pointers to functions,
 2: * init and do_something. */
 3:struct _functions_s
 4:{
 5:    /* Variable for holding some device related state. */
 6:    uint8_t state;
 7:
 8:    /* Do device specific initialisations. */
 9:    int8_t (* init) (void);
10:
11:    /* Operations that are common for all devices, but each
12:     * device require different implementation. */
13:    int8_t (* do_something) (void);
14:};
15:
16:/* Define variable that contains above defined function
17: * structure. */
18:typedef struct _functions_s functions_s;
19:
20:/* Initialise device 1. */
21:static int8_t init_dev_1 (void)
22:{
23:    /* ... */
24:    return 0;
25:}
26:
27:/* Initialise device 2. */
28:static int8_t init_dev_2 (void)
29:{
30:    /* ... */
31:    return 0;
32:}
33:
34:/* Do device specific operations. */
35:static int8_t do_something_dev_1 (void)
36:{
37:    /* ... */
38:    return 0;
39:}
40:
41:/* Do device specific operations. */
42:static int8_t do_something_dev_2 (void)
43:{
44:    /* ... */
45:    return 0;
46:}
47:
48:/* Load device specific function addresses to variable dest and
49: * initialise state variable. Device is selected by variable
50: * dev_nr. */
51:void load_functions (void *dest, const uint8_t dev_nr)
52:{
53:    /* Following two structures can be placeed into program
54:     * memory. */
55:    static const functions_s dev_1_functions =
56:    {
57:        0x00,               /* state */
58:        init_dev_1,         /* init */
59:        do_something_dev_1  /* do_something */
```

```
 60:    };
 61:
 62:    static const functions_s dev_2_functions =
 63:    {
 64:        0x00,               /* state */
 65:        init_dev_2,         /* init */
 66:        do_something_dev_2  /* do_something */
 67:    };
 68:
 69:    if (dev_nr == 1)
 70:    {
 71:       /* Device with index 1 should use device nr 2
 72:        * functions. */
 73:        (void)memcpy (dest, &dev_2_functions,
 74:                      sizeof (functions_s));
 75:    }
 76:    else
 77:    {
 78:        /* All other devices uses device nr 1 functions. */
 79:        (void)memcpy (dest, &dev_1_functions,
 80:                      sizeof (functions_s));
 81:    }
 82:}
 83:
 84:int main (void)
 85:{
 86:    functions_s dev;
 87:
 88:    /* Retrive device information from external function,
 89:     * (device information is stored in database, for
 90:     * example). */
 91:    const uint8_t dev_nr = get_dev_nr ();
 92:
 93:    /* Load device dependent functions, for that purpose is used
 94:     * factory function. */
 95:    load_functions (&dev, dev_nr);
 96:
 97:    /* Initalize device and call device specific functions. */
 98:    dev.init ();
 99:    dev.state = 1;
100:    dev.do_something ();
101:    dev.state = 2;
102:
103:    /* Unlike C++ example we don not need to free the device
104:     * structure, it is placed to the heap and destroyed after
105:     * function return. */
106:
107:    return 0;
108:}
```

Listing 4.9: Same program in C.

As in the C++ example, the lines 84 to 108 show the program main function, which retrieves device identification by calling *get_dev_nr* (line 91) function. This identification number is passed in the line 95 to function *load_functions* (it is implemented in the lines 51 to 82), which will copy corresponding function structures (*dev_1_functions* or *dev_2_functions* structures) from constant memory area into *dev* structure variable. As structures *dev_1_functions* and

dev_2_functions are constant, it is advisable to place these structures in program memory, this will save quite significant amount of RAM. The rest of main function, the lines 98 to 101 use one of the device structures without knowing which *dev* functions structure it uses.

The program examples described above have considerably different sizes – example in C language is 112 bytes long and example in C++ is 260 bytes long. Both lengths are taken before linker. Such variability of program size is caused by using different functions or operators in the above examples. However, considering that many similar programs are used on larger devices it is not essential to have minimal memory footprint. If the size of the memory footprint is important, it is more convenient to use the C language anyway.

Similar methods, which are presented in Listing 4.9, are used in kernels for calling hardware dependent program parts; this allows effectively to hide hardware related code from higher level programs. Although the C++ code has a similar functionality as the C code, it is not widely used in kernels. The main reason for this is historical; older C++ compilers did not create same effective code as C compilers and it did not have any other significant advantage over C compilers. While new programs, which are written from scratch and use new C++ compiler, is reasonable to write completely in C++. While both examples increase code reuse and simplify programs, it is not advisable to use above described methods intensively in embedded systems as both variants can consume significant amount of RAM.

4.3. Program Structures and Improvements on Testing

The following sections describe different program structures and required improvements that are needed when using similar testing methods in embedded systems than are used in desktop computers. Described methods significantly simplify automated tests in embedded systems and are grown out from research such as developing automated testing frameworks for regression and hardware tests by using FOTA [65].

4.3.1. Super Loop Programs

Super-loop programs (sometimes also called main-loop programs) are programs where all data processing is done in one loop, which is typically placed into main function. In these programs the input data is read by using interrupts or by polling inputs. The main difference between super-loop programs and kernel is that super-loop programs do not use scheduler and it is typically designed to perform only one task, while kernel has scheduler that may have several separated tasks. The Listing 4.10 shows a simple super-loop example: the lines 6 and 9 show the hardware initialisation – PORTB pin 0 is set to output and all PORTC pins to inputs. The lines 12 to 27 show an infinite loop which reads input status from PORTC input 0 (line 15) and switches output pin according to input value. Should the input value be logical 1, the output 0 on PORTB is set to

logical 0 (line 19), and should the input value be logical 0, the output 0 on PORTB is set to logical 1 (line 25).

```
 1:#include <avr/io.h>
 2:
 3:int main (void)
 4:{
 5:    /* Initialize port B pin 0 to output. */
 6:    DDRB = (1 << PB0);
 7:
 8:    /* Set all port C pins to inputs. */
 9:    DDRC = 0x00;
10:
11:    /* Enter to infinite loop. */
12:    while (1)
13:    {
14:        /* Test port C pin 0. */
15:        if (PINC & (1 << PC0))
16:        {
17:            /* If port C pin 0 has logial 1, then set port B
18:             * pin 0 to logial 0. */
19:            DDRB &= ~(1 << PB0);
20:        }
21:        else
22:        {
23:            /* If port C pin 0 has logial 0, then set port B
24:             * pin 0 to logial 0. */
25:            DDRB |= (1 << PB0);
26:        }
27:    }
28:}
```

Listing 4.10: Super loop program example.

Typically the writing and testing of a super-loop program is rather simple. During testing, mostly are manipulated by the inputs and observed the reactions on the outputs. Same is also possible by using ICE or emulator. The main drawback of this kind of programs is that the program depends mostly on direct access to registers, which makes it difficult to create a portable program. Due to manual testing, maintenance is also complicated: every change in the source code requires a lot of manual testing.

Due to the high amount of manual testing the super-loop programs are usable in smaller projects. Typically such programs have less functionality and are shorter than multithreaded programs; usually less than 5000 lines of code or 50 function points. It is reasonable to use super-loop programs in commercial products and in projects where specification is available before coding. For example in simpler sensor and actuator systems. This program type is widely used in hobby projects.

Automatically Testable Super Loop Programs

Larger programs that are written for PC and are automatically testable have minimal amount of relations with other software modules [58, 80]. The best approach for testing super-loop programs is to move body of the main function

to separate function, which can be tested independently. Embedded programs that interact directly with hardware should have minimal or even no direct hardware relations. This is as a prerequisite for automated testing.

4.3.2. Minimising Relations Between Submodules

In embedded software development, it is possible to use same methods to decrease software cross dependencies as it are used in desktop computer software development – every software module should have only one responsibility. To illustrate this, an example of a program in Listing 4.11 is given. In this example tasks are separated from different functions:

```
 1:#include <stdint.h>
 2:
 3:/* Hardware initialization. */
 4:static inline void io_init (void)
 5:{
 6:    /* HW specific operations. */
 7:}
 8:
 9:/* AD converter initialization. */
10:static inline void adc_init (void)
11:{
12:    /* AD converter (HW) specific operations. */
13:}
14:
15:/* Read one 16 bit sample from AD converter. */
16:static inline uint16_t read_adc (void)
17:{
18:    /* HW specific operations. */
19:}
20:
21:/* Filter (IIR) for 'smoothing' input data. 'last_value' is last
22: * output value (from this function call). 'new_value' is input
23: * data from AD converter. This function retuns an filtered ADC
24: * value. */
25:static inline
26:uint16_t iir_filter (const uint16_t last_value,
27:                     const uint16_t new_value)
28:{
29:    /* IIR filter code. */
30:}
31:
32:/* Change microcontroller output value accordingly to parameter
33: * 'v_val'. */
34:static inline void output_ctrl (const uint16_t v_val)
35:{
36:    /* HW specific operations. */
37:}
38:
39:int main (void)
40:{
41:    /* Raw value from AD converter. */
42:    uint16_t ad_val;
43:
44:    /* Value that represents voltage which is based on filtered
45:     * AD values. */
```

```
46:    uint16_t voltage;
47:
48:    /* Do hardware initilaization. */
49:    io_init ();
50:    adc_init ();
51:
52:    while (1)
53:    {
54:        adc_val = read_adc ();
55:        voltage = iir_filter (voltage, ad_val);
56:        output_ctrl (voltage);
57:    }
58:}
```

Listing 4.11: AD converter, IIR filter and output control example.

In Listing 4.11, all AD converter functions are isolated (functions *adc_init* and *read_adc*, lines 10 to 19) from filters (function *iir_filter*, the lines 25 to 30) and from the output control functions (function *output_ctrl*, the lines 34 to 37). The lines 39 to 58 show the main function of this program. While in the lines 49 to 50 the hardware initialisations are called, rest of the program is the main loop (lines 52 to 57). In main loop, first the AD converter read function is called (line 54), then AD converter value is added to IIR filter (line 55) and finally filtered voltage value is sent to output function (line 56). This example shows that all tasks are partitioned to different functions, which is quite elementary in most of software projects; however, this is widely ignored in embedded systems. The main reason why the isolation rule is ignored is due to the usage of short calls inside hardware specific functions; all calls to hardware registers are typically one line long, which is relatively easy to integrate into calling function. The major drawback of previous listing is that all hardware dependent functions are in the compilation unit as hardware-independent code, and this does not allow to write unit tests.

The following code listings are modified versions of the code from Listing 4.11. These listings have separated code for hardware dependent functions, and also separated code for hardware-independent functions; rest of the code is the same. The Listing 4.12 is a header file, which contains all hardware dependent function declarations: IO and AD initialisation, AD read function and output control function. The Listing 4.13 contains hardware specific function implementations for functions that are declared in the Listing 4.12. The Listing 4.14 contains the program's main function, which uses functions that are declared in the *hw.h* file.

```
1:/* Header file for hardware specific functions. */
2:
3:#ifndef __HW_H_
4:#define __HW_H_
5:
6:#include <stdint.h>
7:
8:/* Hardware initialisation function. */
9:extern void io_init (void);
```

```
10:
11:/* AD converter initialisation function. */
12:extern void adc_init (void);
13:
14:/* Read one 16 bit sample from AD converter. */
15:extern uint16_t read_adc (void);
16:
17:/* Change microcontroller output value accordingly to parameter
18: * 'v_val'. */
19:extern void output_ctrl (const uint16_t v_val);
20:
21:#endif
```

Listing 4.12: Header file for hardware specific functions "hw.h".

```
 1:/* Implementation of hardware specific functions. */
 2:
 3:#include <stdint.h>
 4:#include "hw.h"
 5:
 6:/* Hardware initialization function. */
 7:void io_init (void)
 8:{
 9:    /* HW specific operations. */
10:}
11:
12:/* AD converter initialization function. */
13:void adc_init (void)
14:{
15:    /* HW specific operations. */
16:}
17:
18:/* Read one 16 bit sample from AD converter. */
19:uint16_t read_adc (void)
20:{
21:    /* HW specific operations. */
22:}
23:
24:/* Change microcontroller output value accordingly to parameter
25: * 'v_val'. */
26:void output_ctrl (const uint16_t v_val)
27:{
28:    /* HW specific operations. */
29:}
```

Listing 4.13: File for hardware specific code, "hw.c", this file has hardware specific code.

```
 1:/* Rest of the program. */
 2:
 3:#include <stdint.h>
 4:#include "hw.h"
 5:
 6:/* Filter (IIR) for 'smoothing' input data. 'last_value' is last
 7: * output value (from this function call). 'new_value' is input
 8: * data from AD converter. This function retuns an filtered ADC
 9: * value. */
10:static inline
11:uint16_t iir_filter (const uint16_t last_value,
```

```
12:                     const uint16_t new_value)
13:{
14:    /* IIR filter code. */
15:}
16:
17:int main (void)
18:{
19:    /* Raw value from AD converter. */
20:    uint16_t ad_val;
21:
22:    /* Value that represents voltage which is based on filtered
23:     * AD values. */
24:    uint16_t voltage;
25:
26:    /* Do hardware initilaization. */
27:    io_init ();
28:    adc_init ();
29:
30:    while (1)
31:    {
32:        adc_val = read_adc ();
33:        voltage = iir_filter (voltage, ad_val);
34:        output_ctrl (voltage);
35:    }
36:}
```

Listing 4.14: The Program's main file ("main.c") which does not have any hardware dependent code.

In the listing above, all relations between the main program and the hardware are now separated; this allows to use unit tests on different hardware by using mocked hardware. Mocks are created in separate source files and are not listed in this document. Testing implementation does not have any hardware dependencies but instead it has only logging and other functions, which are needed for testing.

Similar functionality separation approach is stated by several different authors, but none of them mentioned one significant side effect. In most cases, functionality separation decreases program execution speed and increases memory footprint, mostly the stack size. Both are caused by function calling mechanisms and while they are not significant for PC programs, they are significant for smaller microcontrollers. However, in order to reduce these effects, all hardware dependent functions should be added into the same compilation unit (basically the same code as in the Listing 4.11). There is one possibility to achieve this: define all hardware dependent functions as 'static inline' and place them into separated header file (the Listing 4.15). 'static inline' function code is inserted into at the place of each function call and consequently, inlined functions save the overhead of function call but increase size of the program memory image. Increasing memory image is typically not as significant as high RAM usage.

In the following header file is the modification from the Listing 4.12; in this file all hardware specific code is defined as *static inline*. This modification allows to compile all functions as inline functions, and *hw.c* file is redundant.

```
 1:/* Implementation of hardware specific functions. */
 2:
 3:#ifndef __HW_H_
 4:#define __HW_H_
 5:
 6:#include <stdint.h>
 7:
 8:/* Hardware IO initialization function. */
 9:static inline void io_init (void)
10:{
11:    /* HW specific operations. */
12:}
13:
14:/* AD converter initialization function. */
15:static inline void adc_init (void)
16:{
17:    /* HW specific operations. */
18:}
19:
20:/* Read one 16 bit sample from AD converter. */
21:static inline uint16_t read_adc (void)
22:{
23:    /* HW specific operations. */
24:}
25:
26:/* Change microcontroller output value accordingly to parameter
27: * 'v_val'. */
28:static inline void output_ctrl (const uint16_t v_val)
29:{
30:    /* HW specific operations. */
31:}
32:
33:#endif
```

Listing 4.15: “hw.h” with inline functions.

Above described separation has also one downside. While adding functions through headers, such functions are also added that are not required in this compilation unit. This has slight incompatibility with MISRA C (rule 8.5 [71]) and C++ (rule 0-1-10 [72]) rules.

4.3.3. Stateless Functions

The main prerequisite for unit tests is that tested program should be cross compileable between different architectures, and have hardware dependencies. To create such program, the most effective way is to follow guidelines with described methods to achieve high portability. These guidelines are typically MISRA C, MISRA C++, JSF and strictly following C99 or C++11 standards adds some portability. Second important aspect is that program does not store its states internally; it should be stateless [80]. But in embedded software such stateful functions and variables are used quite widely; mostly these functions

are related with EEPROM. Other hardware dependent code parts like interrupt processing can also be considered as stateful. Stateless functions play also important role in unit testing. A testing framework may not always guarantee the same test order, therefore, with stateful functions when tests are not isolated, sequence of the tests should be taken into account. This section describes how to separate stateful code (hardware dependent) from stateless code (program logic part). In the Listing 4.16, is a simplified example of a typical stateful function. In this example is a function that increases the variable *i* and stores the result in the same variable (the line 7). The size of the program in this example is 20 bytes, in this and the following examples are used maximum optimisation.

```
 1:void fn (void)
 2:{
 3:    /* Variable 'i' is stored to constant location in RAM,
 4:     * consequently all calls to function fn use "saved"
 5:     * variable 'i'. */
 6:    static unsigned int i;
 7:    i++;
 8:
 9:    /* Do something with 'i'. */
10:}
```

Listing 4.16: Function with static variable.

It is possible to rewrite a stateful function *fn* in several different ways without using internal static variable. In the Listing 4.17 (example in C language), from the lines 5 to 15 is a function that increases static variable from static memory area. This variable is passed on function call in the line 19. This example requires 20 bytes of program memory.

```
 1:/* This variable is placed to fixed address in microcontroller's
 2: * memory. */
 3:static unsigned int value_x;
 4:
 5:void fn (unsigned int *p)
 6:{
 7:    /* Increment input value by 1, and store it to temporary
 8:     * variable. */
 9:    unsigned int i = *p + 1;
10:
11:    /* Do something with i. */
12:
13:    /* Store temporary variable to fixed memory location. */
14:    *p = i;
15:}
16:
17:/* This function call take memory location as parameter and
18: * stores it's result to same location. */
19:fn (&value_x); /* Function call */
```

Listing 4.17: Accessing a static variable through pointer.

In the Listing 4.18 (example in C language), from the lines 7 to 16 is a function that modifies input parameter *val* and returns the modified result. In the line 21 is a call to the function *fn*; within this call a value is copied from a static

variable when the function returns the stored result back to the same variable. When this is a static function, then the size of this example is four bytes, but when the same function would be called several times or this function is non static function, then its size would be probably 20 bytes.

```
 1:/* This variable is placed to fixed address in microcontroller's
 2: * memory. */
 3:static unsigned int value_x;
 4:
 5:/* This function only uses input value 'val' and it returns
 6: * modified 'val' value, which can used in other places. */
 7:unsigned int fn (const unsigned int val)
 8:{
 9:    /* Increment input value by 1, and store it to temporary
10:     * variable. */
11:    unsigned int i = val + 1;
12:
13:    /* Do something with i. */
14:
15:    return i;
16:}
17:
18:/* Call function fn. This function take value_x from static
19: * memory location and stores function return value to same
20: * locaion. */
21:value_x = fn (value_x); /* Function call */
```

Listing 4.18: Parameter passing example.

In the Listing 4.19 (example in C++), from the lines 6 to 16 is the function that change a value that is passed by reference. The function call with reference passing is in the line 20. This example is technically similar to the example in Listing 4.17, except that the reference variable address cannot be changed. As this example is technically the same as the example in Listing 4.17 then the program size is 20 bytes.

```
 1:// This variable is placed to fixed address in microcontroller's
 2:// memory.
 3:static unsigned int value_x;
 4:
 5:// This function take input reference and modify its value.
 6:void fn (unsigned int &p)
 7:{
 8:    // Increment input value by 1, and store it to temporary
 9:    // variable.
10:    unsigned int i = p + 1;
11:
12:    // Do something with i.
13:
14:    // Store temporary variable to fixed memory location.
15:    p = i;
16:}
```

```
17:
18:// This function call take memory location as parameter
19:// (reference) and stores it's result to same location.
20:fn (value_x); // Function call
```

Listing 4.19: C++ specific example – changing reference value.

It is possible to write similar functions for EEPROM reading and writing. In the Listing 4.20 is an example function of reading data from EEPROM area (the line 5), modifying read value (the line 7) and storing the result back to the same EEPROM memory location (the line 10). This and all of the following functions are directly dependent on the size of the integer types that used for specific microcontroller and functions that access to memory. Therefore it is difficult to know the exact program size, but the example functions similar to those of the preceding examples have similar sizes.

```
 1:/* This function stores internal states to EEPROM. */
 2:void fn (void)
 3:{
 4:    /* Read value from some predefined EEPROM location. */
 5:    unsigned int i = read_eeprom ();
 6:
 7:    /* Do something with i. */
 8:
 9:    /* Write modified value back to EEPROM. */
10:    write_eeprom (i);
11:}
12:
13:fn (); /* Function call */
```

Listing 4.20: EEPROM read-modify-write function example.

It is possible to replace above used hard-coded read and write functions (*read_eeprom* and *write_eeprom*) using two following methods: conditional compilation, and pass pointer to read and write functions.

The program code is the same as in Listing 4.20 when using conditional compilation, but every architecture and testing implementation has its own implementation of *read_eeprom* and *write_eeprom* functions. sing conditional compilation is easier to implement - it consumes less memory and processor resource, but testing and maintenance is more complicated. The major problem is function implementations that are required for testing. These implementations may require relatively complicated code for hardware emulation, and this code should include different input and output functions for testing purposes. Conditional compilation is preferred in smaller systems where relatively simple hardware related functions are used, or in cases where program execution speed or low memory consumption is essential. This approach is most suitable in interrupt handlers.

It is also possible to use function pointer to isolate stateful code. At first, read and write function types are defined (Listing 4.21, the lines 6 and 10):

```
 1:#ifndef __HEADERS_H_
 2:#define __HEADERS_H_ 1
 3:
 4:/* Type for read function. This function does not take any
 5: * parameters, but returns unsigned integer. */
 6:typedef unsigned int (*read_fn_t) (void);
 7:
 8:/* Type for write function. This function take unsigned int as
 9: * input parameter and does not return any value. */
10:typedef void (*write_fn_t) (unsigned int i);
11:
12:/* Function that read unsigned int from predefined memory
13: * location. */
14:extern unsigned int eeprom_read (void);
15:
16:/* Function that write unsigned int to predefined memory
17: * location. */
18:extern void eeprom_write (const unsigned int v);
19:
20:#endif
```

Listing 4.21: Type definitions for the read and write functions.

Then, the *read_fn* and *write_fn* functions are implemented in a separate source file:

```
 1:#include "headers.h"
 2:
 3:/* Function that read unsigned int from predefined memory
 4: * location. */
 5:unsigned int eeprom_read (void)
 6:{
 7:    /* Read one byte from predefined EEPROM location. */
 8:}
 9:
10:/* Function that write unsigned int to predefined memory
11: * location. */
12:void eeprom_write (const unsigned int val)
13:{
14:    /* Write one byte to predefined EEPROM location. */
15:}
```

Listing 4.22: Read and write function implementations.

Finally, in the Listing 4.23 is a function that uses above described read and write functions and calls to a function that is responsible for changing EEPROM contents. In the lines 7 to 19 is a function that uses above defined read and write functions; in the line 23 is a function that is called when it is needed to modify EEPROM contents.

```
 1:#include "headers.h"
 2:
 3:/* Function that read input value by using function that is
 4: * passed by read_fn parameter, modify input value and store
 5: * result by using function that is passed by write_fn
 6: * parameter. */
 7:static void fn (const read_fn_t read_fn,
 8:                const write_fn_t write_fn)
 9:{
10:    /* Read input value by using function that is passed by
11:     * read_fn parameter. */
12:    unsigned int i = read_fn ();
13:
14:    /* Do something with i. */
15:
16:    /* Write function result by using function that is passed by
17:     * write_fn parameter. */
18:    write_fn (i);
19:}
20:
21:/* Call function fn and use eeprom_read and eeprom_write
22: * functions for input data reading and save. */
23:fn (eeprom_read, eeprom_write); /* Function call */
```

Listing 4.23: Calls to read and write functions and final call modifying function.

The modified version allow to write tests quite efficiently; it only requires different implementation of read and write functions (from the Listing 4.22) on different architectures but rest of the code remains the same. For testing purposes, it is possible to inject different hardware mocking functions [80] – this was not possible using conditional compilation. The downside of the presented method is that it uses function pointers and it requires more microcontroller memory and CPU resources – on AVR microcontrollers it uses at least 4 bytes of stack, and every indirect call with additional memory load instructions requires several CPU cycles, more than a regular function call. However, this implementation is preferred for use in such places that have a lot of relations to different software modules, in non-time-critical sections, and in functions that contain control logic. Both described approaches are also usable in places where it is needed to test interrupt handling functions, but it is preferred to use conditional compilation instead.

Above described approaches allow to use unit tests and behaviour descriptions in embedded programs. It allows to use methods that are heavily used in projects that use agile practices. Described program partitioning and code refactoring, which is presented in above examples, is not strictly related by automated tests, but it allows to create programs that have high maintainability and testability. The main downsides of the presented methods are typically increased CPU or memory resource consumption.

4.3.4. Unit Tests on Target Hardware

It is also possible to run unit tests on target hardware. The main advantage of this is the absence of modules that emulate hardware. Typically, writing unit

tests for target hardware is similar to writing tests for different hardware. The only difference is that tests that run on target hardware should be as small as possible and not consume significant microcontroller resources. While testing on target hardware, tests should send results outside the testing environment, i.e., the development computer.

The main usage of this method is in detecting such program bugs that are impossible to detect in a different environment. These tests may be integration tests as well where a test scenario is previously broadly specified. Generally, these tests are more like hardware self-tests than normal unit tests. Unit testing on target hardware is not widely spread and currently only one framework is known that has support for it – "BSP430 Board Support Package for MSP430 Microcontrollers" [12]. In the context of this thesis, this testing method has been used several times with FOTA [65] enabled telematics modules; however, it is not very convenient in a regular embedded software development process.

4.3.5. Debugging and Testing

While it is possible to use unit tests in embedded software development, it is not possible to debug all embedded software by using only unit tests. Even in programs where unit testing is heavily used, several different hardware specific debugging and testing methods are employed. Mostly, OCD is used, but in smaller extent other manual testing tools like simulators and emulators are used.

For smaller microcontroller software or bootloader development, two different types of debugging approaches are mostly used. Firstly, manipulation of hardware inputs and then waiting for some state to change, and observing the output reaction. Or secondly, using OCD. Both methods require small amount of memory and CPU resource, they can also be considered as manual debugging methods and need the direct interaction of a developer. To illustrate one manual debugging approach, an example code is given in the Listing 4.24. This example involves tracking output change when some internal condition of a program changes. Similar approach is described in at least one of the embedded software related books [28]. In this example, a microcontroller sets its output PB0 to logical one when input parameter has non null value (lines 6 to 13), and sets to logical zero when function input value has null value (lines 15 to 21). As the output value changing rate depends on the function calling rate and function execution time, then with the lower changing rate it is possible to visually monitor the output states by using an LED, which is connected to microcontroller output pin. However, for higher calling rates an oscilloscope or a logic analyser is required to monitor the microcontroller output.

