CHALMERS
UNIVERSITY OF TECHNOLOGY

Assessment and development of robust software for communication in buses

Master’s thesis in Complex Adaptive Systems

JIM STENBERG

Department of Applied Mechanics
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2015

MASTER’S THESIS IN COMPLEX ADAPTIVE SYSTEMS

Assessment and development of robust software for communication in buses

JIM STENBERG

Department of Applied Mechanics
Division of Vehicle Engineering and Autonomous Systems
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden 2015

Assessment and development of robust software for communication in buses
JIM STENBERG

Master's Thesis 2015:14
ISSN 1652-8557
Department of Applied Mechanics
Division of Vehicle Engineering and Autonomous Systems
Chalmers University of Technology
SE-412 96 Göteborg
Sweden
Telephone: + 46 (0)31-772 1000

Cover:
Transparent image of a Volvo bus, model B11R Euro 6.
Reproduced with permission from Volvo Buses AB.

Department of Applied Mechanics
Göteborg, Sweden 2015

Assessment and development of robust software for communication in buses
Master’s thesis in Complex Adaptive Systems
Jim Stenberg
Department of Applied Mechanics
Division of Vehicle Engineering and Autonomous Systems
Chalmers University of Technology

Abstract

New quality demands combined with existing problems with software updates led to a desire of improved robustness in one of Volvo Buses’ control units. The code was analyzed by a static code analysis tool, tested and manual reviewed. A breakdown into five different fields (writing convention for the code, incorrect input, internal error, memory shortage and work overload) allowed for focused analysis. The gathered results were then used to design improvements for the unit, some implemented and some only evaluated.

The aim was to increase the unit’s