```
1:#define MON_PORT PORTB
2:#define MON_PIN PB0
3:
4:void fn_1 (const uint8_t val1)
5:{
6:    if (val1)
7:    {
```

```
 8:        /* Some code here. */
 9:
10:        /* After executing this brach set monitor pin to
11:         * high level (logical one) */
12:        MON_PORT |= (1 << MON_PIN);
13:    }
14:    else
15:    {
16:        /* Some code here. */
17:
18:        /* After executing this brach set monitor pin to
19:         * low level (logical zero) */
20:        MON_PORT &= ~(1 << MON_PIN);
21:    }
22:}
```

Listing 4.24: Monitor example.

Another option to the above described approach is to use a serial port for debugging data output. When using a serial port, two aspects should be taken into account: a program should not change states faster than the hardware is capable of transmitting, and the receiving side should be able to receive and process data fast enough so that there would be no loss of data.

It is also possible to use simulators and OCD. A simulator allows to test programs without using real hardware, which is a clear advantage when using such microcontrollers where it is technically complicated to upload new software. To overcome data inputting and outputting problems, OCD is quite widely used, which is typically connected to microcontrollers by using a JTAG interface. But OCD has also a noticeable drawback: it may alter the program's real-time and asynchronous hardware behaviour [8]. This makes it difficult to use OCD with larger programs that use a kernel and variety of different hardware. Nevertheless, for smaller programs without kernel or sophisticated interrupt system, which generates asynchronous interrupts, there are no significant problems with OCD.

As every above mentioned method has its own weaknesses and strengths, all methods are used in different places. Debugging methods that involve lot of program uploading, like the method that involves output change monitoring, are not usable in large systems – program loading makes this method too time consuming. Using a simulator is limited by the number of input states. The main problem with OCD is asynchronous hardware. All the listed methods have one thing in common: they all have a lot of operations that should be carried out by a developer, which in turn increases software development time and are error prone. With methods described, it is difficult to test programs that are larger than 50 function points (5000 lines of code in C), even when these programs are written in a very modular way. Therefore, it is reasonable to use described testing method only when no other testing method is available. However, if it is possible to use unit tests, one should write all programs to use unit tests.

4.4. Multithreaded Programs on Embedded Systems

Many embedded systems are designed to execute only one task, but also exists also many embedded systems that have concurrently several tasks. In systems that are designed for multiple tasks, it is much more complicated to guarantee correct hardware access and CPU resource sharing. In following sections are described some problems that show up in multithreaded embedded software, also are given some solutions for these situations. Described solutions support wave height measurements in TM side [69] – for wave height measurements, it is needed that one process is active for a long time and that other processes do not change its state for the same amount of time. To use such long run process on TM, the watchdog should not restart microcontroller by only watching non active threads.

4.4.1. Sharing Processor Resource Between Tasks, Schedulers

Real-time programs typically have different tasks that are separated but run as parallel processes; such tasks are called threads. Every thread is like a single super-loop program – it has its own memory area and hardware access. The only difference in a real super-loop program is that a thread may interact with other threads. It can send data from one thread to another, but also every thread is able to interfere with other threads, or may change the state of a shared hardware resource.

Threads and multitasking depend on one program part – the scheduler. This program part is responsible for sharing CPU time between different threads. Small embedded systems have typically two different types of schedulers – preemptive and cooperative. In most cases, only one type of a scheduler is used, and this should be selected before program compilation; for example, FreeRTOS [85] has such option. With powerful CPU, large memory and not very critical timings (i.e., not hard real-time system), there are no significant differences between preemptive and cooperative kernels. However, in smaller microcontrollers it is more difficult to use preemptive kernel. The main difference in scheduler is the task changing mechanism: a preemptive scheduler changes tasks automatically after predefined period, a cooperative scheduler changes tasks by user command (in most cases, a task change command is hidden to developer; these commands are automatically called by other functions like *printf*, which eventually waits for hardware). Due to the differences in task changing mechanism, preemptive kernels are little bit more fault tolerant than cooperative kernels and require more resources. Cooperative schedulers, however, need little more testing, take less microcontroller resources and are simpler to implement.

A preemptive kernel has two major drawbacks. Firstly, when a kernel is configured to change tasks too frequently, it may take significant amount of CPU resources and therefore it may take longer to accomplish some task than with less frequent task switching; although it may decrease response times. Secondly, the kernel needs to have some periodical signal to trigger the task

change. Typically, in embedded systems a hardware timer is used for that purpose. Therefore, it requires one hardware timer and one interrupt[10].

But a cooperative kernel has the ability to lock the whole CPU for infinite time, or trigger a watchdog in a normal operation. While developing programs with cooperative kernel one should take into account that one single task should not consume all the CPU's resources. Consuming all CPU resources will lock CPU to one task and other tasks cannot be run; this usually freezes the whole system. This drawback is the most significant shortcoming, and therefore, this kernel type is not preferred on desktop computers.

While the desktop computer operational systems allows a preemptive kernel to avoid situations where one faulty program freezes all other programs, then in embedded systems this behaviour may not be the optimal solution. It is preferred in several systems, a full system freeze and reboot from watchdog when one task locks up. While using a preemptive kernel without complex thread monitoring, the rest of a system may be operational for long time after one thread freezes. With cooperative kernel all tasks are blocked and a system wide watchdog triggers system reboot. In this case, a watchdog may have quite simple implementation, and watchdog hardware can be reset from an *idle* task.

4.4.2. Multitasking Programs and Watchdog

Typically, most embedded systems have a watchdog circuit, which is used to prevent situations where the system may stay in one state for indefinite time. Using a watchdog in super-loop programs is relatively easy – it should be reset after all tasks in a loop are completed; the only concern is that these tasks should not take too long. Using a watchdog timer in multithreaded programs is much more complicated – it is not possible to monitor several threads by simply using a single watchdog. In a multithreaded system only one process is allowed to reset the watchdog timer; the decision about watchdog resetting should be based on results of monitoring the state of all threads. Depending on scheduler type, it is possible to use different methods for resetting the watchdog. With preemptive and cooperative schedulers, it is possible to use watchdog resetting by monitoring all threads in parallel, or by methods where an unlock token is passed from thread to thread. With a cooperative scheduler it is possible to use above mentioned methods and also to reset a watchdog from an *idle* thread.

For resetting a watchdog by monitoring the threads running in parallel, the scheduler and some other periodical routines (i.e. timer interrupt[11]) should check that the last thread or group of threads have modified some of its parameters (Figure 4.1), and when no changes are detected then the watchdog resets; an example program is given in the Listing 4.25. A parameter which is monitored may be some thread related parameter like a stack pointer or program counter or

[10] In some cases it is possible to use real-time clock for this purpose.

[11] This method is preferred when it is not possible to use a scheduler for thread monitoring.

similar parameter, but this parameter should change after every context switch. A similar approach is used in FunkOS [27]; it uses time-out for every task and all tasks are monitored in a separate process, but this approach requires that all tasks have predefined periods for activity.

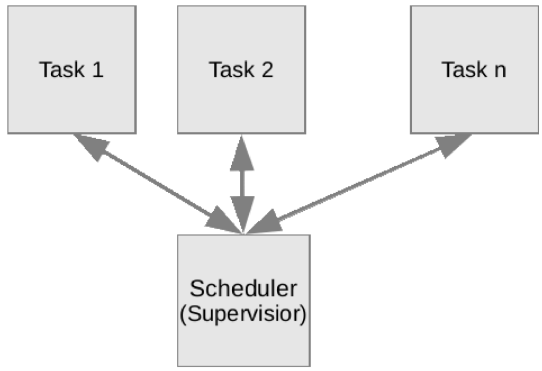

Figure 4.1: Parallel watchdog

The following program illustrates above described watchdog resetting method.

```
 1:#define TH1_COMPLETED 0
 2:#define TH2_COMPLETED 1
 3:#define TH3_COMPLETED 2
 4:#define ALL_TREHADS ((1 << TH1_COMPLETED) |
 5:                     (1 << TH2_COMPLETED) |
 6:                     (1 << TH3_COMPLETED))
 7:
 8:/* Variable that hold lock for watchdog access. */
 9:uint8_t wdt_lock = 0;
10:
11:THREAD1
12:{
13:    while (1)
14:    {
15:        /* Task which completion is monitored by wdt_lock. Flag
16:         * TH1_COMPLETED set only when task is successfully
17:         * completed. */
18:        wdt_lock |= (1 << TH1_COMPLETED);
19:    }
20:}
21:
22:THREAD2
23:{
24:    while (1)
25:    {
26:        /* Task which completion is monitored by wdt_lock. Flag
27:         * TH2_COMPLETED set only when task is successfully
28:         * completed. */
29:        wdt_lock |= (1 << TH2_COMPLETED);
30:    }
31:}
32:
```

```
33:THREAD3
34:{
35:    while (1)
36:    {
37:        /* Task which completion is monitored by wdt_lock. Flag
38:         * TH2_COMPLETED set only when task is successfully
39:         * completed. */
40:        wdt_lock |= (1 << TH3_COMPLETED);
41:    }
42:}
43:
44:WDT_RESET_THREAD
45:{
46:    sleep (MONITOR_SLEEP_TIME);
47:
48:    /* Test that all tasks have been set completed bit. Watchdog
49:     * timer is resetted only when this test returns true. */
50:    if (wdt_lock == ALL_TREHADS)
51:    {
52:        /* All tasks have been completed, reset WDT */
53:        wdt_reset ();
54:
55:        /* Clear also wdt_lock, now is possbile to recheck
56:         * thread states. */
57:        wdt_lock = 0;
58:    }
59:}
```

Listing 4.25: Parallel watchdog resetting.

In Listing 4.25, is a example program where are used three threads (lines 11 to 42), these threads are independent from each other. In this listing is also one thread that monitors the health of other threads, this thread is responsible for resetting watchdog if no activity is detected (lines 44 to 59). All independent worker threads perform designated tasks and after task is completed, a complete THn_COMPLETED bit is set up (lines 18, 29 and 40). WDT_RESET_THREAD monitors that all threads have set THn_COMPLETED bit (the line 50). This bit should be set by all threads, otherwise the watchdog hardware will not reset – the lines 51 to 58 will not be executed, and hardware watchdog circuit will cause consequent restart. The downside of this resetting method is the complexity. It requires additional CPU and memory resources for every thread and it is difficult to implement it in programs where threads have different activity periods (like communication threads, which are activated occasionally).

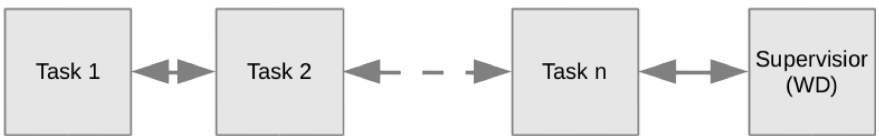

Figure 4.2: Serial watchdog

Another method for resetting watchdog is to use tokens that are passed by work order of threads (Figure 4.2). This implies to clearing of alive bits of the

threads – every thread clears its own alive bit and sets next alive bit of the thread, and finally the watchdog resets when the last thread clears its alive bit. For example, if it is known that the first thread performs its tasks always before the second, and the second thread will always run before the third, then a watchdog should be reset by the third thread when it has cleared the second thread's complete bit. But if the second thread does not set alive bit again within a period that is required for resetting the watchdog, it will trigger a watchdog reset. The following code example 4.26 illustrates above described watchdog resetting method.

```
 1:#define TH1_COMPLETED 0
 2:#define TH2_COMPLETED 1
 3:
 4:/* Variable that hold lock for watchdog access. */
 5:uint8_t wdt_lock = 0;
 6:
 7:THREAD_1
 8:{
 9:    while (1)
10:    {
11:        /* Task which completion is monitored by wdt_lock. Flag
12:         * TH1_COMPLETED set only when task is successfully
13:         * completed. */
14:        wdt_lock |= (1 << TH1_COMPLETED);
15:    }
16:}
17:
18:THREAD_2
19:{
20:    while (1)
21:    {
22:        /* Task, which completion is monitored by wdt_lock. Flag
23:         * TH2_COMPLETED is set only when task is successfully
24:         * completed and also THREAD_1 has set TH1_COMPLETED
25:         * flag (i.e. this thread is also successfully completed
26:         * its tasks).*/
27:        if (wdt_lock & (1 << TH1_COMPLETED))
28:        {
29:            wdt_lock |= (1 << TH2_COMPLETED);
30:
31:            /* Release TH1_COMPLETEF flag. This flag is set by
32:             *  THREAD_1. */
33:            wdt_lock &= ~(1 << TH1_COMPLETED);
34:        }
35:    }
36:}
37:
38:THREAD_3
39:{
40:    while (1)
41:    {
42:        /* Reset watchdog timer only when THREAD_2 has set
43:         * TH2_COMPLETED flag  (i.e. this thread is also
44:         * successfully completed its tasks). */
45:        if (wdt_lock & (1 << TH2_COMPLETED))
46:        {
47:            /* Reset the watchdog timer. */
```

```
48:             wdt_reset ();
49:
50:             /* Release TH2_COMPLETEF flag. This flag is set by
51:              * THREAD_2. */
52:             wdt_lock &= ~(1 << TH2_COMPLETED);
53:         }
54:     }
55:}
```

Listing 4.26: Serial watchdog resetting.

In the listing above is a program example with three independent threads; the only communication between the treads is a shared variable *wdt_lock*. The only requirement for this program is that all threads run sequentially, first runs thread no. 1 (lines 7 to 16), then no. 2 (lines 18 to 36) and lastly, thread no. 3. (the lines 38 to 55). For resetting watchdog, the following steps are taken: the first thread completes its tasks and sets the bit TH1_COMPLETED (line 14) in *wdt_lock* variable; then second thread completes its tasks and checks that the first thread has permitted (test that TH1_COMPLETED bit has set) to set the bit TH2_COMPLETED (lines 27 to 34). Setting the bit TH2_COMPLETED clears also the TH1_COMPLETED (the line 33) bit. In the last step when the TH2_COMPLETED bit is set and the third thread has completed all tasks, the watchdog timer can finally reset (lines 45 to 53). If the TH2_COMPLETED bit is not set, the watchdog will reset in the next watchdog resetting cycle; this also clears the TH2_COMPLETED bit (line 52). Presented watchdog resetting mechanism allows to monitor all threads by using a single hardware watchdog. Should at least one thread freeze, the reset will be carried out by hardware.

The two examples above are both similar in terms of CPU resource requirements and program memory or RAM utilisation.

Third and the simplest method is to reset the watchdog without monitoring all threads directly; instead, an *idle* thread should be used for resetting the watchdog. This is usable only with a cooperative kernel – in most situations where one thread freezes, all CPU resources will be taken and therefore the *idle* thread is not able to reset the watchdog timer. The only problem with this approach is that when a faulty thread has at least one sleep instruction, which allows the *idle* thread to regularly reset the watchdog timer, is not possible to use this method for resetting the system by the watchdog.

To conclude this section, the best watchdog resetting methods are parallel thread monitoring and sequential monitoring, and in some rare cases where the program execution order is not defined, it is possible to use a mixture of both methods. The easiest way to reset a watchdog is to reset it from an *idle* thread, which is reasonable to use in smaller programs; in larger programs it is difficult to use and serial or parallel watchdog resetting is preferred.

4.5. Common Optimisations Methods for Embedded Systems

While the compiler technology is constantly advancing, the modern compilers are able to create very well optimised program code by using several different optimisation techniques. However, still some special source code constructs may have significant impact on the program execution speed and size. Code optimisation by compiler is unfortunately limited by relatively simple methods like loop optimisation, dead code removing and reordering statements [99]. Generally, the optimisation that has highest net impact to program execution speed is carried out by the compiler, but some optimisations remain on developer's responsibility. The following sections describe improvements that allow to decrease significantly memory, CPU or IO resource consumption. Described improvements were necessary to enable resource consuming calculations on a buoy system onboard microcontroller. These calculations include heel angle calculations [67] and buoy collision detection [68]; also methods that are used for data outputting when profiling buoy behaviour are described [66].

4.5.1. Limiting Function Arguments

Data is passed to functions mostly by using parameters, which add overhead to a program. In most cases registers are used for parameter passing. but also RAM is used. Both methods consume a noticeable amount of microcontroller resources. Furthermore, functions that have a large number of parameters have typically several different responsibilities, which make program testing and maintaining more complicated; typically, every function parameter adds at least one test case. Therefore, depending on the coding style, maximum number of parameters is limited between 3 to 5 [58].

Although functions should have minimal amount of parameters, still many functions exist that use a large number of parameters. Several functions that require more than three parameters were used in TM software for acceleration data processing [65, 67] – three parameters for acceleration values, each acceleration value was two bytes long, and one parameter for timestamp, which is a four byte value; all together take up 12 bytes of stack or RAM. However, to overcome the overhead caused by a large number of parameters, it is possible to implement parameter passing by structure pointer passing; in case of 8-bit AVR microcontroller this takes two bytes of memory. In the following example (the Listing 4.27) is such situation where a function with four parameters is needed.

```
1:/* Structure for parameters. */
2:typedef struct
3:{
4:    uint8_t v1;
5:    uint8_t v2;
6:    uint8_t v3;
7:    uint8_t v4;
8:} params_s;
9:
```

```
10:/* Function that require several parameters. */
11:void fn (params_s *par)
12:{
13:    /* Do something with passed parameters. */
14:}
15:
16:void caller (void)
17:{
18:    /* Fill structure that holds parameter values. */
19:    params_s par = {1, 2, 3, 4};
20:
21:    /* Call function that uses large number of parameters, pass
22:     * parameter strucuture by pointer. */
23:    fn (&par);
24:}
```

Listing 4.27: Parameters passed by structure.

In the listing above, first the structure *params_s* (the lines 2 to 8) is defined that contains parameters that will all be passed to function *fn* (the lines 11 to 14). This structure holds four parameters and passing the structure is implemented as passing a pointer to the structure memory area. On the lines 16 to 24 is a function that calls the function *fn*, and passes four parameters to *fn*. If this example would have been implemented to pass four different variables to functions *fn*, the function fn would take four bytes of stack, however, using a pointer it would take only two bytes.

It is worth to note that in C++ where references are implemented as pointers, it is recommended to pass objects as references. This is much more effective than pass by value – a copy constructor [61] is always called when an object is passed by value.

4.5.2. Program Code Inlining

The best way to decrease usage of microcontroller CPU and stack resource is to change smaller functions to inline functions. Inline functions are functions that are copied to places where the call to specified function is located. Such technique reduces significantly CPU and stack usage and also increases the program execution speed. Best candidates for inlining are functions that are only one line long – these functions are typically only several machine instructions long (Listing 4.28). It is not reasonable to consider these functions as real functions. In a program example (Listing 4.28), an inline function *set_output* is defined (lines 5 to 11), which sets microcontroller's PORTB state to logical 0 (line 10) or to logical 1 (line 8), depending on input parameter. This function was called by two different functions, first by *init_output* (lines 15 to 22) and secondly by *main* (lines 24 to 42). When called first time (line 21), it is possible to optimise this function in such a way that only this code is compiled that was in a true branch – this function uses a constant parameter and the compiler allows to ignore the code parts that are not required. If the *set_output* function had been written as non-inline function, then in the line 21 the call instruction would change to real function, which has exactly two branches like

in the function on lines 5 to 11. The second call to *set_output* function is on the line 40, but in this case, it is not possible to predict which branch is needed and therefore the whole *set_output* function is copied to this location.

```
 1:/* Function which change output port state accordingly to input
 2: * parameter 'val'. When parameter val is non null then port B
 3: * pin 0 is set to high level, when parameter is null then port
 4: * B pin 0 is set to low level. */
 5:static inline void set_output (const uint8_t val)
 6:{
 7:    if (val)
 8:        PORTB |= (1 << PB0);
 9:    else
10:        PORTB &= ~(1 << PB0);
11:}
12:
13:/* Initialise output port, and set port B pin initially to low
14: * level. */
15:static inline void init_output (void)
16:{
17:    /* Set port B pin 0 to output. */
18:    DDRB |= (1 << PB0);
19:
20:    /* Set output initially to high. */
21:    set_output (1);
22:}
23:
24:int main (void)
25:{
26:    /* Variable for temporary output value. */
27:    uint8_t output_value;
28:
29:    /* Initialise output port. */
30:    init_output ();
31:
32:    while (1)
33:    {
34:        /* Some code. */
35:
36:        /*  get_output_value give new value to output. */
37:        output_value = get_output_value ();
38:
39:        /* Change output state. */
40:        set_output (output_value);
41:    }
42:}
```

Listing 4.28: Using inline functions.

When inlining larger functions, and if these functions are used in several places, then in most cases, the compiler warns about code growth. In embedded systems where programs are executed from program memory (typically from flash memory), the increased program size is irrelevant in most cases. For example, the ATMega1280 microcontroller has 128 kB of program memory and 8 kB of RAM, and it is not uncommon that programs that take half of the program memory require most of the target microcontroller RAM. In such system, it is reasonable to use inline functions as much as possible and it may

be necessary to suppress compiler warnings, which notify about program size growth. The inline functions are copied to a place of calling, and do not use any stack; compared to regular functions, each inline functions typically save 2...4 bytes of stack. Also the absence of calling and return instructions increases the speed of program execution.

Although inline functions allow to save significant amount of memory, these functions may also have some downsides – as inline functions do not exists as separate functions in program code, it is not possible to take function address. If it is still tried to take address from an inline function, two possible outcomes occur. Firstly, the compiler may refuse to compile this source file or function and secondly, the compiler creates a copy from inline function, and this copy will not be inlined and the returned address will be this non-inlined function's address.

The above paragraph described the use of inline functions in C. In most of the cases, it is applied to C++ as well. The only place where caution should be taken is when inlining is used in constructors, destructors and templates. Inlining these program parts may increase the executable program size significantly [61].

4.5.3. Fast Hardware Access

Interrupts are hardware mechanisms for notifying that new data is present in some hardware register. When data is available, an interrupt is triggered and the program continues on interrupt vector and later returns to the site where program execution was before the interruption. Executing an interrupt routine usually means that the CPU uses jump instruction to enter an interrupt vector and when leaving, uses an interrupt return instruction. Typically, jump and return instructions require several CPU clock cycles. The AVR microcontroller needs four clock cycles to enter and two cycles to exit. In cases when it is needed that some program reacts extremely fast to input change, polling can be used instead of interrupts. Depending on input reading instruction, one input poll takes two to four clock cycles, which is saving significant time compared to interrupts.

In the Listing 4.29, is an input polling example for AVR microcontrollers. In this example a while loop is executed until PB0 input is logical 1; if input goes to logical 0, the program continues after line 8. As reading from hardware PORTB is defined as reading from a volatile variable, the compiler does not optimise the loop body (lines 3 to 8). The *nop* instruction on line 8 serves as a placement mark for the Listing 4.30.

```
1:while (PORTB & (1 << PB0))
2:{
3:    /* While loop body. */
4:
5:    /* This nop instruction is neccerary for finding exact code
6:     * part in disassembled code, it does not have any other
```

```
7:     * purpose in this example. */
8:    __asm__ __volatile__ ("nop");
9:}
```

Listing 4.29: Input polling example.

While disassembling the example above, it is seen that this function is five CPU clock cycles long when the input is logical 1; and three cycles long when the input is logical 0 (Listing 4.30).

```
1:L0: sbis 0x05, 0; skip next instruction when bit 0 is set in
2:                ; register 5 (PORTB), 1 clock cycle when false,
3:                ; 2 clock cycles when true
4:    rjmp L1     ; jump out from loop (jump to label L1), 2 clock
5:                ; cycles
6:    nop         ; placement mark, no operation, 1 clock cycle
7:    rjmp L0     ; jump back to beginning of the loop (jump to
8:                ; label L0), 2 clock cycles
9:L1:
```

Listing 4.30: Assembler output for Listing 4.29.

Program example in Listings 4.29 and 4.30 take 6 to 7 instruction when executed on AVR microcontroller. Both examples take also 7 bytes of program memory.

A similar program for ATMega1280 that has same functionality but uses interrupts takes at least five instructions to enter and return [7]. Other microcontrollers may have different number of clock cycles to enter and return the interrupt handler but using interrupts is typically not as fast as polling.

4.5.4. Byte Order Manipulation

Many programs need to change the order of bytes, to construct a larger variable from the set of bytes, or to change order of bytes in a network message. In the Listing 4.31 is a typical function of this kind. This function constructs a 16-bit variable from two 8-bit variables. In the line 1, a pointer is passed to constant array of bytes; this array should be at least two elements long. In line 6, a byte is taken from buffer element 1 and shifted 8 bits to the left; finally, the result is added (in this example, adding is logical) to buffer element 2; the result will be returned when the functions ends.

```
1:uint16_t change_order_shift (const uint8_t *buf)
2:{
3:    /* Shift fist buffer element to left by 8 bits and add
4:     * buffer second element to result – i.e. {0xAA, 0xBB} ->
5:     * 0xAABB. */
6:    return ((buf[0] << 8) | buf[1]);
7:}
```

Listing 4.31: Using shift to change byte order.

Disassembled code is presented in the Listing 4.32. The AVR GCC version 4.9.0 was used for compiling this example. Disassembled program is 10

instructions long (including return instruction) and uses nearly same instructions, which were required in a source code. In the line 1, the source memory address is copied to register Z, which is an indirect addressing register. In line 3, the first byte is read to register 18, and in line 5 cleared register 19. The registers 18 and 19 are handled as one 16-byte register where the 8-bit value is placed; in lines 7 and 10, the 8-bit is shifted to right. The second element of the array is read in line 11 and placed into a temporary register which then is added to the first variable in line 15. In this example there are several instructions that are not needed for output – for example, line 5 shows clearing of the register 19 where several instructions later a new value is written. Also on lines 13 and 16 are shown redundant *movw* instructions.

```
 1:    movw r30, r24  ; move pointer to indirect addressing
 2:                   ; register
 3:    ld   r18, Z    ; load first byte from buffer to temporary
 4:                   ; register
 5:    ldi  r19, 0x00 ; set lower byte to 0x00 in destination
 6:                   ; register pair (unesseray instruction)
 7:    mov  r19, r18  ; move loaded byte to destination register,
 8:                   ; this instruction performs also 8 bit shift
 9:                   ; left
10:    eor  r18, r18  ; clear temporary register
11:    ldd  r24, Z+1  ; load second byte from buffer to another
12:                   ; temporary register
13:    movw r20, r18  ; move first register pair to another
14:                   ; temporary register
15:    or   r20, r24  ; add lower 8 bits to output register
16:    movw r24, r20  ; move register pair to function output
17:                   ; register
18:    ret            ; return from this function
```

Listing 4.32: Listing of the 'change_order_shift' function.

In the Listing 4.33 is a function that constructs same 16-bit variable but uses union for this purpose. In this example, it is not expected that the compiler uses optimal instructions for compiling with this function. Instead, exact steps are specified to change the order of bytes. In line 3, is passed a pointer to constant array of bytes to the *change_order_union* function; underlying array should be at least two elements long. In the lines 6 to 13 a union is defined, which contains two elements: one array with the length of two bytes (line 9), and one 16 bit variable (line 12). In lines 16 and 19, elements from the input buffer buf (an array) to union *addr_u* are copied and the returned 16-bit variable is shown in line 23.

```
1:/* Change order of bytes by using union. Array of input bytes is
2: * given by pointer.*/
3:uint16_t change_order_union (const uint8_t *buf)
4:{
5:    /* Union for conversion. */
6:    union
7:    {
8:        /* This array contains mapping to the 16-bit element. */
9:        uint8_t byte[2];
```

```
10:
11:         /* This element contains 16-bit value. */
12:         uint16_t word;
13:     } addr_u;
14:
15:     /* Copy last buffer element to union first element. */
16:     addr_u.byte[0] = buf[1];
17:
18:     /* Copy first buffer element to union last element. */
19:     addr_u.byte[1] = buf[0];
20:
21:     /* Return 16-bit element from union, which has reversed
22:      * byte order. */
23:     return addr_u.word;
24:}
```

Listing 4.33: Using union to change byte order.

The Listing 4.34 presents disassembled program code from the Listing 4.33. AVR GCC version of 4.9.0 was used for compiling this example. This code is 4 instructions long (including the return instruction) and it uses exact instructions that are required for changing the order of bytes. In line 1 is the source memory address, copied to register Z (the indirect addressing register). The lines 3 and 5 show code where bytes are copied from memory location and placed to output register in reversed order. The resulting value has an exact reverse order compared to the original value.

```
1:    movw r30, r24 ; move pointer to indirect addressing
2:                  ; register
3:    ldd  r24, Z+1 ; load second byte into lower half of output
4:                  ; register pair
5:    ld   r25, Z   ; load first byte into higer half of output
6:                  ; register pair
7:    ret           ; return from this function
```

Listing 4.34: Listing of the 'change_order_union' function.

Using union to change order of bytes makes the resulting machine code significantly smaller but this function may depend on hardware. It is possible to port the first function (Listing 4.31) without modifications to different architectures, but using the second function in a different architecture may involve conditional compilation.

However, in cases when it is needed to convert an array of bytes to a larger integer, but the byte order remains the same, conversion is a trivial task. This can be accomplished simply by casting array elements to a larger type. Such casting is not compatible with MISRA C rule 11.4 [71].

4.5.5. Optimisation of AES Cryptographic Functions

As embedded systems are used quite widely in control applications, it is reasonable that in unsecured environment the communication channel shall be encrypted. Although many new microcontrollers have hardware cryptography engine [6], the majority of microcontrollers do not have it. In small embedded

systems mostly symmetrical key cryptographic algorithms are used, such as AES, DES (Data Encryption Standard) and XTEA (eXtended Tiny Encryption Algorithm). It is also possible to use algorithms with asymmetrical keys, but these algorithms consume significant amount of microcontroller resource; usually at least one magnitude more than the symmetrical keys, and therefore the microcontroller is not able to perform other tasks. Currently it is known that only elliptic curve cryptography (ECC) is usable in 8-bit microcontrollers [88]. Methods that are described in this section are used with the AES algorithm, with 128 bit keys. But it is also possible to use it with 192 or 256 bit keys.

While the AES cryptographic algorithm [78] requires relatively small amount of processor and memory resources [18], it may still require more resources than smaller microcontrollers may actually have. It is possible to optimise processor or memory resource consumption where the substitution table (*S-Box*) [78] calculations are performed. It is possible to compute the *S-Box* values in three different ways. Firstly, without precomputed *S-Box* values all values are calculated when needed [53]. Secondly, the *S-Box* values are computed during a program start [4]. Thirdly, the *S-Box* values are generated before the compilation – all the values are program memory constants [81]. In an embedded system, the second method where S-Box values are generated when program starts, is the least suitable. This method places the generated values to RAM and either to static or dynamic storage. It is suitable in systems where there is enough spare memory. Other two methods are suitable for embedded systems. The first method where the *S-Box* values are generated when needed is reasonable for use in such cases where cryptographic functions are required only few times, and encryption and decryption are not time critical. This uses quite small amount of memory, both program and RAM, but requires more CPU resource. Typically, it takes around 100 bytes of program memory and up to 32 bytes of RAM (short time storage) but it requires several hundreds of CPU cycles to calculate constants (exact number depends on code inlining and optimisations). In the Listing 4.35, is a function for S-Box calculation. In this listing, function body is left out; the main purpose of this listing is to show the difference between the first and third method (Listing 4.35).

```
 1:/* Function for S-Box calculation, where parameter 'x' is byte
 2: * value that has to be mapped to S-Box. */
 3:uint8_t rj_sbox (const uint8_t x)
 4:{
 5:    uint8_t rj_sbox_value;
 6:
 7:    /* Calculate rj_sbox_value, this may take significant amount
 8:     * of CPU resource. */
 9:
10:    return rj_sbox_value;
11:}
```

Listing 4.35: Function for S-Box value calculation.

In case of the third method, the *S-Box* values are generated before the compilation. This uses more program memory area, it requires at least 256 bytes

but not more than 300 bytes of memory for table storage and accessors, but is nearly same fast as the second one (reading constant values from a program memory uses some CPU cycles more than reading values from RAM). However, it does not require any additional RAM for *S-Box*. In the Listing 4.36 is the *S-Box* implementation for reading constants from a program memory. The *S-Box* values are defined in the lines 2 to 5 and required read function is defined in line 9.

```
1:/* Array of precomputed S-Box values. */
2:static const uint8_t PROGMEM sbox[256] =
3:{
4:    /* AES S-Box constants */
5:};
6:
7:/* Read S-Box value fom constant array. This macro emulates an
8: * rj_sbox function. */
9:#define rj_sbox(x) pgm_read_byte (&sbox[(x)])
```

Listing 4.36: Using program memory for AES S-Box.

```
 1:void aes_sub_bytes (uint8_t *buf)
 2:{
 3:    uint8_t i;
 4:
 5:    for (i = 0; i < 16; i++)
 6:    {
 7:        /* Fill buffer by S-Box values. These values are
 8:         * calculated by 'rj_sbox' function or retrieved by
 9:         * using macro from precalculated array of constants. */
10:        buf[i] = rj_sbox (buf[i]);
11:    }
12:}
```

Listing 4.37: Using S-Box constants in AES.

In the Listing 4.37 is a function that can use functions that are described in the Listings 4.35 and 4.36. In lines 5 to 11 is a function for reading the *S-Box* values from program memory, or calculated on the fly; read values are placed into the output array *buf*.

4.6. Dynamic Memory

Most of the programs for non-embedded devices use dynamic memory intensively. Typically, the program asks the kernel for additional memory during the execution by using special standard library functions. Asking for memory every time when its needed is much easier than to estimate the exact memory requirements during development. In some cases, asking for more memory is the only possible way to handle large data sets. To use dynamic memory, the first required task is to ask for a memory block. This can be done by using special command and later, when the memory block is not required any longer,

it can be released[12]. In embedded systems, which have a multithreaded programs, and several processes using dynamic memory concurrently, without any mechanisms implemented for prevention of memory fragmentation it is possible to fragment the entire free memory and eventually it is not available any continuous free memory with required size; i.e., the memory is exhausted. In this situation it is possible that the system still has enough free memory, however, all the memory blocks are too small for the required size, and it can happen that it is not possible to combine larger blocks by adding several smaller blocks together. Also, effects of memory fragmentation may surface differently every time; bugs, which are related to this, may show up in the field but not in the development environment. Above are described some of the reasons why it is not advised to use dynamic memory in embedded systems. It is not allowed to use dynamic memory by MISRA C (rule 20.4 [71]), MISRA C++ (rule 18-4 [72]) and JSF++ (AV Rule 206 [55]). Unlike in embedded systems, fragmentation is not an issue in desktop computers – there are several hardware and software mechanisms available to mitigate it. Despite quite a lot of research [20, 59] has been done in this area, and several memory allocators are developed (PJSIP Fast Memory Pool [105], which can use stack based pools, Doug Lea memory allocator – *dmalloc* [52] and TLSF [17], memory allocators from Molecular Musings have two different allocation strategies – stack-like and linear [63]) for small embedded systems only one implementation is known that uses pool: it is implemented as part of FunkOS [27].

Memory allocation and deallocation functions presented in the current thesis allow to avoid above described memory exhaustion problems; they are available for TM as an alternative dynamic memory management and used several times for debugging purpose. Presented functions also have overrun detection capability. As the presented solution is quite simple and used in systems where there is only one type of memory available, therefore it is not possible to use different pools, for example one for fast access but small blocks and another for slow access and large blocks. Similar approach is presented in Effective C++ [61], but the current example has been written for C. Effective C++ allocation functions do not contain any underrun detection, i.e., it does not detect writing prior the beginning of an allocated block. These occasions are relatively rare and in most cases are detected by previous block overrun detection. GCC has also similar protection mechanisms but currently these are not available for smaller microcontrollers [29]. Presented functions do not take into account memory alignment and this may cause some issues on larger computers. Alignment is not an issue in an 8-bit microcontroller where the memory is typically byte aligned.

The Figure 4.3 presents the main principle of the developed allocator. Memory blocks are stored in a pool, which is a large two-dimensional array. The pool size is determined before the compilation. Using a predefined memory

[12] Some languages like Java allow to release memory automatically, but some languages like C require that user releases the unused memory block.

pool size allows to determine memory requirements during compilation. In situations where all pool items are shared out, memory request operation will block until at least one new block is available.

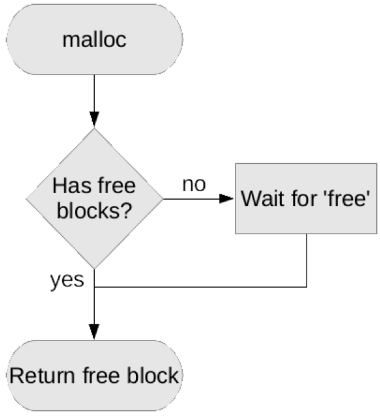

Figure 4.3: Pooled allocator

In the following Listings 4.38, 4.39 and 4.40 is an implementation of described allocator, which uses memory pool for back end. Presented code is more flexible than similar functions in the FunkOS source; it is also a drop-in replacement for regular *malloc*, and it is capable of detecting memory overrides. The Listing 4.38 is a header file for the pooled allocator. The line 8 defines pool element size and the lines 11 to 23 define pool element structure where the variable *is_taken* represents pool state (boolean value). An array named *data* is a memory area that is returned by *pmalloc* function, and finally the member named *guard* that holds a four byte signature for detecting overflows is defined. As elements in the structure *pool_data_t* are 8 and 32-bit wide, this structure may be padded in 32-bit architectures. Whether it will be padded depends on the value of ITEM_SIZE, should after division by four the remainder be three, no padding occurs. When a structure is padded, it is possible to write a number of padded bytes over data element before overwriting is detected. The lines 29 to 30 show function declarations for registering a new memory pool. The last two declarations in line 36 and 40 are the replacements for *malloc* and *free* functions.

```
 1:#ifndef __PMALLOC_H_
 2:#define __PMALLOC_H_ 1
 3:
 4:#include <stddef.h>
 5:#include <stdint.h>
 6:
 7:/* Size of single memory element. */
 8:#define ITEM_SIZE 128
 9:
10:/* Structure for pool information. */
11:typedef struct
12:{
```

```
13:    /* Flag for taken state. */
14:    uint_fast8_t is_taken;
15:
16:    /* Shared memory area. */
17:    uint8_t data[ITEM_SIZE];
18:
19:    /* Pattern to detect memory corruption, this should be
20:     * always after pool data. */
21:    uint32_t guard;
22:}
23:pool_data_t;
24:
25:/* Function for pool registration. Where parameter 'new_pool' is
26: * pointer to memory area that holds pool data. Parameter
27: * 'pool_items' is number of pool elements, which is stored in
28: * 'new_pool'. */
29:extern void register_pool (pool_data_t *new_pool,
30:                           const unsigned int pool_items);
31:
32:/* Malloc function. Where parameter 'size' is requested memory
33: * size. This function returns allocated memory or NULL when no
34: * free memory is available or requested memory is larger than
35: * single pool element can hold. */
36:extern void *pmalloc (const size_t size);
37:
38:/* The 'pfree' function causes the allocated memory referenced
39: * by 'ptr' to be made available for future allocations. */
40:extern void pfree (void *ptr);
41:
42:#endif
```

Listing 4.38: Header file for pooled allocator and deallocator.

In the Listing 4.39 is a source code of a pooled allocator. The line 9 defines pattern for detecting memory overflows, this constant is used as a guard pattern and is compared against guard element in *pool_data_t* structure when memory region is freed. In the lines 11 to 15 are declared holders for memory pool size information (line 12) and pool data (line 15), in the lines 17 to 20 is conditionally compiled lock variable, which is used for locking functions when all memory is shared out. Memory pool registration function is implemented in the lines 25 to 46. Lines 30 and 35 check that only one pool is defined, and if more than one pool is defined, it simply ignores the request for new pool registration. In the lines 38 and 39 the pool and pool information is copied to local static variables, and in the lines 41 and 45 guard bytes are set. It is possible to declare a pool by using static pool structure; by leaving these functions out, it will allow to use *pmalloc* without initialisation.

In the lines 52 to 91 is an implementation of the *pmalloc* function. This function returns a pointer to memory pool, or in case of an error, the NULL pointer and sets the *errno* variable to ENOMEM. In the lines 56 to 62, requested size is tested against maximum element size. When more than a single pool element can hold is requested, NULL will be returned, which means an error and *errno* is the relevant value (ENOMEM). In the lines 64 to 85 a free pool element is searched. If a free element is found, it will be marked as taken

(the line 76) and returned to pointer in this memory area (line 77). If no free pool elements exist, it will wait until the next free pool element (line 83) is released and then the pool will be scanned again for free elements. If no waiting code is present, the lines between 64 to 68 and 81 to 85 are omitted. Without code for signal waiting and when no free pool element is present, NULL will be returned and *errno* value is set to ENOMEM. If event waiting code is present, the code in the lines 89 and 90 is not executed. However, these lines are required as this function is declared to return a value and, typically, compilers refuse to compile functions that are declared to have return value while no return value is present. Memory releasing is done by *pfree* function, which is implemented in the lines 95 to 127. In lines 99 to 126 an address from the memory pool is searched. If given address is found in the pool (line 104) then this pool element is marked as free (line 118). A free signal is sent (line 122) to *pmalloc*, this signal is caught when *pmalloc* is waiting for the pool element release (line 83), and finally, *pfree* function returns in the line 124. If waiting code is not present then after a pool element release (line 118) is conditionally compiled code, which is responsible for sending the “element free” signal to *pmalloc*. The *pfree* function is also responsible checking for memory overflows. Memory checking functionality is in the lines 109 to 114. When memory overflow occurs, the guard byte area will be overwritten and it can be detected by comparing this memory area with known guard bytes (line 109). Should these bytes and memory area not match, a short error message is printed (lines 111 and 112) and the program is terminated (line 113). Memory error message printing and program termination could be changed to some other action. In an embedded system, this is the most reasonable thing to do; after calling the exit function, a watchdog is typically triggered and the whole program restarts, but with corrupted memory the program may have unpredictable behaviour.

```
 1:#include <errno.h>
 2:#include <stddef.h>
 3:#include <stdio.h>
 4:#include <stdlib.h>
 5:#include <string.h>
 6:#include "pmalloc.h"
 7:
 8:/* Pattern for detecting memory overflows. */
 9:#define GUARD_PATTERN 0xDEADBEEF
10:
11:/* Number of pool items. */
12:static unsigned int items = 0;
13:
14:/* Pointer to registred pool. */
15:static pool_data_t *pool = NULL;
16:
17:#ifdef HAS_WAIT_SIGNAL
18:/* Signal for memory release (if it is present). */
19:static signal_t sig_pfree;
20:#endif
21:
22:/* Pool registration. Where parameter 'new_pool' is pointer to
23: * memory area that holds pool data. Parameter 'pool_items' is
```

```
24: * number of pool elements, which is stored in 'new_pool'. */
25:void register_pool (pool_data_t *new_pool,
26:                    const unsigned int pool_items)
27:{
28:    unsigned int i;
29:
30:    if (pool != NULL)
31:    {
32:        /* If pool is already registred then reuturn
33:         * immediately. */
34:        return;
35:    }
36:
37:    /* Store pool pointer and number of pool items. */
38:    pool = new_pool;
39:    items = pool_items;
40:
41:    for (i = 0; i < items; i++)
42:    {
43:        /* Fill guard area with predefined pattern. */
44:        pool[i].guard = GUARD_PATTERN;
45:    }
46:}
47:
48:/* Malloc function. Where parameter 'size' is requested memory
49: * size. This function returns allocated memory or NULL when no
50: * free memory is available or requested memory is larger than
51: * single pool element can hold. */
52:void *pmalloc (const size_t size)
53:{
54:    unsigned int i;
55:
56:    if (size > ITEM_SIZE)
57:    {
58:        /* Requested size is larger than single pool element can
59:         * hold. Return NULL and set error description. */
60:        errno = ENOMEM;
61:        return NULL;
62:    }
63:
64:#ifdef HAS_WAIT_SIGNAL
65:    /* Loop for waiting free element. */
66:    while (1)
67:    {
68:#endif
69:        for (i = 0; i < items; i++)
70:        {
71:            /* Search element that has is_taken field false. */
72:            if (pool[i].is_taken == 0)
73:            {
74:                /* Mark this elemen as taken and return pointer
75:                 * to this area. */
76:                pool[i].is_taken = 1;
77:                return &pool[i].data;
78:            }
79:        }
80:
81:#ifdef HAS_WAIT_SIGNAL
82:        /* Wait until least one element is released. */
```

```
 83:         wait_signal (&sig_pfree);
 84:     }
 85:#endif
 86:
 87:     /* No free block found. This code is compiled only when
 88:      * HAS_WAIT_SIGNAL is not defined. */
 89:     errno = ENOMEM;
 90:     return NULL;
 91:}
 92:
 93:/* The 'pfree' function causes the allocated memory referenced
 94: * by 'ptr' to be made available for future allocations. */
 95:void pfree (void *ptr)
 96:{
 97:     unsigned int i;
 98:
 99:     for (i = 0; i < items; i++)
100:     {
101:         /* Search for pool element by given pointer. It is
102:          * possible to free element that is present in pool.
103:          * Null pointer is also allowed. */
104:         if (ptr == &pool[i].data)
105:         {
106:             /* Check that guard pattern is valid. If guard
107:              * pattern is modified is most reasonable action to
108:              * close program. */
109:             if (pool[i].guard != GUARD_PATTERN)
110:             {
111:                 fputs ("memory pool guard pattern is "
112:                        "corrupted\n", stderr);
113:                 exit (EXIT_FAILURE);
114:             }
115:
116:             /* Guard patten is OK, we can mark this pool
117:              * element as available. */
118:             pool[i].is_taken = 0;
119:
120:#ifdef HAS_WAIT_SIGNAL
121:             /* Send singal that element is released. */
122:             send_signal (&sig_pfree);
123:#endif
124:             return;
125:         }
126:     }
127:}
```

Listing 4.39: Source code for pooled allocator and deallocator.

The Listing 4.40 shows a test program for the above code. In the line 8 is a defined number of pool elements and in the line 11 type definition for void pointer, which is used with pointer arrays. The main function of the program is in the lines 13 to 88. The line 16 defines memory back-end pool and on the lines 17 to 19 are the variables, which hold pointer that is returned by *pmalloc*. The line 22 ensures that all memory pool elements have null value; this is required only for taken variable in *pool_data_t* structure; this variable should be initially null. In the line 23 is initialised memory pool. The lines 26 to 58 show a test for the situation when all memory is shared out and no free memory remains. The

lines 62 and 66 release all the memory, which was taken in the previous test (lines 26 to 58). Finally, the lines 70 to 85 is a test case for memory corruption by overwriting last four bytes (lines 79 to 85). In 8-bit microcontrollers it is required to overwrite one byte but in 32 and 64-bit computers the memory alignment is different, and due to the padding, the guard element may have an offset up to three bytes.

```
 1:#include <errno.h>
 2:#include <stddef.h>
 3:#include <stdio.h>
 4:#include <string.h>
 5:#include "pmalloc.h"
 6:
 7:/* Number of elements in memory pool. */
 8:#define ELEMENTS 8
 9:
10:/* Type for void pointer (useful for casting). */
11:typedef void * void_ptr_t;
12:
13:int main (void)
14:{
15:    unsigned int i;
16:    pool_data_t new_pool[ELEMENTS];
17:    void_ptr_t data[ELEMENTS + 2];
18:    void_ptr_t data_tmp = NULL;
19:    void_ptr_t data_last = NULL;
20:
21:    /* Set all pool elements to 0 and register new pool. */
22:    memset (new_pool, 0x00, sizeof (new_pool));
23:    register_pool (new_pool, ELEMENTS);
24:
25:    /* Test for element allocation. */
26:    puts ("Allocate\n");
27:    for (i = 0; i < (ELEMENTS + 2); i++)
28:    {
29:        /* Request new memory. */
30:        data_tmp = pmalloc (ITEM_SIZE - 8 + i);
31:
32:        if (data_tmp == NULL)
33:        {
34:            /* Error: no memory returned, either requested size
35:             * is larger than single pool element can hold or no
36:             * pool elements are available. */
37:            printf ("%02d\t%s\n", i, strerror (errno));
38:        }
39:        else if (data_last == NULL)
40:        {
41:            /* First test run, previous pointer is not yet
42:             * stored. */
43:            printf ("%02d\tPool: %p; diff (NA)\n",
44:                    i, data_tmp);
45:        }
46:        else
47:        {
48:            /* Print pool element info and address difference
49:             * between pointers. */
50:            printf ("%02d\tPool: %p; diff %ld\n", i, data_tmp,
51:                  (ptrdiff_t)(data_tmp – data_last));
```

```
52:         }
53:
54:         /* Store allocated memory to array. This array allows to
55:          * release previously allocated memory. */
56:         data_last = data_tmp;
57:         data[i] = data_tmp;
58:     }
59:
60:     /* Test for element deallocation. */
61:     puts ("\nFree\n");
62:     for (i = 0; i < ELEMENTS; i++)
63:     {
64:         /* Free all elements that has stored to array. */
65:         pfree (data[i]);
66:     }
67:
68:     /* Test for data corruption. */
69:     printf ("\nData corruption test\n");
70:     data_tmp = pmalloc (ITEM_SIZE);
71:     if (data_tmp == NULL)
72:     {
73:         /* Failed allocate to memory, either no pool elements
74:          * are available or requested size was larger that
75:          * single element can hold. */
76:         printf ("%02d\t%s\n", i, strerror (errno));
77:     }
78:     else
79:     {
80:         /* Fill memory with 0xFF also overwrite guard bytes. */
81:         memset (data_tmp, 0xFF, ITEM_SIZE + 4);
82:
83:         /* This call shold detect error and close program. */
84:         pfree (data_tmp);
85:     }
86:
87:     return -1;
88:}
```

Listing 4.40: Example usage and test for pooled allocator and deallocator.

Above described function simplifies also debugging – when the memory size and memory contents placement is known, it is quite easy to check the overflows and find possible candidate that may cause it.

Program example in Listing 4.40 takes roughly the same amount of program memory than a similar application that is created for using dynamic memory. This example is as fast or faster than similar application that uses dynamic memory.

As a final note, when memory requirements during program development are known it is possible to reserve free memory before compilation. Therefore it is not required to use dynamic memory at all. Also in embedded systems it is not reasonable to use dynamic memory in such processes that do not release memory during program execution. Most of the buffers in drivers do not release memory, and in this case, it is possible to estimate required memory size before compilation.

4.7. Conclusions

The current chapter handles improvements that were directly or indirectly required to develop new TM. The first section outlines a software development process that can be used in embedded software development and gives examples for using hardware related automated tests. The following processes are handled in this section – Code and Fix, using UML as part of other processes, agile practices (TDD and BDD) and sequential processes (V-model and waterfall). In embedded software development, it is most difficult to use agile practices when Code and Fix is appropriate for experimenting and prototyping, and sequential processes for safety or mission critical software. Agile practices are too time consuming for experimenting and are not appropriate for mission or safety critical systems. It is also possible to use UML for modelling embedded software, but due to the higher resource requirements of the resulting program, it is usable only in larger embedded systems.

The second section concentrates on the use of different programming languages within one project. In that section the use of C and C++ programming languages is observed. The first part gives the main reasons for language choice: it is best to use C for smaller microcontrollers, older compilers and creating kernel related code, and C++ is best to be used for creating program logic. However, there is no limitation of usage of C++ with recent compilers. The second part of that section describes how to use several higher level languages within one software project. The main reason for using different languages in one project is that some tasks are best suited for one language while other tasks for another language. Also several examples are given on how to use program structures in C that are known from C++. This might be useful in situations where it is required to use virtual table like approach but due to the compiler or memory limitations it is not possible to use C++.

The third section concentrates on program structures and the influence of program structure in testing. The first part of this section handles super-loop programs. As this kind of programs are tightly coupled with hardware, methods are given for reducing hardware dependencies. It is also shown how to create functions that do not store states internally. Typically such functions are related to EEPROM reading and writing, and are inherently very difficult to test automatically. The following part focuses on hardware testing, and this is accomplished by using unit tests. For this testing is uploaded new test program to microcontroller memory. The last part of that section focuses on the use of OCD, on the limitations of OCD, on situations where it is not possible to test a program by using OCD, and on using one spare IO pin to monitor system states.

The fourth section deals with different type of schedulers and watchdogs. On conventional computers it is reasonable to use a preemptive scheduler, but in embedded systems where a watchdog is also used, it is worth to consider using only a cooperative scheduler. When one thread is stuck with a preemptive scheduler, all other threads may still work normally, including the thread that is

responsible for resetting the watchdog. When a program is stuck with a cooperative kernel, it causes the whole system freeze, including the watchdog's control task, which consequently triggers the watchdog to reset. The second part of that section describes different methods for the use of a single hardware watchdog in programs where all threads are active in predefined order, or in situations where threads are active at the same time. When there is a predefined order, it is required to reset the watchdog's pass lock from one thread to another. When threads are active at same time, then it is required to have one monitoring process that has access to all states of threads.

The fifth section describes methods of optimising several program constructs. First, methods are shown for limiting function arguments; instead of using parameters that are passed by value, pointers can be used for passing data structures. Secondly, function inlining is considered; while inlining large functions is not recommended in conventional computers, in smaller microcontrollers software relatively large functions can be safely inlined. This is due to the fact that conventional computers execute program from RAM (or from cache) but microcontrollers execute program directly from separate memory and therefore do not have similar cache effects like conventional computer have. As well, most of the microcontrollers have a reasonable amount of program memory. After that are given examples of how and when it is possible to use input polling instead of interrupts. Mostly, input polling is used when it is required to react extremely fast to input change. The chapter also describes the change of byte order, which is typically related to programs that interact with external hardware or programs. In order to change the byte order, shifts in C are typically used, but to use unions is more effective. This section also shows two optimisation methods for the AES algorithm: *S-Box* value calculation on the fly, and the use of pre-generated *S-Box* values, which are stored in the program memory.

The last section describes alternative approach for *malloc* and *free*. Regular memory allocation functions, which are used in embedded systems, may significantly fragment RAM, therefore it is possible that free memory can be exhausted. Memory exhaustion by fragmentation is rare in conventional computers but, in embedded systems, this may happen quite frequently. This is one of the reasons why MISRA does not allow to use dynamic memory. The current thesis presents a memory allocation method that uses user supplied memory pool as back-end, and hence makes it impossible to fragment the memory. Presented allocation and deallocation functions are also capable of checking the memory overruns.

This chapter described methods and improvements that are completely or partly applied and tested on TM. Although described solutions are TM specific, they can still be used in other similar devices.

5. SUMMARY

This thesis concentrates on software development improvements and solutions for embedded systems that have small computational power and limited amount of memory, namely marine aid to navigation (AtoN) systems.

More precisely, all research, improvements and solutions that are described in this thesis were required for a Telematics Module of AtoN systems; without these improvements development the product itself and its additional features would have been much more complicated to implement or failed completely. In this thesis the reasons why this research was required were discussed: coping with problems that arise when developing a new AtoN device, or adding some additional functionality. Main focus was on the following six issues:

1. Functions and methods that are described and investigated in this thesis should have low memory and processor utilisation, which make them suitable for use in low power AtoN devices, namely Telematics Module.
2. Mixing source code that is written in different programming languages, namely C and C++. Effective function pointers usage on structures and automated tests.
3. Software testing methods for small embedded systems, including using unit tests for hardware testing.
4. Watchdog hardware handling in multithreaded programs with monitoring all threads simultaneously.
5. Optimisation of code parts that are used quite widely in embedded systems but are not optimised by compiler.
6. Dynamic memory handling that does not fragment memory, is lightweight, and suitable for use in AtoN devices.

All publications that are related to this thesis, use described improvements directly or indirectly. All described improvements are tested and used successfully in AtoN telematics module or similar system software.

The next section outlines the main contributions of this thesis.

5.1. Contributions

The main contributions of the thesis are:

1. The choice of programming language that is usable in embedded AtoN systems.

   Several decades ago, the main programming language for embedded systems was assembler; now there are numerous programming languages available, but mostly C and C++ are being used. In this thesis

pros and cons of both languages and methods how it is possible to mix both languages in embedded AtoN systems to get the best result are shown.

2. Different program structures and improvements on embedded software testing.

   In embedded systems mostly two different program structures are used: super-loop programs and multithreaded programs. Super-loop programs have less functionality than multithreaded programs but use hardware resources more effectively. Multithreaded programs may have a lot of different functionalities but require a lot of CPU and memory resources and are not as efficient as super-loop programs. In both cases it is possible to use similar testing methods, but testing of super-loop programs is more complicated than it is in multithreaded programs, mainly because of the program structure, which is more related to hardware and therefore requires more hardware writing mocks.

3. Methods for resetting watchdog timers is non-trivial in multithreaded programs.

   In multithreaded systems it is possible to choose between several different schedulers – cooperative or preemptive. A preemptive scheduler allows to create such programs that do not hang when one thread hangs. However, programs with a cooperative scheduler tend to hang completely when at least one thread hangs. In embedded systems it is possible to take advantage of the hanging of the whole cooperative kernel. This excludes the situations where a system stays partially in working condition but is not able to perform its tasks. Another important aspect while writing embedded multithreaded programs is the watchdog resetting mechanism. Most hardware watchdogs require that only one thread is responsible for resetting the watchdog and, therefore, all threads should send alive messages to one thread or function that is responsible for handling the watchdog. For this purpose, it is possible to use two different methods; all threads send alive messages to one monitor thread, or all threads pass alive message from one thread to another.

4. Optimisation methods for embedded system software.

   In some cases, the compiler is not able to generate the most optimal code, e.g., does not automatically inline functions that are one line long or functions that are called only once. Also the compiler is not able to optimise program parts that are responsible for byte order manipulation. When using AES cryptographical functions, it is possible to use two different substitution table calculation methods. One of the methods uses less memory but is slower, another one uses more memory but is significantly faster.

5. Alternatives for dynamic memory.

   Using dynamic memory in embedded systems contains several risks, however, it is possible to replace traditional dynamic memory functions with memory pools. While using a memory pool, it is possible to set dynamic memory size before compilation and also monitor RAM usage.

5.2. Conclusions

This thesis concentrates on software development and testing methods of AtoN embedded systems. These systems are mainly designed to work in remote places like buoys or lighthouses or in other similar navigational applications. Described improvements and methods were required for developing new generation Telematics Module (TM) and are currently used in navigation light systems around Estonian coastal area and on the larger rivers and lakes. The proposed methods and improvements in this thesis enable to achieve low microcontroller CPU and memory consumption. Despite simplicity the methods are also quite robust to allow them to be used successfully in other similar systems which require long-term autonomous work, low power consumption, and where it is not essential to have high computational power. Although described methods and improvements are TM specific, they can be also used in other similar embedded systems.

This thesis discussed also programming languages that are suitable for use in smaller embedded systems. Different testing methods which can be used in various situations in embedded software developement were presented. Watchdogs that are used with multitasking kernels and different schedulers were also discussed. Several ways how to optimise embedded programs is shown – changing the order of bits in larger data word, using inlining and also some techniques for using cryptographic functions on smaller embedded systems. And finally the replacement for standard memory allocation and deallocations functions is presented, which allows to reduce memory fragmentation.

The main conclusion of this thesis is that it is possible to use above described methods, improvements and solutions on different AtoN systems that require low energy consumption long autonomy and high reliability. All described improvements have been in use in the deployed AtoN systems in Estonian and abroad. These improvements were made feasible to measure wave heights with navigational buoys, allows to detect buoy heel angle and buoy collisions with other objects.

The thesis presents also a short overview about Estonian AtoN systems and an overview of problems that raised during new TM development.

References

[1] "AT90S8515 - 8-bit AVR Microcontroller with 8K Bytes In-System Programmable Flash", Atmel Corporation, 2001.

[2] "AT90S1200", Atmel Corporation, 2002.

[3] "AVR230: DES Bootloader", Atmel Corporation, 2005.

[4] "AVR231: AES Bootloader", Atmel Corporation, 2006.

[5] "AVR32 Architecture Document", Atmel Corporation, 2011.

[6] "8/16-bit Atmel XMEGA A3BUMicrocontroller ATxmega256A3BU", Atmel Corporation, 2013.

[7] "ATMega640/1280/1281/2560/2561", Atmel Corporation, 2014.

[8] "JTAGICE mkII Special Considerations", http://www.atmel.no/webdoc/jtagicemkii/jtagicemkii.special_considerations_mega_debugwire.html, 2014.05.25.

[9] "STK500", http://www.atmel.com/tools/STK500.aspx, 2014.09.24.

[10] "STK600", http://www.atmel.com/tools/STK600.aspx, 2014.09.24.

[11] R. D. Banker, G. B. Davis and S. A. Slaughter, "Software development practices, software complexity, and software maintenance performance: A field study" - *Management science*, 1998, vol. 44, pp. 433-450.

[12] "BSP430 Board Support Package for MSP430 Microcontrollers", http://pabigot.github.io/bsp430, 2014.05.10.

[13] E. Blem, J. Menon and K. Sankaralingam. "Power struggles: Revisiting the RISC vs. CISC debate on contemporary ARM and x86 architectures" - *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pp. 1-12, 2013.

[14] C. Boogerd and L. Moonen. "Assessing the value of coding standards: An empirical study" - *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, pp. 277-286, 2008.

[15] Q. Cao, X. Wang, H. Qi and T. He. "r-Kernel: An operating system foundation for highly reliable networked embedded systems" -

INFOCOM, 2011 Proceedings IEEE, pp. 2507-2515, 2011.

[16] C. Constantinescu, "Trends and challenges in VLSI circuit reliability" - *Micro, IEEE*, 2003, vol. 23, pp. 14-19.

[17] "TLSF Memory Allocator Implementation", http://tlsf.baisoku.org, 2014.11.30.

[18] J. Daemen and V. Rijmen, "AES Proposal: Rijndael", 1999.

[19] D. Dahlby, "Applying agile methods to embedded systems development" - *Embedded Software Design Resources*, 2004, vol. 41, pp. 1014123.

[20] D. Dhurjati, S. Kowshik, V. Adve and C. Lattner, "Memory safety without garbage collection for embedded applications" - *Trans. on Embedded Computing Sys.*, 2005, vol. 4, pp. 73-111.

[21] "Toyota's killer firmware: Bad design and its consequences", http://www.edn.com/design/automotive/4423428/Toyota-s-killer-firmware--Bad-design-and-its-consequences, 2015.10.16.

[22] M. D. Earle, "Nondirectional and directional wave data analysis procedures", NDBC, 1996.

[23] "AVR-Ada", http://sourceforge.net/projects/avr-ada, 2014.11.30.

[24] F. P. Engelbertink and H. H. Vogt, "How to save on software maintenance costs", 2010.

[25] "M68HC12B Family Data Sheet", Freescale Semiconductors, 2005.

[26] "HC08", http://www.freescale.com/webapp/sps/site/taxonomy.jsp?code=HC08FAMILY, 2014.11.09.

[27] "FunkOS", http://funkos.sourceforge.net, 2014.11.30.

[28] J. Ganssle, "The Art of Designing Embedded Systems", Elsevier Science, 2008.

[29] "GCC, the GNU Compiler Collection", https://gcc.gnu.org, 2014.11.30.

[30] "PIC Series Microcomputer", General Instrument Corporation, 1977.

[31] "PC-lint for C/C++", http://www.gimpel.com/html/pcl.htm, 2015.10.16.

[32] "JamaicaVM", https://www.aicas.com/cms/en/JamaicaVM, 2014.12.03.

[33] J. W. Grenning, "Test-driven development for embedded C", Pragmatic Bookshelf, 2011.

[34] R. Grisenthwaite, "ARMv8 Technology Preview", 2011.

[35] L. Hatton, "Safer C: Developing Software for in High-Integrity and Safety-Critical Systems", McGraw-Hill, 1995.

[36] L. Hatton, "Safer language subsets: an overview and a case history, MISRA C" - *Information and Software Technology*, 2004, vol. 46, pp. 465 - 472.

[37] S. Heath, "Embedded Systems Design", Elsevier Science, 2002.

[38] H. Henderson, "Encyclopedia of computer science and technology", Infobase Publishing, 2009.

[39] "The final ISA showdown: Is ARM, x86, or MIPS intrinsically more power efficient?", http://www.extremetech.com/extreme/188396-the-final-isa-showdown-is-arm-x86-or-mips-intrinsically-more-power-efficient, 2014.11.02.

[40] B. Hughes and M. Cotterell, "Software Project Management", McGraw-Hill, 2006.

[41] "Use of Modern Light Sources in Traditional Lighthouse Optics", IALA, 2007.

[42] "Light Sources used in Visual Aids to Navigation", IALA, 2011.

[43] "Technical Reference: Personal Computer, Personal Computer Hardware Reference Library", IBM, 1984.

[44] "WebSphere Real Time", http://www.ibm.com/software/products/en/real-time, 2014.09.24.

[45] IEC, "61508 Functional safety of electrical/electronic/programmable electronic safety-related systems", The International Electrotechnical Commission, 1998.

[46] "MCS-48 Microcomputer User's Manual", Intel, 1978.

[47] "MCS 51 Microcontroller Family Users's Manual", Intel, 1994.

[48] T. Jamil, "RISC versus CISC" - *Potentials, IEEE*, 1995, vol. 14, pp. 13-16.

[49] J. V. Jerry Doland, "C Style Guide", NASA, 1994.

[50] "JPL Institutional Coding Standard for the C Programming Language", Jet Propulsion Laboratory, California Institute of Technology, 2009.

[51] "Ethernut Project", http://www.ethernut.de, 2014.09.22.

[52] "A Memory Allocator", http://gee.cs.oswego.edu/dl/html/malloc.html, 2014.06.01.

[53] "A byte-oriented AES-256 implementation",

http://www.literatecode.com/aes256, 2014.05.25.

[54] J.-L. Lions and others, "Ariane 5 flight 501 failure", 1996.

[55] "C++ Coding Standards for the System Development and Demonstration Program", Lockheed Martin Corporation, 2007.

[56] D. Markus, M. Kambiz and A. Nancy, "Software Defined Radio: Architectures, Systems and Functions", John Wiley & Sons, 2005.

[57] P. Marounek, "Simplified approach to effort estimation in software maintenance" - *Journal of systems integration*, 2012, vol. 3, pp. 51-63.

[58] R. C. Martin, "Clean Code: A handbook of agile software craftsmanship", Prentice Hall, 2009.

[59] M. Masmano, I. Ripoll, P. Balbastre and A. Crespo, "A constant-time dynamic storage allocator for real-time systems" - *Real-Time Systems*, 2008, vol. 40, pp. 149-179.

[60] S. McConnell, "Rapid Development: Taming Wild Software Schedules", Microsoft Press, 1996.

[61] S. Meyers, "Effective C++: 55 Specific Ways to Improve Your Programs and Designs", Pearson Education, 2005.

[62] "PIC32MX Family Data Sheet", Microchip, 2008.

[63] "Memory allocation strategies: a stack-like (LIFO) allocator", http://blog.molecular-matters.com/2012/08/27/memory-allocation-strategies-a-stack-like-lifo-allocator/, 2015.09.15.

[64] G. E. Moore, "Cramming More Components onto Integrated Circuits" - *Electronics*, 1965, vol. 38, pp. 114-117.

[65] E. Moorits and G. Jervan. "Low resource demanding FOTA method for remote AtoN site equipment" - *OCEANS 2010*, pp. 1-5, 2010.

[66] E. Moorits and G. Jervan. "Profiling in deeply embedded systems" - *Electronics Conference (BEC), 2012 13th Biennial Baltic*, pp. 127-130, 2012.

[67] E. Moorits and A. Usk. "A numerically efficient method for calculation of the angle of heel of a navigational buoy" - *Electronics Conference (BEC), 2010 12th Biennial Baltic*, pp. 357-360, 2010.

[68] E. Moorits and A. Usk. "Buoy collision detection" - *ELMAR, 2012 Proceedings*, pp. 109-112, 2012.

[69] E. Moorits, A. Usk and T. Kouts. "Wave height measurement as a secondary function of navigational buoys" - *OCEANS 2011*, pp. 1-5, 2011.

[70] "Microcomputer 3870/F8 Data Book", Mostek, 1978.

[71] Motor Industry Software Reliability Association and others, "MISRA-C: 2004 Guidelines for the Use of the C Language in Critical Systems", MIRA Limited, 2004.

[72] Motor Industry Software Reliability Association and others, "MISRA-C++: 2008 Guidelines for the Use of the C++ Language in Critical Systems", MIRA Limited, 2008.

[73] "M6804 MCU Manual", Motorola, 1984.

[74] "MC68(7)05P Series", Motorola, 1984.

[75] "MC6801 MC6803", Motorola, 1984.

[76] "MC68HC16Y1 16-Bit Modular Microcontroller", Motorola, 1992.

[77] "MC68HC08AB16A HCMOS Microcontroller Unit", Motorola, 2000.

[78] National Institute of Standards and Technology, "FIPS 197" - *National Institute of Standards and Technology, November*, 2001, vol. , pp. 1-51.

[79] O'Regan, G. "History of Programming Languages". In: (Ed.), *A Brief History of Computing*, Springer London, 2012.

[80] R. Osherove, "The Art of Unit Testing: With Examples in .Net", Manning Publications Co., 2009.

[81] "AVR-Crypto-Lib/en", http://www.das-labor.org/wiki/AVR-Crypto-Lib/en, 2014.05.25.

[82] B. O'Connor, "NASA Software Safety Guidebook", NASA, 2004.

[83] "Python Programming Language – Official Website", http://www.python.org, 2014.09.24.

[84] "Open On-Chip Debugger", http://openocd.sourceforge.net, 2014.09.27.

[85] "FreeRTOS", http://www.freertos.org, 2014.11.30.

[86] "Unexpected trends", http://www.embedded.com/electronics-blogs/programming-pointers/4372180/Unexpected-trends, 2014.09.24.

[87] F. Siebert. "The impact of realtime garbage collection on realtime Java programming" - *Object-Oriented Real-Time Distributed Computing, 2004. Proceedings. Seventh IEEE International Symposium on*, pp. 33-40, 2004.

[88] "Opencryptotoken - Atmel's AVR based usb crypto device using Elliptic Curves Cryptography", https://code.google.com/p/opencryptotoken, 2014.01.26.

[89] "When was C++ invented?", http://www.stroustrup.com/bs_faq.html#invention, 2014.09.24.

[90] C. R. Symons, "Function point analysis: difficulties and improvements" - *Software Engineering, IEEE Transactions on*, 1988, vol. 14, pp. 2-11.

[91] "GM862 Product Description", Telit Communications S.p.A., 2006.

[92] "TMS1000 Series Data Manual", Texas Instruments, 1976.

[93] "MSP430x1xx Family Users's Guide", Texas Instruments, 2006.

[94] "TI-RTOS: Real-Time Operating System (RTOS)", http://www.ti.com/tool/ti-rtos, 2014.09.27.

[95] "64-Bit MIPS-Based Microcontroller With PCI Interface", http://www.toshiba.co.uk/innovation/jsp/news.do?service=UK&year=NONE&ID=00000005a4, 2014.09.20.

[96] "Product Brief TMPR4927ATB–200 (TX4927) 64-Bit RISC Processor", Toshiba, 2003.

[97] "Scaled CMOS technology reliability users guide", M. White, 2010.

[98] "SafeRTOS", http://www.freertos.org/FreeRTOS-Plus/Safety_Critical_Certified/SafeRTOS.shtml, 2014.09.27.

[99] "AVR Libc", http://www.nongnu.org/avr-libc, 2014.11.02.

[100] "diet libc", http://www.fefe.de/dietlibc, 2014.11.02.

[101] "eCos", http://ecos.sourceware.org, 2014.11.30.

[102] "Embedded GLIBC (EGLIBC)", http://www.eglibc.org, 2014.04.19.

[103] "GCC toolchain for MSP430", http://sourceforge.net/projects/mspgcc/files/msp430-libc, 2014.04.19.

[104] "Newlib", http://sourceware.org/newlib, 2014.04.19.

[105] "PJSIP", http://www.pjsip.org, 2014.11.30.

[106] "Uclibc", http://www.uclibc.org, 2014.04.19.

[107] "Pharos Marine Automatic Power Inc.", http://www.automaticpower.com, 2015.03.22.

[108] "Sabik", http://www.sabik.com, 2015.03.22.

[109] "Sealite", http://www.sealite.com.au, 2015.03.22.

[110] "Splint", http://www.splint.org, 2015.10.16.

[111] "SRT Marine Technology", http://www.srt-marine.com, 2015.04.24.

[112] "Tideland", http://www.tidelandsignal.com, 2015.03.22.

[113] "Zeni Lite Buoy Co., Ltd", http://www.zenilite.co.jp/english, 2015.03.22.

ACKNOWLEDGEMENTS

This thesis is based on my work done during the last ten years in the field of marine navigation light systems.

I would like to express my gratitude to my supervisors Prof. Gert Jervan and Aivar Usk for the valuable guidance during my studies. I also wish to thank all my colleagues in Cybernetica AS for support and help.

Finally, I want to thank my family for the encouragement and support that helped me though the difficult task of writing this thesis.

This work were supported by European Social Fund's Doctoral Studies and Internationalisation Programme DoRa, which is carried out by Foundation Archimedes, Tiger University Program of the Information Technology Foundation for Education and the Estonian Doctoral School in Information and Communication Technology.

Erkki Moorits,
Tallinn, May 2016

ABSTRACT

In recent decades, along with the development of microcontrollers, embedded systems are increasingly frequently used in areas designed for long-term autonomous operation. In addition, many of these systems are installed in hard to access areas for support personnel and also require very low power consumption. This thesis handles the software component of the new generation marine navigation light systems, problems that araised during the development of the embedded software, solutions for these problems and also tools and methods for testing embedded software.

The main objectives of the thesis are methods, technical solutions, and recommendations of using existing embedded software developement methods for development highly constrained embedded systems. In this thesis are described methods and technical solutions, which are used in the marine navigation light systems. These methods have low memory and processor resources consumption, which is in many cases more important than accuracy of the mathematical functions. This thesis also handles briefly programming languages, which suits for embedded system development. Also are described different ways to develop automatically testable embedded software and are presented methods which allows to simplify testing of embedded systems, including using automated tests with continuous integration servers. The thesis provides recommendations and discuss disadvantages of various schedulers as well as proposes the preferred scheduler for small embedded systems. Both co-operative and preemptive schedulers are discussed, also was pointed out the possible performance and memory bottlenecks, which have influence on smaller embedded systems. Various approaches to resetting the watchdog resulting from the characteristics of multitasking programs are presented. Described methods are suitable for different schedulers and program structures. Optimisations for function calls, effectively changing the sequence of bytes, and some recommendations of using the AES cryptographic functions were given. Optimised functions are mainly targeted for smaller embedded systems allowing to reduce the use of memory and CPU consumption. An alternative approach to the use of dynamic memory, which is mostly designed for using in smaller embedded systems is presented, this approach can also be used in larger computers as well. The developed solution also allows to take into account the memory requirements when compiling the program.

All of the above-mentioned methods, techniques and solutions have been applied in AS Cybernetica marine navigation light systems.

KOKKUVÕTE

Viimastel kümnenditel on koos mikrokontrollerite arenguga hakatud erinevaid sardsüsteeme kasutama ka sellistes kohtades kus on ette nähtud ilma tugiisikute sekkumiseta pikemaajaline autonoomne töö. Lisaks, on veel ka paljud sellised süsteemid paigutatud kohtadesse, mis on teenindavale personaalile raskesti ligipääsetavad ja samas nõuavad ka väga väikest energiatarvet. Käesolev väitekiri käsitlebki uue generatsiooni AtoN seadmete tarkvara osa, tarkvara ja lisafunktsioonide loomisel tekkinud probleeme ja sobilikke lahendusi ning tarkvara loomisel ja testimisel kasutatavaid tööriistu ja meetodeid.

Väitekirja peamisteks väljunditeks on meetodid, tehnilised lahendused ja soovitused olemasolevate sardtarkvara arendamise meetodite ja praktikate kasutamiseks piiratud võimalustega sardsüsteemides. Kõikide kirjeldatud meetodite juures on oluline see, et nende mälu tarbimine oleks minimaalne ja samas võtaksid ka minimaalselt protsessori ressurssi, mis on ka paljudel juhtudel olulisem kui matemaatiliste funktsioonide täpsus. Töös on välja toodud sobilikud programmerimiskeeled, kirjeldatud erinevaid lähenemisi ja programmi struktuure mis võimaldavad lihtsustada sardsüsteemides automaatset testimist, mis omakorda annab võimaluse kasutada automatiseeritud teste koos pideva integratsiooni serveritega. On toodud soovitused erinevate planeerijate kasutamiseks ja ka puudused mis võivad avaldada mõju programmide ülesehitusele. Planeerijate juures on käsitletud co-operative ja preemptive planeerijaid, arvestades seejuures ka võimaliku jõudluse ja mälu vajaduse piiranguid. Kirjeldatud multitegur programmide eripäradest tulenevad lähenemisi valvetaimeri nullimisele, töös on ka kirjeldatud meetodeid valvetaimeri nullimiseks mis on sobilikud kasutamiseks erinevate planeerijatega ja programmi struktuuridega. Kirjeldatud peamiselt väiksemate sardsüsteemide spetsiifilised optimeerimised funktsioonide väljakutsumisele, efektiivsemaid meetodideid baitide järjekorra manipuleerimieks ja mõned soovitused AES krüptograafiliste funktsioonide kasutamisel. Optimeeritud funktsioonid võimaldavad vähendada mälu või protsessori ressursi kasutamist. On ka näidatud alternatiivne lähenemine dünaamilise mälu hõivamisele ja vabastamisele. Näidatud lahendus on eelkõige mõeldud kasutamiseks väiksematel sardsüteemidel, kuid sobilik kasutada ka suurematel arvutitel. Väljatöötatud lahendus võimaldab võtta arvesse ka vajaliku mälu suurust juba programmi kompileerimisel.

Kõik eelpool mainitud meetodid, tehnikad ja lahendused on realiseeritud AS Cybernetica mere navigatsioonitulesüsteemides.

APPENDIX 1

E. Moorits, G. Jervan, "Low resource demanding FOTA method for remote AtoN site equipment", Proceedings of the OCEANS '10 MTS/IEEE Seattle, 2010, pp. 1 – 5.

Low Resource Demanding FOTA Method For Remote AtoN Site Equipment

Erkki Moorits[1,2], Gert Jervan[2]
erkki.moorits@cyber.ee, gert.jervan@pld.ttu.ee

[1]Department of Navigation Systems
Cybernetica AS
Tallinn, Estonia

[2]Department of Computer Engineering
Tallinn University of Technology
Tallinn, Estonia

***Abstract* — This paper presents a method for firmware update in memory constrained low-power controllers used in marine aids to navigation (AtoN) and telematics systems. The developed method allows carrying out firmware updates regardless of the communication channel used. This approach differs from other similar methods mainly by its low requirements to hardware and high flexibility; hence it is applicable to relatively small microcontrollers. The paper is concluded with experimental results performed on operational marine buoys.**

I. Introduction

Like many recently developed embedded products, flashers and telematics modules employed in marine AtoN systems are relatively complicated devices with complex firmware. For example, a typical navigational buoy or a lighthouse is equipped with at least two microcontrollers (in addition to modems). Depending on the application specifics, each microcontroller may have quite a complex program, typically between 20 kB and 100 kB of program code which is roughly 7,000 to 40,000 lines of code. Also it is not uncommon that functional requirements to firmware of already deployed units change over the time, necessitating several updates during product's usable lifetime. In some rare cases, the already deployed firmware may need to be updated due to programming errors in some functions that have passed the initial testing and surface only when certain set of conditions appear. Usually, in such cases it is required to undertake a field trip either to retrieve a controller to the depot, or to perform the firmware upgrading process at the remote site since direct physical connection to the system is needed. This becomes costly in case of systems which have numerous devices distributed over a wide geographic area, or even if a few devices are located at places not easily accessible – like marine buoys. A solution for such cases (for devices capable to communicate wirelessly) is to use firmware update over the air (FOTA). Currently, firmware updates over the air are mainly used in automobiles, cellular phones and various smart sensor solutions. Taking into account the constantly decreasing mobile data communication prices (for example GPRS, 3G and beyond), firmware updating over the air is clearly the most cost effective method for AtoN and telematics systems.

A wide range of FOTA capable systems exist, mainly in mobile communication [1], [2] and automotive electronics [3] domains. The latter systems are somewhat similar to marine AtoN systems – all modern cars have at least one internal network and several devices are connected to this local network. But in-car electronic systems usually differ from marine AtoN systems by having significantly higher power consumption. FOTA applications for automotive electronics and mobile phones can typically update firmware of fairly large microcontrollers as it is possible to use data compression and encryption. Unfortunately, these methods are much too resource demanding for low power AtoN controllers and cannot be used in autonomous marine AtoN systems. On the other hand, many memory and power efficient firmware updating methods have been recently developed for smart sensors and similar systems [4]. Most of these methods suffer from one drawback – they cannot update firmware on other devices connected to the remote controller over its local wired network (LAN). While marine AtoN systems typically consist of several programmable devices operating in a local wired network that lack the ability to communicate directly with a FOTA server, they would benefit from such indirect FOTA capability. Some firmware updating methods proposed in [4] would almost be suitable, but these approaches are tied too closely to certain specific smart sensor kernel or toolflow, thus not being suitable for our development environment.

In order to carry out firmware updating over the air for remote marine systems we developed a new method especially suitable to our AtoN systems. The method is designed to be reliable and enables over the air updates while providing also support for external devices connected to a local wired network of the remote AtoN, i.e. the method supports medium independent firmware updating. Beside medium independent firmware updating, another requirement for AtoN systems is power and memory efficiency (small memory footprint). In our telematics module the described method requires approximately 3.2 kB of program-memory and 280 bytes of RAM. Such low memory requirements make it suitable for small microcontrollers which have certain program sections where it is possible to use self-programming instructions.

It is also anticipated that the communication channel might not be fully reliable and may occasionally have high failure probability, i.e. it may not be possible to transfer the new

firmware in full during a single communication session. Therefore a fault tolerant procedure is foreseen, allowing transferring only certain program parts at a time. In case of transmission faults, or partially corrupted program memory, it is possible to save several firmware backups inside the controller's backup memory. Most notably, this firmware updating method with previously downloaded firmware image interrupts the normal operation of the system for up to 15 seconds only, performing all preparations in the background. Due to the properties above described, this method is well suited for application in mission critical AtoN systems, including buoy systems subject to synthetic Automatic Identification System (AIS) reporting, and lighthouse systems.

This paper highlights the design considerations which are used in our new firmware updating system. Section 2 gives a brief overview of our current system. The firmware updating method itself is described in section 3. Section 4 outlines the results of experimental testing that was performed. Section 5 gives an overview of the impact of firmware updating to the operation of a buoy tasked with synthetic AIS reporting mission, and finally, the concluding remarks are presented in Section 6.

II. Current System Overview

Embedded systems which are used in marine AtoN systems are often composed of several controllers which are connected to the local area network of the AtoN site while one of the controllers acts as a Telematics Module (TM) – a gateway to the Remote Control and Monitoring Systems (RCMS) central server. A typical marine AtoN system is presented in Fig. 1 where TM is a communication controller (network gateway), and C1, C2, …, Cn are internal controllers in charge for the AtoN site's mission. The main task of a telematics module is packet forwarding between wireless and local area interfaces; a TM may also be configured to fulfill some additional tasks like time synchronization with GPS or certain measurement tasks. Internal controllers C1, C2, …, Cn may be navigational lantern flashers, smart power supply system controllers, or certain measurement controllers.

Since data communication is the slowest phase of the firmware updating process and the firmware updating should not interrupt the normal operation of a marine AtoN (typically, a buoy or a lighthouse) for a significant period of time, then the only possible way to minimize the interrupts caused by firmware updating would be to buffer the new firmware into external buffer or backup memory. For fast firmware copying, it is necessary that each controller shall have such a buffer memory since the internal AtoN network can be either too slow or have too high probability of errors to load the firmware directly into the program memory of a microcontroller connected to the site LAN. Firmware buffering must not interfere with the normal operation of the system. Figure 2 presents a block diagram of such a node which allows firmware buffering. The MCU (microcontroller unit) is an 8-bit Atmel AVR family (ATmega1280) microcontroller that has program memory in two sections – a boot section and an application section – where the application section is writable from the boot section, but the boot section is writable only from the boot section itself. In our current design, the boot section is a read-only section. CI1 and CI2 are communication interfaces – CI1 is a GR64 GSM/GPRS modem with integrated TCP/IP stack and CI2 is an RS-485 interface. Currently, the interfaces C1 and C2 are nearly equal – all commands can be passed from one interface to another and all commands are in the same format.

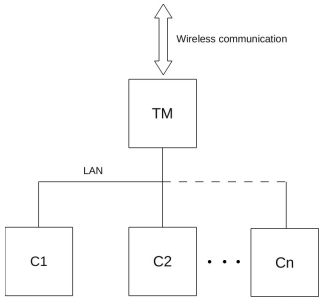

Figure 1: AtoN internal network

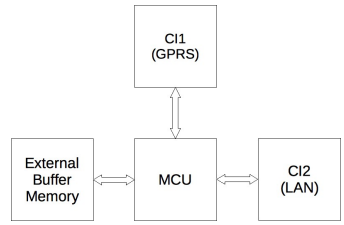

Figure 2: Telematics module

In systems with significant computational power and memory, the most widely spread communication protocol is TCP/IP – typically incoming connections are TCP/IP based and the LAN may also be TCP/IP based. Although it is possible to run TCP/IP stack in 8 bit microcontrollers then in most of the cases it is too resource demanding for low power microcontrollers. Therefore, it is preferable to use some lightweight communication protocol on such 8-bit devices. In our communication controller it is used two level communication stack where the second level is common for LAN and GSM/GPRS and the first level is GSM/GPRS specific. In the LAN side the first level is underlying on RS-485 link and the GSM/GPRS is underlying on modem integrated TCP/IP link. All commands on the second level are the same.

Figure 2 presents a telematics module that can store several different working firmware versions to external memory, including a backup version which may have minimal functionality but has been exhaustively tested. For reliable firmware updating, controller's ability to hold a previous and a backup firmware version in external memory is essential. In

the worst case where communication fails or firmware corrupts during transmission it is always possible to roll back to a previous or a backup firmware version. Firmware corruption during communication is quite rare on GSM/GPRS link but quite common on LAN, which has our proprietary protocol on RS-485 link. Minimal capacity of external memory should be not less than twice the size of the writable program memory of the controller, but the optimal size is three times the controller's memory size. This external buffer memory is a prerequisite for our firmware updating method.

Inside the microcontroller all program code is situated in the boot section and strictly isolated from any other program code. The boot section also holds a self contained program which is responsible for copying the firmware from the external memory into the controller program memory.

III. Firmware Updating

Our firmware updating method is basically a three step process:

1. First, the new firmware image is transferred over a GSM/GPRS link or over LAN to the external memory of the controller.
2. After successful firmware loading to the external memory a firmware copying program is started. This program will verify the 32-bit checksum of the firmware in the external memory and after successful checksum verification initiates loading of the new firmware into the program memory of the controller.
3. The main program of the controller is started after successful firmware loading into program memory. Once the firmware update is successfully completed, a self-test function is started. In a situation where the self-test fails, the old or backup firmware is loaded back. If the self-test succeeded, the main program clears the self-test flags and the controller starts operating using the new firmware.

As was mentioned earlier our system has common command format for all interfaces. For firmware updating this is important as commands should not depend on what communication interface is used during the updating process. Such common command acceptance is needed for the firmware updating service to be capable of extension to any compatible locally connected devices. Should the commands or command formats be different for each interface then the telematics module would need to be equipped with command translation capability which may become overly resource demanding. A common command format makes also the whole system design simpler.

While the external memory should store several different firmware versions, it is necessary to use a strictly specified memory layout. Since the external memory can be accessed only from special interfaces that all have their own read and write commands, it does not need to have any special file system. Without any file system utilized, it is only necessary to segment the memory properly. Figure 3 presents an external memory layout with all segments of same size as the

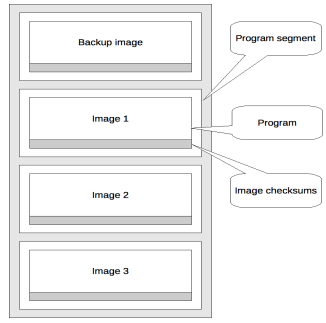

Figure 3: Memory allocation in external memory

controller's program memory. Since it is not possible to write to the boot loader section of the microcontroller, the area in external memory that contains the boot loader section addresses does not need to hold any program code and therefore can hold image block checksum values. Block checksums are necessary to track corrupted firmware, and in partial update mode, non matching parts in external memory. The difference between two checksums is that block checksums are taken from a block of 256 bytes while the 32-bit checksum is taken for whole program image, excluding the block checksum area. Block checksums are optional in the loading process; the only purpose of those checksums is aiding the detection of corrupt firmware part locations in external memory, or indicating mismatching parts between two different firmware versions. Block checksums are calculated by the controller after special calculation request from the FOTA server. An incorrect block checksum itself does not interrupt the firmware loading process.

The first step of firmware updating process is to load the new firmware image into the external memory, which can be performed by loading fragments of code into specified memory regions, i.e. loading only updates, or by transfer of a new image in full. New loadable firmware image is generated by GNU Binutils, and may be either in binary, IHEX, SREC, or preferably, in the ELF format, but all formats must contain 32-bit firmware checksum information. Firmware loading can take place either using a wired connection over LAN, or a wireless (GSM/GPRS) connection. Since the firmware updating method supports both partial and full firmware uploading modes, it is reasonable to use partial firmware upload mode when communication link failures exceeding a single session break are expected during firmware updating. Expected communication link failures in one session can be found as follows:

$$p_{fail} \times d_s \geq 1 \quad (1)$$

where p_{fail} is byte failure probability e.g. one failure per ten kilobytes, and d_s is transferred data size. In case of deployed devices it is reasonable to transfer the firmware over GSM/GPRS in one continuous part during one continuous

session, while over LAN it can be accomplished by transferring several parts in one or several sessions. Both transfer methods are the same for the controller side, but in case of using the partial update mode the server has to ask for block checksums, to verify these, and to start a new transfer in a case a faulty block checksum was detected. Also, both transfer methods do not inflict any interference upon the operation of the whole remote AtoN site system.

Two alternatives are available for tracking faulty firmware: using block checksums and whole program memory checksum (32-bit checksum). While it is possible to use block checksums to track faulty firmware data, it is not necessary to compute block checksums after transfer when the new firmware has been transferred within one session. In such case, it is more efficient to make the controller to calculate a single long checksum and if this checksum does not match to re-initiate the transfer of the new image in whole. In case of partial image loading, it is necessary to check block checksums after each 256 byte block is transferred; in case that a controller computed block checksum does not match with a block checksum computed at the server side, retransfer of only the faulty block of the new firmware image is required.

Once the firmware transfer into external memory is completed, a non-critical moment from the system mission's point of view is awaited to start up a small firmware copying program which will copy the new firmware from the external memory into the program memory of the microcontroller as fast as possible. This firmware copying program is basically part of the boot loader, being responsible for correct firmware copying from external memory to microcontroller's program memory. The copying program also has two EEPROM regions which are designated for signaling system self-test functions when the firmware is updated. One of the EEPROM regions holds a successive firmware update counter and another region holds a new firmware and a faulty checksum flag. Before the firmware copying process is started, the 32-bit firmware checksums are checked; if these do not match, a faulty checksum flag is set and a system reset is initiated. If checksums are correct then firmware copying from the external memory to microcontroller program memory is started. After the firmware copying process is completed, the successive firmware update counter is incremented and the new firmware flag is set. Thereafter a system reset is initiated, starting up the operation with newly loaded firmware.

As soon as the new firmware is started up, a check of the flags set is performed. In case when the new firmware flag is set, a self-test shall be initiated. If the new firmware does not succeed completing the self-test within a specified timeframe, the operation is discontinued and a previous or backup version of the firmware is loaded back into the program memory of the controller. In case that a faulty checksum flag is set, the controller sends a failure indication message to the FOTA server in the first communication session and no self-test is started. The firmware copying program with successive firmware update counter and new firmware flag can also track cases where new firmware is unable to operate at all and is therefore immediately terminated by a watchdog (WD) reset. When more than three WD resets occur, the old or backup firmware will be loaded back into the controller program memory. The successive firmware update counter and new the firmware flag are both cleared when the controller has successfully completed the self-test.

IV. Experimental Results

The above described method has been successfully implemented in our AtoN telematics module. Since the new firmware updating method must be compatible with our legacy boot loader software, it contains a quite large portion of the wired boot loader program code. Despite the compatibility with our legacy boot loader, the new boot loader with firmware copying code needs only 3.2 kB of program memory, and approximately 280 bytes of RAM to operate. The firmware copying code itself needs approximately 1 kB of program memory.

The laboratory tests with our new boot loader and FOTA server showed that the previously described method can copy firmware from the external buffer memory to main memory in about 8 seconds, and the interruption of the main program operation lasts less than 15 seconds. Firmware update was tried with two different remote control and monitoring system (RCMS) servers – our test server and an actual AtoN RCMS server. The following test results are taken from the test server; the only difference between a test server and a full RCMS server was the upload speed where the test server was roughly two times faster. Firmware updating over the GSM/GPRS data link was tested with two different firmware images: one of 71.634 kB of program code, and another of 79.021 kB of program code. Firmware loading over GSM/GPRS data link to the controller's external memory is quite slow: loading of a 71.634 kB firmware image takes typically 52 sec to 56 sec, and loading of a 79.021 kB image 55 sec to 60 sec. Such a low firmware downloading speed is mainly caused by slow internal connections between the external buffer memory, the microcontroller, and its GSM/GPRS modem. In theory, it is possible to increase the downloading speed, but this is not practical.

In the laboratory, roughly 95% of firmware transferring attempts into the external memory over GSM/GPRS succeeded in the first attempt and 5% of the failures were largely caused by GSM/GPRS communication failures; all repeated attempts for firmware transferring were successful already at the second trial. Most GSM/GPRS failures were caused by network delays which were over 15 seconds long, resulting in server timeout.

Firmware updating was also tested on deployed devices within the operational AtoN infrastructure and as was anticipated, the results displayed slightly lower first-time success rate than in the laboratory: roughly 90% of firmware transferring attempts into the external memory succeeded in

the first attempt, 5% in the second attempt and remaining 5% of transfers succeeded within ten attempts. Transfer failures were caused by long transfer delays or connection loss, where both are quite common at remote AtoN sites operating in the conditions of low GSM field strength.

In addition to GSM/GPRS transmission trials, tests were carried out to investigate server side failures: the server was shut down during an active communication session, resulting in a new connection to the server to be established after a 15 seconds timeout; the new firmware was downloaded again without unwanted effects on the remote AtoN system operation.

Since the integrity of the firmware is protected by a fairly long checksum value, a faulty firmware image was never copied into controller's program memory. When the new firmware was found to be faulty, the controller never tried to copy it into the program memory; therefore in such cases the program memory remained unchanged. In case when both, the program memory and the new image were corrupt, the controller always copied the last-known-to-work or backup firmware back to program memory. When a faulty firmware image with a correct checksum value was copied to microcontroller's program memory, it could never pass the self-test and after a while a working firmware version was loaded back.

V. Firmware Updating in Synthetic AIS Reporting Mission

AtoN device firmware updating over the air may have certain impact on continuity of the synthetic AIS reporting mission of a buoy system. AtoN devices subject to synthetic AIS reporting are expected to broadcast their status information typically at a three minute interval while the data sent to the RCMS remain valid only for one minute. Therefore, it is necessary that an AtoN device in synthetic AIS configuration can update the firmware within about two minutes. Typically, an AtoN telematics module needs about 15 seconds for its measuring tasks and tests, but following a reset event it takes roughly another 30 seconds for registration into the GSM/GPRS communications network. If the firmware update and controller reset is completed within less than three minutes of time from the moment when the controller submitted its regular synthetic AIS report to the RCMS server, the firmware updating process will have no impact to the AIS mission, presenting a negligible impact to the availability of the navigational signal.

While our AtoN controller currently cannot send AtoN status information while receiving new firmware update from the RCMS server, a gap in forwarding the synthetic AIS messages into the AIS shore infrastructure for broadcasting is inevitable. Although such communication gap will be present during the firmware loading process, the AtoN controller will continue the operation with its regular tasks during this time. This communication gap can be avoided by two different methods: the first option would be a faster data transmission between the server and the telematics module, and the second option would be data transmission in smaller firmware parts which are transmitted over a longer period. The faster data transmission is in principle possible, but it requires some modifications in the firmware architecture of the existing controller, and is currently not practical. The second option where the firmware is transmitted in several parts is currently possible on controller side, but would require several modifications in the existing RCMS server software.

VI. Conclusion

The objective of the current work was to develop a reliable method for remote firmware updating in embedded AtoN controllers with minimum impact on operational availability. The developed method differs from other similar methods mainly by short program interruption time, which is very important in case of synthetic AIS reporting mission of targeted telematics modules; the above described firmware updating method has negligible impact upon AIS mission. Furthermore, it enables updating of the firmware even in programmable equipment units with the AtoN site system that are connected to the local area network of the AtoN site. This firmware updating method also features a roll back capability allowing reverting to previous known-to-work firmware in case of transmission faults of the firmware, or partially corrupting program memory. The method has been successfully implemented and deployed in our buoy telematics electronic systems with all firmware tests both in the laboratory conditions and at the sea environment successful.

References

[1] Innopath, "Understanding Firmware over the Air-FOTA", http://www.innopath.com/pdf/fota.pdf, 2010

[2] Red Bend, "Firmware Updates", http://www.redbend.com/solutions/firmware-updates.asp, 2010

[3] Moshe Shavit, Andrew J. Gryc, Radovan Miucic. "Firmware Update Over The Air (FOTA) for Automotive Industry". Technical Report 2007-01-3523, SAE 2007

[4] Jonathan Hui, "Deluge: TinyOS Network Programming", http://www.cs.berkeley.edu/~jwhui/deluge/index.html, 2010

APPENDIX 2

E. Moorits, A. Usk, "A Numerically Efficient Method for Calculation of the Angle of Heel of a Navigational Buoy", Proceedings of the 12th Biennial Baltic Electronic Conference BEC2010: 2010, pp. 357 – 360.

A Numerically Efficient Method for Calculation of the Angle of Heel of a Navigational Buoy

E. Moorits[1], A. Usk[2]

[1]*Cybernetica AS, 12618 Tallinn, Estonia, E-mail: erkki.moorits@cyber.ee*
[2]*Cybernetica AS, 12618 Tallinn, Estonia, E-mail: aivar.usk@cyber.ee*

ABSTRACT: This paper presents a numerically efficient method developed for obtaining heel angle information on navigational buoys by the use of onboard low power embedded controllers equipped with solid state acceleration sensors, focusing on the signal processing principles employed. Calculation of the buoy heel (tilt angle or inclination) is based on continuous measurement of acceleration of the buoy in all three planes of movement, accomplished using a 3-axial solid state accelerometer (g-sensor) with the maximum range of ±3 g. The sensor is integrated with an Aid to Navigation (AtoN) telematics module that is subject to low power consumption requirements and size restrictions resulting in limited computational capability. Results of tests performed on operational marine buoys are presented at the end of the article.

1 Introduction

Despite the widespread use of mature electronic technologies for marine navigation, visual light navigation stations remain an indispensable part of marine navigation safety infrastructure for foreseeable future. While the navigational buoys are widely used for generating light signals of strictly specified visibility range and flashing character, the environmental effects like wave action, winds, tidal currents and ice may adversely affect both by tilting the buoy, introducing either a dynamic or a static heel angle, or a combination of both. This may introduce significant reduction of the visibility range and parasitic modulation of the light signal as described in [1], therefore timely awareness of the relevant authorities of the typical and critical angles of heel of deployed navigational buoys can be considered a precondition for provision of a high quality light navigation service. In addition, specifics of the Nordic region introduce a problem that goes undetectable by a traditional AtoN remote monitoring system: a buoy may be on station (within the limits of the assigned and GPS monitored geographical position), but stuck in the ice at a significant heel angle making the navigation light to fail in most of directions (Figure 1). The above situations can introduce significant navigational hazards due to the fact that buoy lanterns often use rather narrow vertical beam profiles (divergence) providing only five to fifteen degrees full width at half of maximum intensity.

Figure 1: Buoy in Ice

Most of such buoy light signal visibility problems could be avoided by using buoy lanterns with a light source that is either always horizontally aligned by mechanical means (gimballed), or equipped with a light source and optics that guarantee a sufficient vertical beam width for any environmental conditions encountered. In practice, such solutions are too expensive to manufacture, too bulky and power consuming for most applications. With many buoys at critical stations equipped with remote monitoring (telematics) equipment featuring low power embedded controllers, a feasible alternative to reducing the navigational risks associated with excessive buoy heel angles is tilt monitoring in conjunction with reporting of critical heel angles to the operations centre, allowing to issue navigational warnings and to take appropriate action as necessary. The same setup provides efficient means for service quality control (statistics) as well as for researching the behaviour of specific buoys deployed at specific locations in order to determine sufficiency of the stability provided by the buoy platform selected, and to decide upon required minimum vertical divergence of the buoy lantern to be employed.

The most cost effective of contemporary methods for inclination measurement is achieved by application of a programmable microcontroller equipped with a three-axial micromechanical accelerometer sensor (a solid state g-sensor – a Micro-Electro-Mechanical System (MEMS)) that can measure acceleration levels on all three axes simultaneously, including the static component of gravitational acceleration distributed over the sensor axes

depending on inclination of the controller carrying the sensor. Similar angle detection methods have been developed for directional drilling systems [2] and monitoring of patient's head position during a post-operative period after vitreoretinal surgery [3]. Some MEMS g-sensor manufacturers have published certain application notes describing tilt measurement and calculation [4], but none of the published material reviewed offered angle calculation methods suitable for dynamic environment. In addition, most of the suggested methods require powerful microcontrollers for implementation of complex algorithms while implementing of autonomous heel angle calculation capability onboard a buoy is only feasible when using a simple algorithm.

The rest of this paper is organized as follows: Section 2 provides a brief overview of the developed heel angle measurement system; Sections 3 and 4 describe the proposed heel angle calculation method; Section 5 outlines results of experiments and tests performed in both laboratory and marine environment; the concluding remarks are given in Section 6.

2 Remote Monitoring and Acceleration Measurement System Overview

While AtoN remote control and monitoring systems (RCMS) of varying degrees of sophistication have been around for decades, measurement of buoy heel angles has not been widely used due to complexity and cost - in an autonomous system that needs to provide reliable operation from primary batteries for years, spending of every mA of current must be well substantiated. Our concept foresaw integration of a single new hardware component (3-axial g-sensor) with the existing telematics module (TM) used for remote monitoring of navigational buoys, and accomplishment of heel angle calculation and monitoring tasks using the available ADC ports and spare computational capacity of the existing microcontroller. In a typical application, the TM is installed inside a protective enclosure together with a flasher module and an LED array, and mounted on a buoy superstructure, typically 2 to 4.5 meters above the sea level. Communication protocols of the TM that serves primarily as a communications gateway between the remote site equipment and the RCMS centre server were updated to accommodate heel angle information and associated alarms.

A TM is performing acceleration data acquisition in blocks where each block consists of three 10-bit acceleration measurement values representing acceleration levels sampled from three axes of the g-sensor. All samples in one block are separated from each other by 0.6ms in time. Block sampling period is user selectable, typically set to 100ms considering the dynamics of the buoy platform. Due to the fact that acceleration data acquisition is not the primary task for the TM, under certain circumstances the acceleration data blocks may be sampled at slightly uneven intervals due to coinciding higher priority tasks of the processor: at most 0.33% of blocks may be delayed by 10% to 20% of configured sampling interval. Due to the considerably slow movement of a buoy, with a typical buoy moving cycle between 2s and 10s, and the considerably high sampling rate used, such occasional uneven sampling does not have any significant detrimental impact on autonomous heel angle calculation.

Figure 2 shows relevant subsystems of a TM utilized for acceleration measurement, inclination angle calculation and status/alarm communication tasks, leaving out all parts which are not involved in the process. The MCU used is an 8-bit AVR microcontroller, performing analog to digital conversion of g-sensor output data, heel angle calculations, angle value monitoring, maintaining statistics and initiating communications when necessary. When a heel angle value exceeding a pre-configured level is detected, the MCU initiates a communications session with the RCMS centre server using the communications interface (CI) to report a critical heel angle. In the current system implementation, the CI is a GSM/GPRS modem with integrated TCP/IP stack. The acceleration sensor utilized is an ADXL330 by Analog Devices [5] which is connected directly to the analog input channels of the MCU, allowing reducing the power consumption compared to the situation where a smart digital g-sensor would be used.

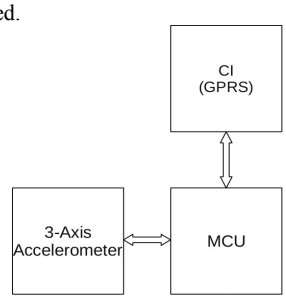

Figure 2: TM Block Diagram

It is still possible to calculate the heel angle at the server side, but this is practical only in special cases focusing on a more detailed research of buoy movements since transmission of raw acceleration data to the RCMS would be required. For example, in case of a 100ms sampling period, nearly 1MB of raw data has to be transmitted to the RCMS server hourly. In addition to direct communication costs, other factors limiting the utilization of server side data processing are increased power consumption of the TM due to the systematic transmissions over the open communication channel and, last but not least, inability of immediate local decision making due to the calculation results being available only at the remote operations centre.

3 Buoy Heel Angle Calculation

Calculation of the inclination angle based on digitized real-time acceleration data can be performed by using

simple trigonometric functions like sine or tangent. For systems that have hardware floating point support, the most elegant and easiest way would be to use tangent. In 8-bit embedded systems where all numbers have quite small range, the only feasible option is to use the sine function. 8-bit systems cannot use tangent because tangent have infinite value when the angle is 90 degrees, and it is very inefficient to use fixed point variables to store such values. When using the sine function for angle calculation, it is necessary to use an additional hypotenuse calculation; this is not a problem in all systems with sufficient available computational power. This additional calculation makes all data processing a little more resource consuming and may introduce small inaccuracies, making this approach not suitable for raw acceleration data processing on the server side.

It is possible to simplify the rest of calculations by finding the length of the vector in the X-Y plane, and using it as a single value describing the horizontal plane. This simplification is possible due to the absence of directional data in the horizontal plane of the buoy.

Length of the horizontal acceleration vector:

$$c=\sqrt{x^2+y^2} \tag{1}$$

where x and y are acceleration values from X and Y axis outputs of the sensor.

Sine function also needs the hypotenuse:

$$h=\sqrt{c^2+z^2} \tag{2}$$

where c is the length of the horizontal acceleration vector and z is acceleration value from the Z axis output of the sensor.

The inclination angle (buoy heel angle) can be calculated using the following formula:

$$\alpha=\arcsin\frac{c}{h} \tag{3}$$

In actual marine environment, the acceleration values obtained from the acceleration sensor are changing continuously which can cause certain errors in short term calculations. However, it is possible to perform a long term calculation that averages all input data and thus eliminates most errors caused by the continuously changing acceleration. Averaging is possible because buoy movement is mainly symmetrical to all axes and averaging provides a central value without short term excessive acceleration peaks.

4 Algorithm Implementation in Telematics Module Firmware

The most important limitation at using the inclination angle calculation algorithm directly in TM firmware is the absence of square root, hardware floating point support and fast trigonometric functions. These functions can only be used in the server side applications. Inclination angle calculation algorithm for TM is based on the following simplification that results in a metavariable that is directly proportional to the inclination angle, allowing taking actions at detecting certain threshold angles when necessary.

First, the squares of catheti and hypotenuse are calculated:

$$c^2=x^2+y^2 \tag{4}$$

$$h^2=c^2+z^2 \tag{5}$$

The inclination angle metavariable is calculated as follows:

$$\theta=2^{16}\cdot\frac{c^2}{h^2} \tag{6}$$

The result of formula 6 is a value that holds enough information to unambiguously determine the inclination angle of the buoy from the vertical axis. It can be averaged locally, or forwarded to the server side inside corresponding messages of the TM for use by other systems.

Due to the limitations of the TM hardware, it is possible to use only fixed point values; therefore, all values shall be in 16 bit range, which will cause some errors in our heel angle calculation. Hence the worst case accuracy δ is calculated as follows:

$$\theta=2^{16}\cdot(\sin\alpha)^2 \tag{7}$$

$$\delta=\left|\alpha-\arcsin\sqrt{\frac{\theta}{2^{16}}}\right| \tag{8}$$

where the variable θ is rounded down to a nearest integer.

Worst case accuracy can then be found by inserting angle values between 0º and 90º into formulas 7 and 8, resulting in heel angle calculation errors as shown in detail on graphs in Figure 3.

Resultantly, the worst case computational accuracy at determining the heel angle of a TM is 0.220º in the range of 0º to 2º and 88º to 90º, and 0.010º in the range of 2º to 88º.

Implementation of the average inclination algorithm in TM firmware is accomplished as follows. First, the average square of the catheti and the hypotenuse is calculated:

$$\bar{c}^2=\left(\frac{1}{n}\sum_{i=1}^{n}x_i\right)^2+\left(\frac{1}{n}\sum_{i=1}^{n}y_i\right)^2 \tag{9}$$

$$\bar{h}^2=\bar{c}^2+\left(\frac{1}{n}\sum_{i=1}^{n}z_i\right)^2 \tag{10}$$

The average inclination angle over the averaging period is calculated as shown in formula 11:

$$\theta=2^{16}\cdot\frac{\bar{c}^2}{\bar{h}^2}= =2^{16}\cdot\frac{\left(\frac{1}{n}\sum_{i=1}^{n}x_i\right)^2+\left(\frac{1}{n}\sum_{i=1}^{n}y_i\right)^2}{\left(\frac{1}{n}\sum_{i=1}^{n}x_i\right)^2+\left(\frac{1}{n}\sum_{i=1}^{n}y_i\right)^2+\left(\frac{1}{n}\sum_{i=1}^{n}z_i\right)^2}= =2^{16}\cdot\frac{\left(\sum_{i=1}^{n}x_i\right)^2+\left(\sum_{i=1}^{n}y_i\right)^2}{\left(\sum_{i=1}^{n}x_i\right)^2+\left(\sum_{i=1}^{n}y_i\right)^2+\left(\sum_{i=1}^{n}z_i\right)^2} \tag{11}$$

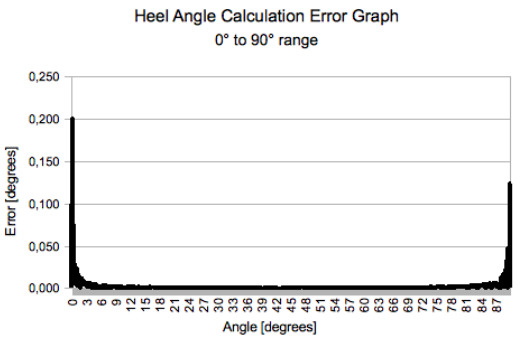

Figure 3: Heel angle calculation errors over the full range of 0° to 90°.

Decoding of instantaneous or average angle values from the metavariables received from a TM on the server side is accomplished as follows (the angle α is given in radians):

$$\alpha = \arcsin\sqrt{\frac{\theta}{2^{16}}} \qquad (12)$$

where θ is a coded angle metavariable received from a telematics module.

5 Test Results

Inclination angle calculation tests were carried out both on our in-house rotating test bench and on the navigational buoys deployed in actual marine environment. Table 1 presents the results of our laboratory tests.

Angle	Average angle, measured by controller	Error
0.0°	1.0°	1.0°
1.0°	0.7°	0.3°
2.0°	1.7°	0.3°
3.0°	2.3°	0.7°
5.0°	4.6°	0.4°
10.0°	9.6°	0.4°
20.0°	19.9°	0.1°
30.0°	30.2°	0.2°
45.0°	45.1°	0.1°

Table 1: Test results

As seen in Table 1, all values that are measured by TM are quite close to actual inclination angles of the TM; the average error over the tested range was below 0.5°. Only two cases exhibited larger errors – 0° and 3°, where the first one was caused by the controller and second error was caused by test bench, but both errors were below 1° which is acceptable for a device that is not intended for precise angle measurement.

In addition to laboratory tests, our method was verified in actual operational environment of the navigational buoys (Figure 1). The angle reported by TM based on autonomous calculations was consistent both with the visually identified buoy angle as well as the results of server side calculations based on raw acceleration data. When observing the static heel angle of a buoy frozen in an ice field over a longer period it is clearly seen that heel angle changes remain below one degree as expected (Figure 4).

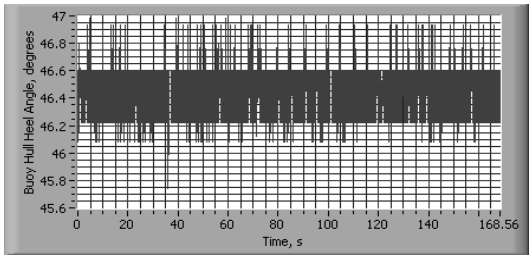

Figure 4

6 Conclusions

The objective of the current work was to develop a low resource demanding heel angle calculation method that is feasible for application on navigational buoys. The developed method is suitable for low power marine AtoN embedded systems with an integrated 3-axis acceleration sensor with analog output. This method is capable of carrying out heel angle calculations in real time with an error levels of up to 0.220° in the range 0°…2° and 88°…90° and 0.01° in the range 2°…88°. The method was successfully tested in our laboratory as well as in expected operational environment at the sea; all test results were in accordance with actual heel angles while the errors were neglectable considering the intended use.

References

[1] "IALA Guideline No. 1065 On Aids to Navigation Signal Light Beam Vertical Divergence." Edition 1, May 2009. Publication of the International Association of Marine Aids to Navigation and Lighthouse Authorities.

[2] Jian Kang, BoXiong Wang, ZhongXiang Hu, Rui Wang and Tao Liu, "Study of Drill Measuring System Based on MEMS Accelerative and Magnetoresistive Sensor". In Proc. 9th International Conference on Electronic Measurement & Instruments (ICEMI 2009, Beijing, Aug. 2009), pp 2-112 – 2-116.

[3] Jirí Dlouhý, Martin Cížek, Igor Vícha, Jirí Rozman, "MEMS Technology in Head Tilt Monitoring after Vitreoretinal Surgery". In Proc. : Radioelektronika, 2008 18th International Conference, pp 1 – 4.

[4] Freescale Semiconductor, "AN3107 - Measuring Tilt with Low-g Accelerometers", Rev 0, 05/2005

[5] Analog Devices, Inc. "ADXL330: Small, Low Power, 3-Axis ±3g iMEMS® Accelerometer", 2007

APPENDIX 3

E. Moorits, A. Usk, T. Kõuts, "Wave Height Measurement as a Secondary Function of Navigational Buoys", Proceedings of the OCEANS '11 MTS/IEEE Kona, 2011, pp. 1 – 5.

Wave Height Measurement as a Secondary Function of Navigational Buoys

Erkki Moorits[1], Aivar Usk[2]
Department of Navigation Systems
Cybernetica AS
Tallinn, Estonia
erkki.moorits@cyber.ee[1], aivar.usk@cyber.ee[2]

Tarmo Kõuts
Marine Systems Institute
Tallinn University of Technology
Tallinn, Estonia
tarmo.kouts@sea.ee

***Abstract* — This paper presents a method for measuring wave height with navigational buoys which are equipped with an acceleration sensor. The developed method differs from other similar methods by the ability to measure wave height with buoys which are not perfect wave followers – typical in case of navigational buoys. The paper summarizes the experimental study performed using operational marine navigational buoys as sources of wave data, in comparison with wave measurements performed with pressure based wave height and period gauges. Comparative measurements were made in variable forcing conditions and results show good agreement between those two datasets. Some differences that occur mainly during rapid changes of wave parameters, such as during build up and decay of the wave field, can be explained by physical properties of navigational buoys (shape and weight).**

Keywords: wave height measurement, navigational buoys

I. Introduction

Wave height is one of the key factors influencing the navigational conditions on any waterways. Wave field is a result of a complex set of factors; active, forcing factors (wind strength, direction and duration), and passive factors (depth profile, topography of the sea bottom, coastline configuration, etc). In fluid dynamics, wind-generated waves are surface waves that occur on the free surface of the water bodies.

Since the wave regime in a certain sea area can have remarkable spatial and temporal variability, in-situ wave height measurements with results made available to the mariners in real time are needed for the purposes of navigational decision support. Wave measurement with dedicated equipment is still costly while emerging satellite based solutions are not well suited for on-line provision of localized wave height information. At the same time, numerous navigational buoys are floating around in almost all navigable waters, waiting to be tasked with wave height measurement.

Main limitation at developing of technological solutions for enabling the navigational buoys with wave height measurement functionality is the restricted power availability: to avoid interfering with the primary mission of the buoys, any added equipment must draw insignificant amount of power in comparison with on-board AtoN light signaling, measurement and radio systems. Other issues involve platform specific calibration, data quality control and broadcasting methods.

II. System Overview

The key to cost efficiency of a wave measurement network based on navigational buoys lays in the fact that it utilizes the telematics equipment and GSM/GPRS cellular data links already present on the buoys for the purposes of aid to navigation (AtoN) remote control and monitoring. Targeted buoys are equipped with telematics modules with integrated three-axial accelerometers which are used as sensors for wave height measurements. In our case, the module had computational capabilities sufficient for acceleration based calculation of buoy heel angle. Designed for lowest achievable power consumption and certain AtoN specific communications tasks, the telematics module cannot offer sufficient data processing capability to perform complex computations of wave heights in-situ. Instead, it will periodically transmit buoy acceleration data to a shore side wave height calculation server which can broadcast the results to the mariners using the shore side Universal Automated Identification System (AIS) network. This research is ongoing, therefore it is too early to say whether a computationally more efficient wave height measurement algorithm could be developed for utilization directly on-board a navigational buoy using the current embedded hardware.

III. Wave Height Measurement Method

Wave measurement by the means of acceleration sensors is not brand new: the method was introduced in 1950-s and commercial products are available, e.g. Datawell Waverider. In our case the navigational buoys serve as measurement platforms – typical steel spar buoys with the weight of roughly 5 tons (Fig 1). These buoys are deployed for around the year operation, capable to withstand ice conditions. Chain moorings are used as standard, increasing overall buoy weight by 0.5 to 1.5 tons, and also keeping the buoys from riding the waves freely. Since the primary task of these buoys is to serve as a source of a navigation light signal, they are designed in a way allowing only limited wave following. Due to buoy hull design specifics and the mooring, each buoy has its own individual up and down movement period that does not depend significantly on the weather or wave motions: Figures 2 and 3 compare acceleration measurement results in calm and stormy weather that demonstrate matching wave periods. Figure 3 shows noticeable differences between two acceleration spectrum magnitude peaks, but the period of the higher peak remains almost the same with waves which are higher than 1.2 meters.

Figure 1: Steel spar buoy

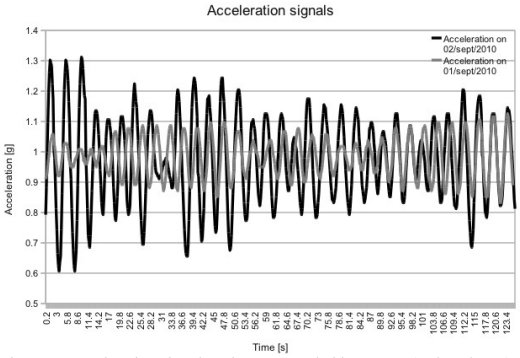

Figure 2: Acceleration signal packages recorded in stormy (02/sept/2010) and calm weather (01/sept/2010)

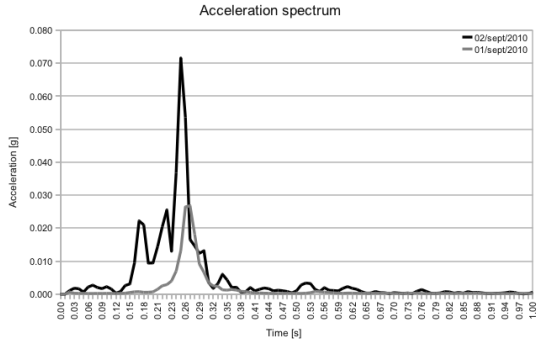

Figure 3: Acceleration spectrums calculated from data of stormy (02/sept/2010) and calm weather (01/sept/2010)

Therefore we can conclude that the navigational buoy platform used is not dynamic enough for successful utilization of common or straightforward wave height estimation algorithms (e.g. FFT based methods [1]).

To obtain wave height values, we have analyzed buoy acceleration data and found a good correlation between values of acceleration and wave heights measured nearby by reference sensor for this particular type of navigation buoy. During the pilot study we also noticed that behavior of different navigation buoys differ from place to place and that's not only because of differences in buoy shape and weight mentioned above, but also due to local peculiarities of wave field defined by bottom topography and local wind condition – which makes of course our task even more complicated. Method of calculation of the wave parameters based on the movements of a navigational buoy introduced here is based on finding the maximum acceleration values during certain time periods and correcting these with expected wave period value (buoy up-down movement period). Currently, the method provides significant wave height as output and we have compared obtained data with reference wave measurements.

The main problem for this method of wave height estimation lies in the filtering of the wave-induced buoy motions and leaving the buoy's own motion aside. To solve this problem and estimate wave heights from motions of navigational buoys, we have developed an empirical method described below.

In case of ideal waves and measuring equipment, most developed waves can be considered quite sine like with same maximum (crest) and minimum (through) values of acceleration. To find the amplitude of acceleration a_w in such simplest case, we only need to find the maximum and minimum values from the acceleration signal and subtract them from each other:

$$a_w = max(A) - min(A) \quad (1)$$

In the Equation 1, is parameter A an array of acceleration values. In this simple case we can also obtain maximum and minimum values just collecting highest and lowest signal values over a period of time in interest, which is a relatively simple operation for accomplishing in software.

With a constantly changing signal, finding of maximum and minimum values becomes a much more complex task. The main challenge is to minimize the weight of the values which are close to the maximum and minimum values. One possible solution in such case is to sort all accelerations in decreasing order and to construct an array of differences between maximum and minimum values (using the same method as in Equation 1) and averaging the highest N values.

$$a_w = \frac{1}{N} \sum_{i=1}^{N} \Delta A_i \quad (2)$$

where ΔA_i is array of decreasing differences between the maximum and minimum of acceleration data. The parameter N is found by trial and error, but when setting the parameter N correctly, one can obtain a rather precise result by such calculation. For example, in our collected data set we may have every 30th wave height as a real significant wave height, every wave is approximately 20 samples long and the acceleration signal from a real significant wave is two times higher than the acceleration signal caused by the normal buoy up and down movement. With a signal 30 wave periods long, we shall set N to 3, resulting in acceleration value which is 96.7% of real significant wave height acceleration value. This calculation

method has proven to provide statistically relevant results with acceleration signals where the dominating frequency is not the actual wave frequency but certain occasional jumps may carry the correct wave height information.

In order to calculate the significant wave height, we need to convert the acceleration values to wave height values. For the simplicity we can assume that the analysed waveform is a good approximation of a sinusoidal wave. This allows to calculate the amplitude of the waveform using the double differentiation of sinusoidal acceleration function, resulting in the following formula:

$$D = \frac{a_w}{2\pi^2 f^2} \tag{3}$$

where a_w is the amplitude of the acceleration, f is the frequency of oscillation and D is the actual displacement.

To increase the accuracy of measurements, we also need to consider the buoy up and down movement frequency to correct the calculation results. The fact that most navigational buoys are located in the area with vessel traffic may cause significant short term measurement errors due to the wakes of ships hitting the buoys; this is particularly noticeable in calm weather. In case of long term calculations, the wake waves do not change the buoy up and down movement period to such extent which would degrade the precision of wave measurement significantly. To take the buoy movement frequency into account, we need to perform FFT analysis of the acceleration data set and to insert the main up-down movement frequency into Equation 3.

Before the calculated wave heights are saved or forwarded to other software, each wave height measurement result is corrected using a look-up-table; if necessary, the wave heights will be further interpolated with cubic spline to 10-minute interval data segments and calculated a 2-hour running average wave height after the initial correction. The need for correction of the acceleration measurement results is due to the fact that the constant N in Equation 2 is based on average wave heights, but with this equation it may not result in correct wave heights when the waves are higher or lower than average. This correction could remove or at least reduce the caused effect by using double-pass wave height calculation, where in first pass the wave heights are calculated and a new N value is found, while in the second pass the correct wave heights are calculated using the new N value obtained in the first pass. The wave height compensation is carried out by using a look-up-table, where values are taken from reference measurements for a specific navigational buoy. The main drawback of this wave height measurement method is the need for reference measurement for every buoy type used.

Interpolation of wave heights to 10 minute periods is needed for two reasons: firstly, the acceleration data transmission is accomplished using GSM/GPRS radio network which may have quite many connection breaks during one measuring session in stormy weather. Secondly, the acceleration measurement is not a priority task for the onboard electronic system of a navigational buoy and therefore it may become interrupted any time, thus the measuring sessions may be of unequal duration. In both cases the data transmission periods are of unpredictable length and therefore we could not calculate any average wave height without using interpolation.

IV. Tests

In order to develop the wave analysis algorithm and validate the obtained wave data, the Estonian Maritime Administration, Cybernetica AS and the Marine Systems Institute at Tallinn University of Technology have performed trials since late 2008 to establish feasibility of such wave height measurement network based on navigational buoys. Even if navigation buoys are not ideal wave following platforms, it is still possible to calculate a rather close approximation of the actual wave height based on their acceleration. Tests and validation of the wave height estimation method were performed in five reference measurement sessions in three different locations, each lasting at least two weeks. In all cases the reference sensor was deployed at a distance less than 3 nautical miles from the buoys under testing (Fig. 4). Pressure based wave gauge was used for reference measurements performed by the Marine Systems Institute at Tallinn University of Technology.

Two wave recorders were used during experiments, the working principle of which is based on measurement of pressure at fixed position of the probe with absolute pressure sensor (Keller Ltd.). Anchored instruments were deployed 5 to 8m below sea surface and the measured pressure is converted into height of water column with 4Hz sampling rate, while water temperature variations are automatically compensated by

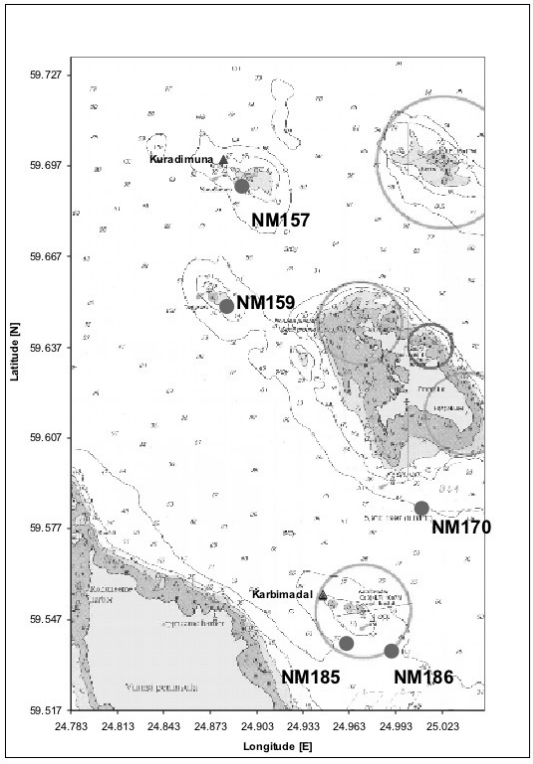

Figure 4: Location of navigational buoys hosting the acceleration sensors used in the wave parameter measurement experiment, and pressure based wave measurement equipment used for reference measurements shown with triangles.

sensor electronics. All data as raw pressure values are recorded on internal memory (an SD type card). Wave parameters are calculated from raw pressures after return of the recorder to shore and readout of the data from the instrument. This instrument has proved itself well in the past, most important raw data for wave calculation is available and if needed, several different methods of wave calculation could be used. In our case here the hydrostatic pressure is measured and following conversation procedure is applied to get wave parameters out of raw data series [2]. Sub-surface pressure transducers measure the instantaneous pressure that is the sum of air pressure, hydrostatic pressure and wave-induced dynamic pressure. If air pressure and hydrostatic pressure are assumed to remain constant at least during the wave period, the dynamic pressure under water is expressed with equations derived from the linear wave theory [3].

That pressure is a function of three parameters: the height of the pressure sensor from the seabed, wave frequency, and water depth. At an intermediate water depth, the pressure decreases hyperbolically with depth, therefore a sub-surface attenuation coefficient has to be applied in order to get a realistic picture of wave height.

First the pressure time-series (units of pressure) are converted to a subsurface elevation time series (units of height). Then the time series is divided into five-minute sections called wave packets. Additionally, the packets are de-averaged and de-trended. The mean value is used in order to calculate gauge depth, which is needed for the calculation of the attenuation coefficient. Further on, the power spectral density is estimated by using the Welch method, and a Hanning window is applied to smoothen the spectrum. The obtained subsurface elevation spectra $Ss\eta$ are converted to surface elevation spectra (S_η) using the linear wave theory:

$$S_\eta = Ss\eta \left(\frac{\cosh(kd)}{\cosh(k(d+z))} \right)^2 \quad (4)$$

with k denoting the wave-number calculated from the linear dispersion equation, d water depth, and z elevation of the pressure gauge relative to the mean water surface (negative downwards).

From the surface elevation spectrum, two important characteristics are derived: significant wave height and the period corresponding to the first moment of the spectrum. Significant wave height is defined as follows:

$$H_s = 4\sqrt{\int S_\eta(f)\,df} \quad (5)$$

The period corresponding to the first moment reads:

$$T_{01} = \frac{\int S_\eta(f)\,df}{\int f S_\eta(f)\,df} \quad (6)$$

Time series of measured wave parameters were conditioned same way for each of the measurement locations and periods, stored in ASCII files and used for further analysis and comparison with wave parameters from navigational buoys.

Results of the comparison of two datasets are good. Measurement periods captured different wind conditions and wave field realizations. Two datasets fit with each other very well for waves below 2m, 95% of the resulting wave heights differed from the reference wave heights by less than 41cm. In case of wave heights of over 2m, the maximum difference was 86 cm (Table 1 and figures 5-8), although the number of such larger wave heights was probably not sufficient for drawing a proper statistical conclusion. Therefore, future development of the wave calculation algorithm would be focused on storm situations with larger wave heights that are more important for navigation. Wave field variability parameters could be estimated using wave modeling methods, e.g. SWAN that gives general background for wave parameters, providing specific benefits on coastal sea where morphometry of the coastline is as complicated as the wave field itself.

TABLE 1: DIFFERENCES BETWEEN WAVE HEIGHT PRESSURE BASED REFERENCE MEASUREMENT AND CALCULATED RESULTS

Percentage of calculation results within the maximum difference	**Maximum difference in calculated significant wave height [m]**	
	Range: 0.0 m to 2.0 m (21794 reference points)	***Range: 2.25 m to 5.0 m (401 reference points)***
68.27%	0.29	0.63
90.00%	0.37	0.78
95.00%	0.41	0.86
95.45%	0.41	0.87
99.73%	0.53	1.10

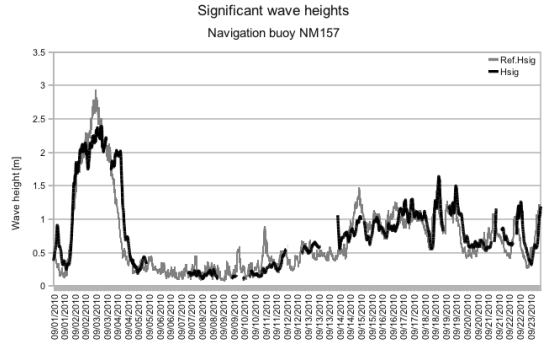

Figure 5: Results of first test period on buoy NM157 (Sept. 2010)

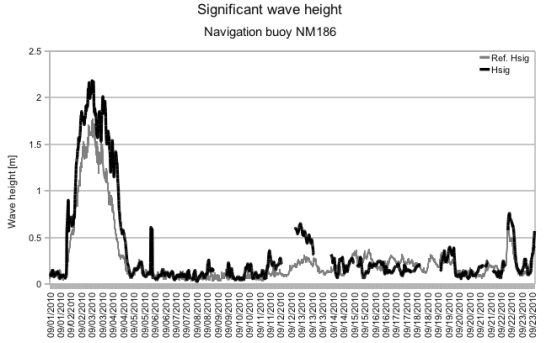

Figure 6: Results of first test period on buoy NM186 (Sept. 2010)

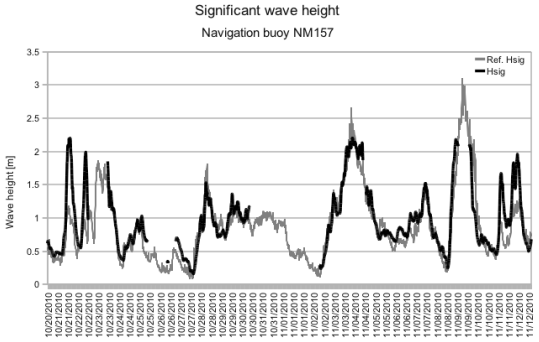

Figure 7: Results of second test period on buoy NM157 (Oct.-Nov 2010)

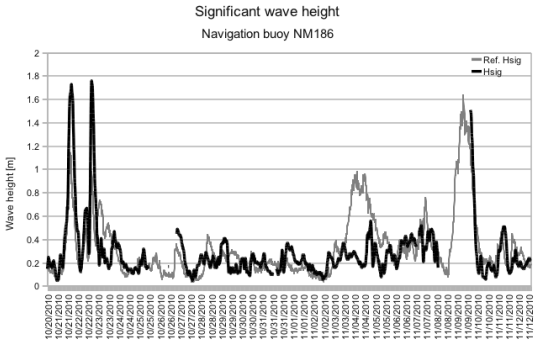

Figure 8: Results of second test period on buoy NM186 (Oct.-Nov. 2010)

Certain errors can be at least partly attributed to the different measurement and reporting intervals and sometimes short data acquisition periods, with both due to the fact that the primary task of a navigational buoy is AtoN signaling. Nevertheless, both errors have almost negligible impact on measured wave heights. Another issue is natural variability on wave field which play role if there is distance between navigational buoy and reference measurement site and always it is. Wave parameters vary both in global scale in the Baltic Sea [4] as well local scale [5]. In both cases seasonal variability has important role, which we took into account planning comparisons in at least two seasons. Main driving force for waves is wind and for future developments we can take into account the fact of anisotropy of wind field over the Baltic Sea [6] (defines the well more probable wind situations, also most extremes, and accordingly the realization of the wave field in certain sea area). Utilizing this fact improves planning of which navigational buoys to use for wave measurements to get better situational awareness of navigation conditions on sea routes.

Once the information from a large number of calibrated wave height sensors is available to the shore side server application that maintains sea state awareness over the whole monitored area, utilization of it for provision of an e-Navigation service to aid the mariner becomes feasible and the system currently in trial state is brought to full operational capability.

V. Conclusions

A method is developed to use navigational buoys equipped with acceleration sensors for estimation of wave parameters. The method is implemented and validated with independent measurements with pressure sensors. Low to medium wave heights are quite well captured by heavy navigation buoys. Two algorithm verification tests were carried out at two different navigational buoy deployment sites, with both tests two weeks long. The differences between the measured and reference wave heights were typically in few tens of centimeters for waves below 2 meters. Problems with higher wave heights are caused both by instrumental reasons, failure of data transmission during stormy weather, and by natural variability in wave field in sea areas with complicated bottom topography what is typically the case nearby the navigational buoys. The main advantage of the developed method is that parameters of the wave field could be measured in situ in open sea conditions –there are very few operational wave data sources in the Baltic Sea and the current development is a step forward. The experiments showed that reference wave height measurements in different forcing conditions are needed for obtaining buoy specific wave height calibration coefficients. Once such calibration effort is done, wave data from extensive sea area is operationally available to support navigation on sea routes.

References

[1] National Data Buoy Center, Nondirectional and Directional Wave Data Analysis Procedures, NDBC Technical Document 96-01, Stennis Space Center [Available on-line at: http://www.ndbc.noaa.gov/wavemeas.pdf]

[2] Alari, Victor; Raudsepp, Urmas. (2010). Depth induced breaking of wind generated surface gravity waves in Estonian coastal waters. Boreal Environment Research, 15, 295 – 300.

[3] C.H. Tsai, M.C. Huang, F.J. Young, Y.C. Lin and H.W. Li. On the recovery of surface wave by pressure transfer function. Ocean Eng., vol. 32, pp. 1247-1259, 2005.

[4] Jönsson, A., Broman, B., Rahm, L. 2002. Variations in the Baltic Sea wave fields. Ocean Engineering, 30, 107-126.

[5] Räämet, A., Soomere, T. 2010. The wave climate and its seasonal variability in the northeastern Baltic Sea. Estonian Journal of Earth Sciences, 59(1), 100 – 113.

[6] Soomere, T. 2003. Anisotropy of wind and wave regimes in the Baltic Proper. J. Sea Res. 49, 305-316.

APPENDIX 4

E. Moorits, G. Jervan, "Profiling in Deeply Embedded Systems", Proceedings of the 13th Biennial Baltic Electronic Conference: 2012 13th Biennial Baltic Electronics Conference (BEC2012), 2012, pp. 127 – 130.

Profiling in Deeply Embedded Systems

E. Moorits[1], G. Jervan[2]

[1]*Department of Navigation Systems, Cybernetica AS, Tallinn, Estonia, E-mail: erkki.moorits@cyber.ee*
[2]*Department of Computer Engineering, Tallinn University of Technology, Tallinn, Estonia, E-mail: gert.jervan@pld.ttu.ee*

ABSTRACT: During the software development stage, every developer observes the program behaviour by using assertions, traces or other debugging methods, but most program bottlenecks and some bugs may surface only during program profiling. Software profiling in desktop systems is a relatively simple task, but unlike in desktop systems, profiling of deeply embedded systems is quite complicated task. In this paper we present a profiling approach for deeply embedded systems which uses GNU toolchain – GCC C compiler for code instrumentation and GProf tool for analysing output data. While we use code instrumentation and transmit profiling data immediately without any buffering, we lose only small amount of program performance.

1 Introduction

Software profiling plays a crucial role in the software development cycle. It can reveal many bottlenecks and in some circumstances even a few bugs in a program. In desktop or server applications profiling is relatively simple task, but in embedded or deeply embedded systems, which may be 8 bit systems with very limited amount of RAM, lack of writeable storage media and sometimes no kernel at all, profiling may be quite a challenging task.

The main driving force for this research was lack of profiling tools, capable to profile deeply embedded systems. Most integrated development environments – (IDE's – e.g. AVR Studio, Code Composer Studio) have simulators but none of them were capable to collect accurately profiling data. Although some CPU-level simulators [1],[2] are capable to collect profiling data, none of those simulators can handle programs which have lot of interaction with the real environment. Therefore, the only feasible way to profile deeply embedded systems is to use compiler assisted profiling methods where profiling is switched on during the program compilation stage and only for interested source files.

In this paper we describe a method for profiling deeply embedded systems. The rest of this paper is organized as follows: Section 2 provides a brief overview of different profiling methods; Section 3 describes the proposed profiling method; Section 4 outlines results of the experiments and performed tests; the concluding remarks are given in Section 5.

2 Profiling Methods

Four different profiling methods exist – manual instrumentation, hardware or kernel assisted methods, automated methods, and simulation based methods. Manual instrumentation is the simplest instrumentation method, which can also give quite good overview about the program behaviour in some sections, but this method is too labour-intensive method, thus it is not usable in our work. Although, one variation of manual instrumentation is used in real time kernels, where instrumentation counters are hard-coded into device drivers, and those counters are accessible through special functions.

CPU-level Simulation

Profiling by using CPU-level simulation is carried out by using special simulator which can record hardware states, events or subroutine calls. Data collected during the simulation can be analyzed typically after simulation [1]-[3], but it should be possible to monitor the program status during simulation. This method is good for analyzing systems which have limited range of input data or very limited interaction with the environment. But it is quite useless in larger systems where it is needed to describe all input states over long period of time. Such system may have so many input states that we can not describe all input combinations and therefore we can not simulate such system. In addition, it is possible, but quite rare, that we need to simulate such situations where some hardware failures or some other hardware related issues, like faulty contacts, fast temperature rise or supply voltage change, etc. might occur. In such special cases we can modify simulator so that it gives output like we want, but this modification is usually too time consuming. In our case we do not expect that we have all input data and want to profile program on the final hardware with all other components in extreme environment conditions, like maximum and minimum operational temperature.

Sampling

Another profiling method is based on periodical program counter sampling [4],[5]. This method is most commonly used in larger systems, like desktop or server systems but sometimes also in embedded systems. Sampling mode profiling has two main advantages – it does not need compiler support and it usually has negligible impact to a program execution. However, this profiling method has

some drawbacks as well. First, as data is collected in regular intervals it may not be so accurate as other profiling methods. Usually this is not a big issue because programs or routines which are monitored several times are statistically correct. Secondly, this method needs typically special kernel driver which is responsible for program counter sampling or kernel which supports program counter sampling for other programs. Therefore in deeply embedded systems it is not possible to use such method because at hardware level this method is too resource consuming – it needs one spare timer and also quite a lot of RAM space. In addition, it may be too complicated to implement because of lack of supporting kernel and/or hardware. Thus, this method is a good choice in larger systems, mostly in Linux based systems, but not for deeply embedded systems.

Instrumentation

Profiling by code instrumentation is the most accurate method and easiest to implement but it has usually quite a big impact to program execution speed – it may cause longer delays in quite unexpected places.

Several different instrumentation-based profiling methods exist: 1) automatic source level instrumentation (e.g. [6]) 2) compiler assisted (e.g. GCC [7]), 3) binary translated, 4) runtime instrumentation (e.g. Pin [8] and Valgrind [9]) and runtime injection (e.g. Paradyn/Dyninst [10]). As deeply embedded systems do not have such computation or hardware resources to modify program code – then we can not use binary translated, runtime instrumentation and runtime injection methods. Therefore, we have only two possible profiling methods – automatic source level instrumentation and compiler assisted. Both methods have eventually the same result, but GCC (GNU Compiler Collection) has a support for compiler assisted instrumentation and we need to rewrite only some small part of GCC which is much easier than to implement full automatic source level instrumentation. Therefore we have chosen compiler assisted instrumentation.

In compiler assisted instrumentation compiler adds profiling function calls to every subroutine call and, depending on the profiling method, to every subroutine return. Profiling functions usually increase call counters by one or in some rare cases can perform some other task, i.e. to send entry or exit event to a capturing host. And collected profiling data is usually stored to "gmon.out" file by *mcleanup* function as the program exits. *Mcleanup* function also disables all further profiling and adds file headers to the output file. While this method is relatively easy to implement, it has highest impact from all profiling methods to program execution speed and also consumes quite a lot of processor and memory resources. Therefore, we can not assume that the processor has the same resources when profiling is enabled – we lose some performance during profiling as we get back some profiling information. As long as the impact of profiling data collection to the system functionality is acceptable, we can use it in our development.

For profiling deeply embedded systems we can transmit all subroutine entry and exit calls immediately to the capturing PC instead of storing profiling data into a RAM. This kind of modification of a typical profiling method makes it possible to profile deeply embedded systems without a simulator. But it may not work in systems which have very critical timing sequences or no suitable communication interface.

3 Implementation

The basic principle of our profiling method is to send all instrumentation data out to a capturing system, which have significant amount of memory and processing power i.e. to a desktop PC. This capturing system collects and analyzes the data. For profiling in deeply embedded software we need to generate instrumented firmware, which also have profiling data transmission functions, load this new firmware into targeted microcontroller and start programs which are responsible of collecting profiling data in capturing side. After specified time or functions entry/exit calls, capturing program translates collected information to *gmon* statistics format which can be later analyzed with GProf.

In our current work we have tested this method with a GCC C compiler, linker from GNU binutils package and AVR microcontrollers, ATtiny2313, ATmega64 and ATmega1280 [11]. All targeted microcontrollers have 16-bit program counters, but the developed method is suitable for an arbitrary microcontroller regardless of the program counter width.

Compiling, Linking and Instrumentation

In order to produce instrumented firmware we can use two different methods to add instrumentation code into the final program.

The first method is based on compiler *mcount* function (*_mcount* or *__mcount*, depending on the OS and compiler) which can be switched on during the compilation phase with the *-pg* command line option. On most architectures GCC have working instrumentation functions, but for deeply embedded systems (e.g. 8 bit AVR family microcontrollers) GCC usually does not have working instrumentation functions. The main reason for absence of the working instrumentation functions is the lack of writable storage media on microcontroller. To add profiling capability to GCC we need to modify GCC in some extent. Most important is to add the right references to *mcount* function – a function which is responsible for capturing profiling data from the microcontroller side, and we also need to provide our *mcount* function. While calling any function the processor stores call site information to a stack, thus we can read this call site information directly from the stack and transfer it to a capturing PC. In order to place instrumentation function (*mcount* function) calls before the real function is called,

i.e. calls are placed before function prologue, we need to define PROFILE_BEFORE_PROLOGUE macro in GCC source (in *config/avr/avr.h*). This defined macro allows us to read both call site addresses from stack – function which calls the *mcount* function and the function before that. Also, we have modified FUNCTION_PROFILER macro in GCC source (in *config/avr/avr.h*) to add *call* or *rcall* (relative call to subroutine) instruction for calling *mcount* function. In addition, AVR GCC is shipped with such *mcount* function that has empty body and returns immediately after its calling, so we need to add our *mcount* function which transfers data out to a capturing host. To add our own *mcount* function we have to change the called *mcount* function reference to a weak reference – if we do not provide any *mcount* function then linker adds automatically an empty *mcount* which is shipped with GCC.

Second profiling method is to use `__cyg_profile_func_enter` and `__cyg_profile_func_exit` functions. Both functions execute pre-defined routines when function is called or when it returns. But `__cyg_profile_func_enter` and `__cyg_profile_func_exit` functions behave a little bit differently from *mcount* function. When *mcount* functions are added after optimization then `__cyg_profile_func_enter` and `__cyg_profile_func_exit` are added before optimization. Therefore, both *cyg_profile* functions are added to all static inline functions. Usually many libraries have defined many static inline functions, therefore it is quite difficult to use above two functions as compiler have inserted those into the final code, and the profiling functions take too many resources. To overcome this it is possible to redefine function headers by adding an attribute which do not allow compiler to add *cyg_profile* function to inline function. Unfortunately, this approach is quite error borne and also may not work in all compilers in the same way. In addition to the above, when program calls `__cyg_profile_func_enter` function then AVR GCC usually does not return correct call site address. Instead it will return some faulty data from the stack, and therefore it is possible to use the function's own address, but not the call site data.

Currently we have implemented in assembly-language our own *mcount* function which is 84 instructions long and does not use any additional memory to save profiling data. Therefore, no additional RAM is taken by this method, but it needs some additional program memory (S_{total}):

$$S_{total} = S_{call} n_{functions} + S_{profiling\,function} \quad (1)$$

where S_{call} is the size of a *call* instruction in bytes, (for AVR microcontroller 2 bytes), $n_{functions}$ is the number of called functions and $S_{profiling\ function}$ is the size of the profiling function, in current system it is 168 bytes (84 instructions). For example, systems where ten functions are instrumented it is needed only 188 bytes more program memory storage.

Data Transmission

The easiest and the most cost effective method to transfer profiling data to a capturing PC is to use serial interface. In our system where microcontroller with a 16-bit program counter is used, we need to transfer in every function call at least four bytes of data – two bytes for call site address and two bytes for called function address. The same also applies for cases when returned from calling function. While synchronizing the target and host systems we need to add some negotiation packets or to use special input data format. Negotiation packets are suitable in such cases where systems do not lose synchronization during data transmission. In our case we expect that the host PC and microcontroller stays in synchronization for all the time and therefore we have included one byte constant synchronization header to every profiling packet. Therefore, for every function call we need to transfer five bytes of data – one byte for header, two bytes for call site and two bytes for caller – and the number of function entry and exit calls per one second (N_{fn}) would be:

$$N_{fn} = \frac{BR}{2 N_b} \quad (2)$$

where N_b is the number of bits for one function exit or entry call and BR is the data link baud rate. For example, when we use standard serial interface with speed 115200 b/s, 5 bytes of data, and 2 stop bits then we can have 1152 function calls per second. Also, with special hardware it is possible to use 9 bit (standard PC does not support 9 bit serial data) transmission which eliminates the need for the packet header – we can use ninth bit for a header. In such case, with one stop bit, we can have 1440 function calls per second. Calling *mcount* function gives also a small overhead – for 84 cycle long function with call, return and with five internal branches it takes at least 88 CPU cycles. Compared with serial throughput it does not add any significant impact to the overall speed. To increase data throughput it is also possible to use JTAG, but in our current work we have not considered this because on current architecture passing data through JTAG is technically quite complicated and all targeted microcontrollers does not support JTAG.

Data collection and Output

At the capturing PC side all function calls are counted and saved to "gmon.out" statistics file. In our case, where controller has 16-bit program counter, we can store all function call counts into big array which is saved to *gmon* file after a certain number of collected entry/exit calls or after certain amount of time. Capturing program also examines the call address of the called subroutine, the return address to the calling subroutine and compares them with addresses which are decoded from the ELF (Executable and Linkable Format) file. If capturing program finds any discrepancy between calling subroutine or called subroutine with decoded program then all captured data are saved and capturing program exits. In

our current work our capturing program counts only function calls and does not hold function call graph. For statistics output we used BSD profiling file format, which is easiest to implement but does hold only very basic profiling data. Generated statistics file, which holds profiling information, can be analyzed with most profiling tools that can read BSD profiling format. In our case we used GProf.

4 Test Results

We have carried out three different profiling tests. In the first test we used ATtiny2313 microcontroller with 4 MHz CPU clock. The test program was simple super-loop program which toggles microcontroller output pin in one second period. We also did not use any interrupts for precise timing. In this test, profiling code increased program by 184 bytes, which is nearly 9% from all program memory. This test was carried out with two different baud rates – 38400 b/s and 115200 b/s. To compare with non instrumented code all cycles were delayed by 8 ms at the first baud rate, but at the second baud rate we did not detect any significant delay in program execution. This test shows quite well the limitations of this method – it is quite difficult to instrument simple super-loop real-time programs without introducing extra delays. Therefore, the current method quite likely violates real-time constraints in real-time super-loop programs.

Second test was simple super-loop program which writes its current up-time to one serial port. In this test we used ATmega64 microcontroller with 8 MHz CPU clock. After instrumenting the program, its size was increased by 222 bytes, which is less than 1% of total program memory. Compared with non instrumented program, all uptime writing to serial port were delayed only by 40 ms with baud rate of 38400 b/s. This delay corresponds to 14 calls of *mcount* function.

The third test was carried out in with ATmega1280 microcontroller which, was clocked at 7.3728 MHz, instrumentation interface baud rate was 115200 b/s and a RTOS was used. In this test we instrumented some of the test programs and RTOS functions, leaving out time-critical functions and interrupt service routines (ISR's). After instrumentation the whole program size increased by 2 kB which is less than 2% of the total program memory. Instrumented programs wrote its up-time to a serial port and answered queries from RS-485 line. In comparison with non instrumented program we did not detect any significant delay during program execution. The main reason why we did not see any significant delay or other effects from the instrumentation is that we instrumented only those parts of the program which worked only on user request or were not time critical. With interrupt driven programs we may expect to have the same behavior as in tests with RTOS.

To summarize the performed tests – in most tests profiling slowed the program execution only a little but with lower transmission baudrate it may slow down the whole program execution speed significantly. With RTOS it is quite easy to separate the time critical and ISR subroutines from the user programs, which makes the whole profiling much easier than it is in super-loop programs.

5 Conclusions

The objective of this work was to develop a profiling method which is suitable for deeply embedded systems. The developed method is capable of profiling programs in systems which have as low as two bytes of free RAM and at least 170 bytes of free program memory and one free serial interface. Developed method uses compiler assisted instrumentation and stores all collected data to a host computer. The host computer analyses the collected data and writes results to a file in BSD profiling file format. The method was successfully tested with three different AVR microcontrollers, where one test was with RTOS and two tests with simple super-loop programs. Two tests showed that this method has minor problems with super-loop real-time programs, but with RTOS or interrupt driven programs this method has negligible effect on program execution.

References

[1] Avrora; http://compilers.cs.ucla.edu/avrora; 2012

[2] MSPSim; http://www.sics.se/project/mspsim; 2012

[3] Ben L. Titzer, Jens Palsberg; "Nonintrusive precision instrumentation of microcontroller software", LCTES 2005; pp. 59-68; June 2005

[4] Liu Fagui, Li Shengwen, Xie Ran, Luo Chunwei; "A low-overhead method of embedded software profiling", Computing, Communication, Control, and Management, 2009. CCCM 2009. ISECS International Colloquium on; vol. 4, pp. 436-439; August 2009

[5] Oprofile; http://oprofile.sourceforge.net; 2012

[6] Quan Sun, Hui Tian; "A flexible automatic source-level instrumentation framework for dynamic program analysis"; Software Engineering and Service Science (ICSESS), 2011 IEEE 2nd International Conference; pp. 401-404; July 2011

[7] GCC, the GNU Compiler Collection; http://gcc.gnu.org; 2012

[8] Pin; http://www.pintool.org; 2012

[9] Valgrind; http://valgrind.org; 2012

[10] Paradyn/Dyninst; http://www.dyninst.org; 2012

[11] Atmel AVR 8- and 32-bit Microcontrollers; http://www.atmel.com/products/microcontrollers/avr/default.aspx; 2012

APPENDIX 5

E. Moorits, A. Usk, "Buoy Collision Detection", Proceedings of the 54th International Symposium Electronics in Marine ELMAR-2012, 2012, pp. 109 – 112.

Buoy Collision Detection

Erkki Moorits, Aivar Usk
Cybernetica AS, Akadeemia Tee 21, Tallinn, Estonia
erkki.moorits@cyber.ee

Abstract - This paper presents a method developed for collision detection on navigational buoys by the use of onboard low power embedded controllers equipped with solid state acceleration sensors, focusing on the signal processing principles employed. Detection of a collision of a vessel with a buoy is based on continuous monitoring of the acceleration profile of the buoy in all three planes of movement, accomplished using a 3-axial solid state accelerometer (g-sensor) with the maximum range of ±3 g. The sensor is integrated with a marine Aid to Navigation (AtoN) telematics module that is subject to low power consumption requirements and size restrictions resulting in limited computational capability. Initial operational testing was performed on navigational buoys in actual marine environment, including sea ice conditions. The results have validated the usability of the method, although no ship-to-buoy collisions have been encountered.

Keywords – real-time collision monitoring; ship-to-buoy collision; navigational buoy

I. INTRODUCTION

Despite the widespread use of mature electronic technologies for marine navigation, visual light navigation stations remain an indispensable part of marine navigation safety infrastructure for foreseeable future. Availability of the light signalling service provided by the fleet of floating aids of any responsible maritime authority depends in part on awareness of collision events that may breach the integrity of the floating platform and result in failure of the light signal.

Several crash detection methods have been developed during last decades for control of automotive airbag deployment; all of these utilize acceleration sensors and use very fast and robust filtering methods on high acceleration peaks. In case of ship-to-buoy collisions, absolute acceleration levels are expected to remain rather low, therefore existing methods that are developed for ground transport systems are not suitable. We have not found any information on previous published works in this particular field which may be due to the facts that establishing a collision reporting system may require a long time to earn the investment to the end user, and most buoys are not equipped with sensors that are capable of detecting the collision events. However, buoy collision events are not encountered very often, mostly because of rather low collision probability [1]. According to ship/platform collision incident database [2], only a few accidents relate to buoys, but most likely many collision events of smaller magnitude are either not reported to relevant authorities, or remain completely unnoticed while possibly causing latent failure of the platform or equipment. Receiving an immediate notification about collision events exceeding a pre-set criticality threshold is necessary not only for timely re-establishment of the AtoN signal when needed, but also for reduction of pollution risks and identifying the particular vessel responsible for the damages by correlating the collision event time stamp with external vessel movement information sources (AIS or VTS databases).

The most cost effective of contemporary methods for detection of collision of a buoy with other floating objects is achieved by application of an embedded microcontroller equipped with a three-axial micromechanical accelerometer sensor (a solid state g-sensor – a Micro-Electromechanical System (MEMS)) that can measure acceleration levels on all three axes simultaneously. While collision detection methods based on acceleration measurement have been developed for car crash detection systems, mainly for activating airbag inflation, such methods are based on detection of high acceleration levels occurring in a very short timeframe. In case of floating AtoN collisions, the event profile is rather different, displaying typical acceleration levels even below 5g with the duration of up to several seconds.

This paper presents a collision detection method which is suitable for implementation on navigational buoys. Previously we have implemented a method for in-situ determination of heel (inclination) angle of navigational buoys [3] based on same embedded telematics hardware; the subject collision detection method is a second component of the floating platform status monitoring subsystem.

II. CURRENT SYSTEM OVERVIEW

While AtoN remote control and monitoring systems (RCMS) of varying degrees of sophistication have been around for decades, collision detection has not been widely used due to complexity and cost - in an autonomous system that needs to provide reliable operation from primary batteries for years, spending of every mA of current must be well substantiated. Our concept foresaw integration of a single new hardware component (3-axial g-sensor) with the existing telematics module (TM) used for remote monitoring of navigational buoys, and accomplishment of collision detection as well as heel angle calculation [3] and monitoring tasks using the available ADC ports and spare computational capacity of the existing embedded microcontroller. In a typical application, the TM is installed inside a protective enclosure together with a flasher module and an LED array, and mounted on a buoy superstructure, typically 2 to 4.5 meters above the sea level. Communication protocols of the TM that serves primarily as a communications gateway between the remote site equipment and the RCMS centre server were updated to accommodate support for collision detection alarms.

A TM is performing continuous acceleration data acquisition of three 10-bit acceleration measurement values representing acceleration levels sampled from three axes of the g-sensor. All samples are acquired with a 20 ms interval, with

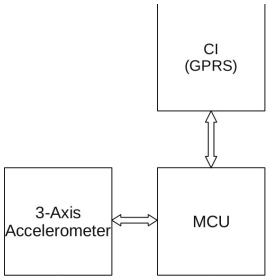

Figure 1. TM Block Diagram

sequential delays of 0.2 ms between readings acquired from x-, y-, and z-axes that in practice can be considered simultaneous sampling due to the slow progress of acceleration events. Since acceleration data acquisition is not the primary task for the TM, under certain circumstances the acceleration data may be sampled at slightly uneven intervals due to coinciding higher priority tasks of the processor. Due to the considerably long typical acceleration signal duration of 5 to 20 seconds, such occasional uneven sampling does not have any significant detrimental impact on collision detection.

Fig. 1 shows relevant subsystems of a TM utilized for acceleration measurement, collision detection and heel angle calculation and status/alarm communication tasks, leaving out all parts which are not involved in the process. The MCU used is an 8-bit AVR microcontroller, performing analog-to-digital conversion of g-sensor output data, collision detection, heel angle calculations and initiating communications with the monitoring centre when necessary. When a collision event of significant magnitude is detected, the MCU initiates a communications session with the RCMS centre server using the communications interface (CI) to report a collision event. In the current system implementation, the CI is a GSM/GPRS modem. Collision event reporting, which includes collision event detection, time-stamping and connection to the RCMS typically takes 5 seconds, which is quite acceptable in most cases. The acceleration sensor utilized is an ADXL330 by Analog Devices [4] which is connected directly to the analog input channels of the MCU.

III. COLLISION DETECTION

To detect collisions with a navigational buoy, we must continuously monitor acceleration signals from all three axes. Assuming that we use this method only with navigational buoys, the sampling period can be set quite low but not below 20 ms. In a typical buoy installation we may assume that a collision may appear from any direction and therefore we must take into account signals from all three acceleration axes. To detect a collision event in case of unlimited computational resources available, one would calculate the acceleration vector length, taking into account all acceleration values, and base the decision on that vector length. In our case, the system has rather limited amount of memory and computational capability, therefore it is not practical to calculate the vector length; instead, we can achieve almost same results by adding up the acceleration values. Using only such summation, we must take into account the fact that during collisions we get much higher resulting acceleration than in case of using acceleration vectors; this is usually the case when an impact comes in between two or three axes. Due to the specifics of our application, we can tolerate errors which are introduced by higher acceleration values since we do not need very exact values, but we need to know when maximum acceleration value exceeds certain threshold level.

Therefore, we can sum up the axial components for total acceleration:

$$A = A_x + A_y + A_z \tag{1}$$

where A_x, A_y and A_z are acceleration measurement values read directly from the g-sensor.

In order to detect collision from total acceleration A, we should filter out the static (DC) component from obtained signal. To filter out the DC component, we can use the following IIR filter:

$$y1_n = a1_0 \cdot x1_n + a1_1 \cdot x1_{n-1} + b1_1 \cdot y1_{n-1} \tag{2}$$

where $a1_0$ is IIR filter polynomial multiplier value 0.4844, $a1_1$ is multiplier value -0.4844 ($a1_0 = -a1_1$), $b1_1$ is feedback polynomial multiplier with value 0.9375, $x1_n$ is the last input value, $x1_{n-1}$ is the previous input value, $y1_n$ is the current output value and $y1_{n-1}$ the last output value. Multipliers $a1_0$, $a1_1$ and $b1_1$ are found as follows:

$$a1_0 = (1 + xt)/2 \tag{3}$$

$$a1_1 = -(1 + xt)/2 = -a1_0 \tag{4}$$

$$b1_1 = xt \tag{5}$$

where xt is a frequency-dependent constant:

$$xt = e^{-2\pi f_c} \tag{6}$$

where f_c is the normalized filter cut-off frequency.

In case of the particular buoy platforms used in operational testing of this method, we have chosen 0.5135 Hz for the DC cut-off frequency; this allows to filter out most of the acceleration related to wave action. Other buoy types may require a different DC cut-off frequency. With this cut-off frequency, the normalized cut-off frequency for the 20 ms sampling period is 0.01027 f_{sampl}

Values $a1_0$ and $a1_1$ that are found using formula 3 and 4 both have two times higher values than those values which are inserted into formula 2. Mostly this dividing by two is needed for preventing overflow of the next filter stage, while smaller values are also easier to process in 8 bit microcontrollers than original values.

A filter described by formula 2 with values $a1_0$ and $a1_1$ forms a differential stage that may provide a negative output signal $y1_0$, but negative accelerations do not have any meaning in this collision detection system. Therefore, we can square the $y1_0$:

$$x2_n = y1_n^2 \tag{7}$$

where $x2_n$ is an input value to the next stage. To eliminate negative values from $y1_n$, we can also use absolute value, but taking a square also reduces smaller values which are mostly noise or signal changes which do not carry any significant information for collision detection.

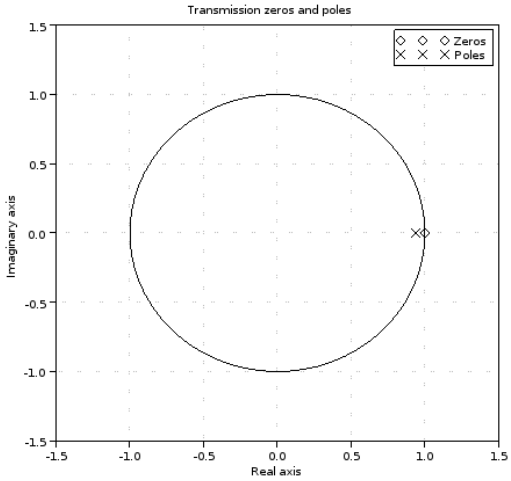

Figure 2: Poles and zeros of the first stage of crash detection filter

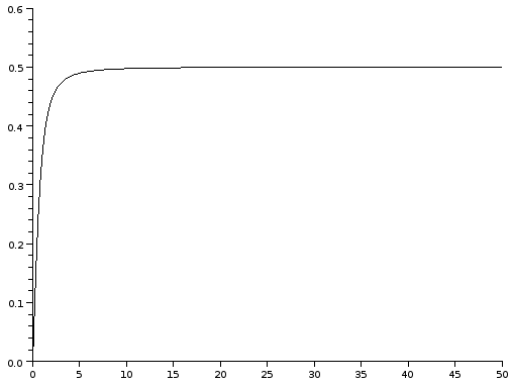

Figure 3: Transfer function of the first filter

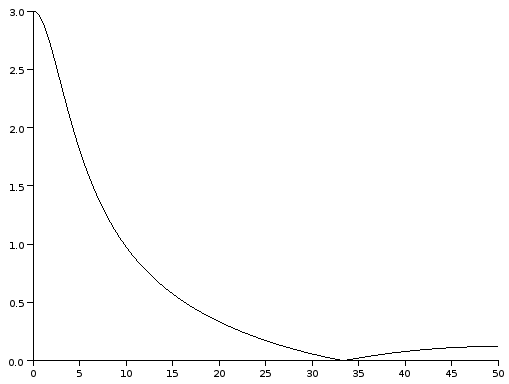

Figure 4: Transfer function of the second filter

After DC level removal, we apply the second stage of filtering which plays a major role in our collision detection system. A collision detection filter should be rather fast, with acceptable filter delay in our case of less than 1 second; this filter must be quite robust as well to avoid false collision reports. Therefore, we need to make this filter partly a pure averaging filter and also a low-pass filter (LP filter). Such a filter can be described using the following polynomial:

$$y2_n = a2_0(x2_n + x2_{n-1} + x2_{n-2}) + b2_1 \cdot y2_{n-1} \quad (8)$$

where $a2_0$ is a polynomial multiplier value 0.2188, $b2_1$ is a feedback value 0.7812, $x2_n$ is an input value, $x2_{n-1}$ and $x2_{n-2}$ previous input values, $y2_n$ an output value and $y2_{n-2}$ the last output value. $a2_0$ is also an input signal multiplier value. Both constants $a2_0$ and $b2_1$ are found as follows (where g is the input signal gain):

$$a2_0 = (1 - xt) \cdot g \quad (9)$$

$$b2_1 = xt \quad (10)$$

The value $y2_n$ holds enough noise free averaged acceleration information to detect collision with an object. In the last step we only compare series of $y2_n$ values to the pre-configured collision threshold value; when $y2_n$ are successively higher than the collision threshold value during a predefined timeframe, we register a collision event.

A. *Algorithm stability and transfer functions*

The filters described above (formulae 2 and 8) are stable when $|b1_1| < 1$ and $|b2_1| < 1$. When using formula 6 to calculate the constants $b1_1$ and $b2_1$, the condition $f_c > 0$ should be true. To ensure that above mentioned condition is maintained, all poles must be inside a unit circle (Fig. 2).

Note that the signal in Fig. 4 is three times higher than it should be; this is caused by adding up the acceleration values from all three axes. In our application where the second filter is an integrator, it is not necessary for this filter to be very precise.

B. *Crash detection algorithm adaptation to a 8 bit MCU*

To use the above described methods on an 8-bit embedded microcontroller, we need to convert all constants to fixed point or integer values. This introduces one additional division operation in every filter. After introducing additional division operation into formula 2 we arrive at the following equation:

$$y1_n = \frac{1}{b1_0}(a1_0 \cdot x1_n + a1_1 \cdot x1_{n-1} + b1_1 \cdot y1_{n-1}) \quad (11)$$

where $a1_0$ is an IIR filter polynomial multiplier with value 62, $a1_1$ is a multiplier with value -62 ($a1_0 = -a1_1$), $b1_0$ is a feedback polynomial multiplier with value 128, and $b1_1$ is a feedback polynomial multiplier with value 120.

To find value $x2_n$, we need to square $y1_n$ and scale it down:

$$x2_n = \frac{y1_n^2}{256} \quad (12)$$

After changing the second filter (formula 8) to fixed point values we get following formula:

$$y2_n = \frac{1}{b2_0}(a2_0(x2_n + x2_{n-1} + x2_{n-2}) + b2_1 \cdot y2_{n-1}) \quad (13)$$

where $a2_0$ is a polynomial multiplier value 56, $b2_0$ is the feedback value 256 and $b2_1$ is the feedback value 200. Since all results are stored in 8-bit variables, we need to check before

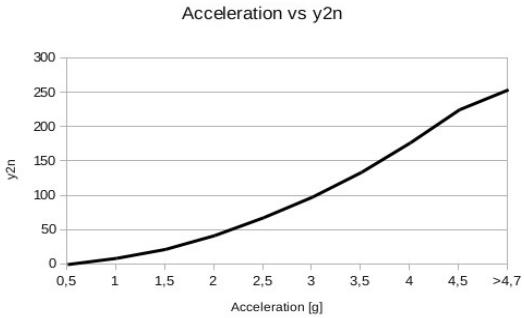

Figure 5: Output value vs acceleration

storing the result that the computations ended with a value with a maximum of 255; higher values have to be coerced to 255.

C. *Detected crash values*

Since we use IIR filters, we cannot directly link certain input values to output values. Table 1 and Fig. 5 presents results of testing of our filter with one typical crash signal:

TABLE I. Y2N VALUE VS TOTAL ACCELERATION

Total acceleration [g]	y2n
0.5	1
1.0	10
1.5	23
2.0	43
2.5	69
3.0	99
3.5	135
4.0	178
4.5	226
>4.7	255

IV. SIMULATIONS AND TESTS

We have carried out several simulations and tests to verify our method, but (fortunately) none of over 100 buoys fitted with this technology have registered any real collisions with a vessel yet.

Fig. 6 presents a typical simulation. For simplification, all three acceleration vectors are summed up, and the Earth's gravitational acceleration is subtracted. Resulting acceleration signal is the signal that we expect to get when a collision of the buoy with a ship is encountered. As is shown in Fig. 6, with sufficient acceleration we can get our collision events quite fast. In that simulation, first event may be recorded 50 ms after the first acceleration peak and the second one 400 ms after the first acceleration peak. We have also carried out several tests where we mounted the TM to a heavy object and tried to hit this object with another object, i.e. make an artificial collision, and the results were comparable to the simulation in Fig. 6.

In last two years we have tested this method on actual navigational buoys but have not registered any ship-to-buoy collision events. Nevertheless, we have registered two interesting events – first, when the ice moved over the buoy

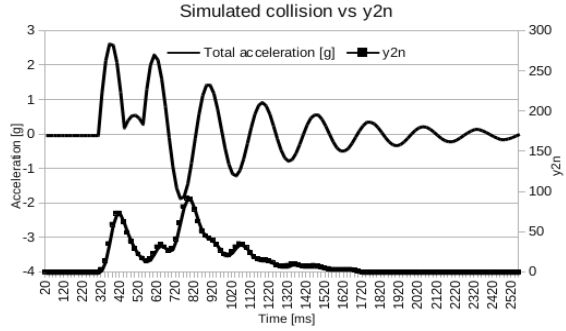

Figure 6: Simulated collision

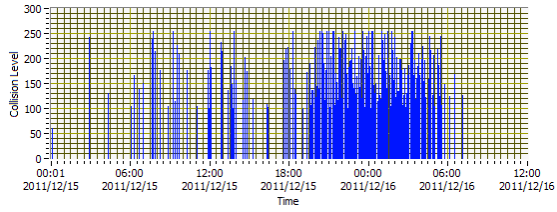

Figure 7: Collision reports from Estonian AtoN No. 861 due to short mooring pull during a buildup of north winds that started approximately at noon of 2011-12-15 and sustained speeds in gusts in excess of 12 m/s to 19 m/s. Wind speed dropped below 8 m/s and started to change the direction at 06:00 (UTC) the next day.

and second, when the mooring was too short for the current buoy's deployment site, causing significant deceleration pulls in wind gusts and wave action (Fig. 7).

V. CONCLUSIONS

The objective of the current work was to develop a low resource demanding collision detection method that is feasible for application on navigational buoys. The developed method is suitable for low power marine AtoN based embedded systems with an integrated 3-axis MEMS acceleration sensor. This method is capable of collision detection in real time even with rather low acceleration signals. The method was successfully simulated and tested in our laboratory as well as in expected operational environment; all simulation results were in line with our expectations, although no real ship-to-buoy collision events have been detected at seas by this time.

REFERENCES

[1] Margaret Loudon Flohberger, "Suggested Improvements For Ship-Installation Collision Risk Models To Reflect Current Collision Avoidance Systems", University of Stavanger, Faculty of Science and Technology, 2010, pp. 3-5

[2] Serco Assurance, "Ship/platform collision incident database (2001)", Research report 053, HSE Books, 2003, pp. 70-96

[3] E. Moorits, A. Usk, "A Numerically Efficient Method for Calculation of the Angle of Heel of a Navigational Buoy", 2010 Biennial Baltic Electronics Conference (BEC2010), October 2010, pp. 357-360

[4] Analog Devices, Inc. "ADXL330: Small, Low Power, 3-Axis ±3g iMEMS® Accelerometer", 2007

CURRICULUM VITAE

Personal data

Name:	Erkki Moorits
Date and place of birth:	01 October 1981
Nationality:	Estonian

Contact information

Address:	Tildri 17-17, Tallinn, Estonia
Telephone:	+3725215577
E-mail address:	erkki.moorits@mail.ee, erkki.moorits@cyber.ee

Education

2006 – 2008	M.Sc. in Electronics engineering, TUT
2001 – 2005	B.Sc. in Electronics and biomedical engineering, TUT
1997 – 2001	Telecommunication, Tallinna Polütehnikum

Career

2006 – present	Cybernetica AS , Programmer
2001 – 2005	Artvali OÜ, Security Equipment Specialist
2000	Eesti Telefon (Connecto), Telecom Specialist

Defended theses

2008	Hardware Means for Provision of High Availability Operation of a Server Component in a Mission Critical Remote Monitoring System

2005	Remote Vibration Sensor With Integrated Programmable LED Matrix Display

Main areas of scientific work/current research topics

Deeply embedded systems for AtoN devices

ELULOOKIRJELDUS

Isikuandmed

Ees- ja perekonnanimi:	Erkki Moorits
Sünniaeg ja -koht:	01 oktoober 1981
Kodakondsus:	Eesti

Kontaktandmed

Aadress:	Tildri 17-17, Tallinn, Eesti
Telefon:	+3725215577
E-posti aadress:	erkki.moorits@mail.ee, erkki.moorits@cyber.ee

Hariduskäik

2006 – 2008	tehnikateaduste magister (elektroonika), Tallinna Tehnikaülikool
2001 – 2005	tehnikateaduse bakalaureus (elektroonika ja biomeditsiinitehnika), Tallinna Tehnikaülikool
1997 – 2001	telekommunikatsioon, Tallinna Polütehnikum

Teenistuskäik

2006 – veel töötan	Cybernetica AS, Programmeerija
2001 – 2005	Artvali OÜ, Elektroonik/Turvaseadmete spetsialist
2000	Eesti Telefon (Connecto), Telekommunikatsiooniseadmete spetsialist

Kaitstud lõputööd

2008	Missioonikriitilise kaugseiresüsteemi

	serverkomponendi kõrgkäideldavuse tagamine riistvaraliste vahenditega
2005	Integreeritud programmeeritava valgusdioodmaatriksnäidikuga vibratsiooni kaugandur

Teadustöö põhisuunad

Valgusnavigatsioonis kasutatavad sardsüsteemid

DISSERTATIONS DEFENDED AT TALLINN UNIVERSITY OF TECHNOLOGY ON *INFORMATICS AND SYSTEM ENGINEERING*

1. **Lea Elmik**. Informational Modelling of a Communication Office. 1992.

2. **Kalle Tammemäe**. Control Intensive Digital System Synthesis. 1997.

3. **Eerik Lossmann**. Complex Signal Classification Algorithms, Based on the Third-Order Statistical Models. 1999.

4. **Kaido Kikkas**. Using the Internet in Rehabilitation of People with Mobility Impairments – Case Studies and Views from Estonia. 1999.

5. **Nazmun Nahar**. Global Electronic Commerce Process: Business-to-Business. 1999.

6. **Jevgeni Riipulk**. Microwave Radiometry for Medical Applications. 2000.

7. **Alar Kuusik**. Compact Smart Home Systems: Design and Verification of Cost Effective Hardware Solutions. 2001.

8. **Jaan Raik**. Hierarchical Test Generation for Digital Circuits Represented by Decision Diagrams. 2001.

9. **Andri Riid**. Transparent Fuzzy Systems: Model and Control. 2002.

10. **Marina Brik**. Investigation and Development of Test Generation Methods for Control Part of Digital Systems. 2002.

11. **Raul Land**. Synchronous Approximation and Processing of Sampled Data Signals. 2002.

12. **Ants Ronk**. An Extended Block-Adaptive Fourier Analyser for Analysis and Reproduction of Periodic Components of Band-Limited Discrete-Time Signals. 2002.

13. **Toivo Paavle**. System Level Modeling of the Phase Locked Loops: Behavioral Analysis and Parameterization. 2003.

14. **Irina Astrova**. On Integration of Object-Oriented Applications with Relational Databases. 2003.

15. **Kuldar Taveter**. A Multi-Perspective Methodology for Agent-Oriented Business Modelling and Simulation. 2004.

16. **Taivo Kangilaski**. Eesti Energia käiduhaldussüsteem. 2004.

17. **Artur Jutman**. Selected Issues of Modeling, Verification and Testing of Digital Systems. 2004.

18. **Ander Tenno**. Simulation and Estimation of Electro-Chemical Processes in Maintenance-Free Batteries with Fixed Electrolyte. 2004.

19. **Oleg Korolkov**. Formation of Diffusion Welded Al Contacts to Semiconductor Silicon. 2004.

20. **Risto Vaarandi**. Tools and Techniques for Event Log Analysis. 2005.

21. **Marko Koort**. Transmitter Power Control in Wireless Communication Systems. 2005.

22. **Raul Savimaa**. Modelling Emergent Behaviour of Organizations. Time-Aware, UML and Agent Based Approach. 2005.

23. **Raido Kurel**. Investigation of Electrical Characteristics of SiC Based Complementary JBS Structures. 2005.

24. **Rainer Taniloo**. Ökonoomsete negatiivse diferentsiaaltakistusega astmete ja elementide disainimine ja optimeerimine. 2005.

25. **Pauli Lallo**. Adaptive Secure Data Transmission Method for OSI Level I. 2005.

26. **Deniss Kumlander**. Some Practical Algorithms to Solve the Maximum Clique Problem. 2005.

27. **Tarmo Veskioja**. Stable Marriage Problem and College Admission. 2005.

28. **Elena Fomina**. Low Power Finite State Machine Synthesis. 2005.

29. **Eero Ivask**. Digital Test in WEB-Based Environment 2006.

30. **Виктор Войтович**. Разработка технологий выращивания из жидкой фазы эпитаксиальных структур арсенида галлия с высоковольтным p-n переходом и изготовления диодов на их основе. 2006.

31. **Tanel Alumäe**. Methods for Estonian Large Vocabulary Speech Recognition. 2006.

32. **Erki Eessaar**. Relational and Object-Relational Database Management Systems as Platforms for Managing Softwareengineering Artefacts. 2006.

33. **Rauno Gordon**. Modelling of Cardiac Dynamics and Intracardiac Bio-impedance. 2007.

34. **Madis Listak**. A Task-Oriented Design of a Biologically Inspired Underwater Robot. 2007.

35. **Elmet Orasson**. Hybrid Built-in Self-Test. Methods and Tools for Analysis and Optimization of BIST. 2007.

36. **Eduard Petlenkov**. Neural Networks Based Identification and Control of Nonlinear Systems: ANARX Model Based Approach. 2007.

37. **Toomas Kirt**. Concept Formation in Exploratory Data Analysis: Case Studies of Linguistic and Banking Data. 2007.

38. **Juhan-Peep Ernits**. Two State Space Reduction Techniques for Explicit State Model Checking. 2007.

39. **Innar Liiv**. Pattern Discovery Using Seriation and Matrix Reordering: A Unified View, Extensions and an Application to Inventory Management. 2008.

40. **Andrei Pokatilov**. Development of National Standard for Voltage Unit Based on Solid-State References. 2008.

41. **Karin Lindroos**. Mapping Social Structures by Formal Non-Linear Information Processing Methods: Case Studies of Estonian Islands Environments. 2008.

42. **Maksim Jenihhin**. Simulation-Based Hardware Verification with High-Level Decision Diagrams. 2008.

43. **Ando Saabas**. Logics for Low-Level Code and Proof-Preserving Program Transformations. 2008.

44. **Ilja Tšahhirov**. Security Protocols Analysis in the Computational Model – Dependency Flow Graphs-Based Approach. 2008.

45. **Toomas Ruuben**. Wideband Digital Beamforming in Sonar Systems. 2009.

46. **Sergei Devadze**. Fault Simulation of Digital Systems. 2009.

47. **Andrei Krivošei**. Model Based Method for Adaptive Decomposition of the Thoracic Bio-Impedance Variations into Cardiac and Respiratory Components. 2009.

48. **Vineeth Govind**. DfT-Based External Test and Diagnosis of Mesh-like Networks on Chips. 2009.

49. **Andres Kull**. Model-Based Testing of Reactive Systems. 2009.

50. **Ants Torim**. Formal Concepts in the Theory of Monotone Systems. 2009.

51. **Erika Matsak**. Discovering Logical Constructs from Estonian Children Language. 2009.

52. **Paul Annus**. Multichannel Bioimpedance Spectroscopy: Instrumentation Methods and Design Principles. 2009.

53. **Maris Tõnso**. Computer Algebra Tools for Modelling, Analysis and Synthesis for Nonlinear Control Systems. 2010.

54. **Aivo Jürgenson**. Efficient Semantics of Parallel and Serial Models of Attack Trees. 2010.

55. **Erkki Joasoon**. The Tactile Feedback Device for Multi-Touch User Interfaces. 2010.

56. **Jürgo-Sören Preden**. Enhancing Situation – Awareness Cognition and Reasoning of Ad-Hoc Network Agents. 2010.

57. **Pavel Grigorenko**. Higher-Order Attribute Semantics of Flat Languages. 2010.

58. **Anna Rannaste**. Hierarcical Test Pattern Generation and Untestability Identification Techniques for Synchronous Sequential Circuits. 2010.

59. **Sergei Strik**. Battery Charging and Full-Featured Battery Charger Integrated Circuit for Portable Applications. 2011.

60. **Rain Ottis**. A Systematic Approach to Offensive Volunteer Cyber Militia. 2011.

61. **Natalja Sleptšuk**. Investigation of the Intermediate Layer in the Metal-Silicon Carbide Contact Obtained by Diffusion Welding. 2011.

62. **Martin Jaanus**. The Interactive Learning Environment for Mobile Laboratories. 2011.

63. **Argo Kasemaa**. Analog Front End Components for Bio-Impedance Measurement: Current Source Design and Implementation. 2011.

64. **Kenneth Geers**. Strategic Cyber Security: Evaluating Nation-State Cyber Attack Mitigation Strategies. 2011.

65. **Riina Maigre**. Composition of Web Services on Large Service Models. 2011.

66. **Helena Kruus**. Optimization of Built-in Self-Test in Digital Systems. 2011.

67. **Gunnar Piho**. Archetypes Based Techniques for Development of Domains, Requirements and Sofware. 2011.

68. **Juri Gavšin**. Intrinsic Robot Safety Through Reversibility of Actions. 2011.

69. **Dmitri Mihhailov**. Hardware Implementation of Recursive Sorting Algorithms Using Tree-like Structures and HFSM Models. 2012.

70. **Anton Tšertov**. System Modeling for Processor-Centric Test Automation. 2012.

71. **Sergei Kostin**. Self-Diagnosis in Digital Systems. 2012.

72. **Mihkel Tagel**. System-Level Design of Timing-Sensitive Network-on-Chip Based Dependable Systems. 2012.

73. **Juri Belikov**. Polynomial Methods for Nonlinear Control Systems. 2012.

74. **Kristina Vassiljeva**. Restricted Connectivity Neural Networks based Identification for Control. 2012.

75. **Tarmo Robal**. Towards Adaptive Web – Analysing and Recommending Web Users` Behaviour. 2012.

76. **Anton Karputkin**. Formal Verification and Error Correction on High-Level Decision Diagrams. 2012.

77. **Vadim Kimlaychuk**. Simulations in Multi-Agent Communication System. 2012.

78. **Taavi Viilukas**. Constraints Solving Based Hierarchical Test Generation for Synchronous Sequential Circuits. 2012.

79. **Marko Kääramees**. A Symbolic Approach to Model-based Online Testing. 2012.

80. **Enar Reilent**. Whiteboard Architecture for the Multi-agent Sensor Systems. 2012.

81. **Jaan Ojarand**. Wideband Excitation Signals for Fast Impedance Spectroscopy of Biological Objects. 2012.

82. **Igor Aleksejev**. FPGA-based Embedded Virtual Instrumentation. 2013.

83. **Juri Mihhailov**. Accurate Flexible Current Measurement Method and its Realization in Power and Battery Management Integrated Circuits for Portable Applications. 2013.

84. **Tõnis Saar**. The Piezo-Electric Impedance Spectroscopy: Solutions and Applications. 2013.

85. **Ermo Täks**. An Automated Legal Content Capture and Visualisation Method. 2013.

86. **Uljana Reinsalu**. Fault Simulation and Code Coverage Analysis of RTL Designs Using High-Level Decision Diagrams. 2013.

87. **Anton Tšepurov**. Hardware Modeling for Design Verification and Debug. 2013.

88. **Ivo Müürsepp**. Robust Detectors for Cognitive Radio. 2013.

89. **Jaas Ježov**. Pressure sensitive lateral line for underwater robot. 2013.

90. **Vadim Kaparin**. Transformation of Nonlinear State Equations into Observer Form. 2013.

92. **Reeno Reeder**. Development and Optimisation of Modelling Methods and Algorithms for Terahertz Range Radiation Sources Based on Quantum Well Heterostructures. 2014.

93. **Ants Koel**. GaAs and SiC Semiconductor Materials Based Power Structures: Static and Dynamic Behavior Analysis. 2014.

94. **Jaan Übi**. Methods for Coopetition and Retention Analysis: An Application to University Management. 2014.

95. **Innokenti Sobolev**. Hyperspectral Data Processing and Interpretation in Remote Sensing Based on Laser-Induced Fluorescence Method. 2014.

96. **Jana Toompuu**. Investigation of the Specific Deep Levels in *p*-, *i*- and *n*-Regions of GaAs p^+-*pin*-n^+ Structures. 2014.

97. **Taavi Salumäe**. Flow-Sensitive Robotic Fish: From Concept to Experiments. 2015.

98. **Yar Muhammad**. A Parametric Framework for Modelling of Bioelectrical Signals. 2015.

99. **Ago Mõlder**. Image Processing Solutions for Precise Road Profile Measurement Systems. 2015.

100. **Kairit Sirts**. Non-Parametric Bayesian Models for Computational Morphology. 2015.

101. **Alina Gavrijaševa**. Coin Validation by Electromagnetic, Acoustic and Visual Features. 2015.

102. **Emiliano Pastorelli**. Analysis and 3D Visualisation of Microstructured Materials on Custom-Built Virtual Reality Environment. 2015.

103. **Asko Ristolainen**. Phantom Organs and their Applications in Robotic Surgery and Radiology Training. 2015.

104. **Aleksei Tepljakov**. Fractional-order Modeling and Control of Dynamic Systems. 2015.

105. **Ahti Lohk**. A System of Test Patterns to Check and Validate the Semantic Hierarchies of Wordnet-type Dictionaries. 2015.

106. **Hanno Hantson**. Mutation-Based Verification and Error Correction in High-Level Designs. 2015.

107. **Lin Li**. Statistical Methods for Ultrasound Image Segmentation. 2015.

108. **Aleksandr Lenin**. Reliable and Efficient Determination of the Likelihood of Rational Attacks. 2015.

109. **Maksim Gorev**. At-Speed Testing and Test Quality Evaluation for High-Performance Pipelined Systems. 2016.

110. **Mari-Anne Meister**. Electromagnetic Environment and Propagation Factors of Short-Wave Range in Estonia. 2016.

111. **Syed Saif Abrar**. Comprehensive Abstraction of VHDL RTL Cores to ESL SystemC. 2016.

112. **Arvo Kaldmäe**. Advanced Design of Nonlinear Discrete-time and Delayed Systems. 2016.

113. **Mairo Leier**. Scalable Open Platform for Reliable Medical Sensorics. 2016.

114. **Georgios Giannoukos**. Mathematical and Physical Modelling of Dynamic Electrical Impedance. 2016.

115. **Aivo Anier**. Model Based Framework for Distributed Control and Testing of Cyber-Physical Systems. 2016.

116. **Denis Firsov**. Certification of Context-Free Grammar Algorithms. 2016.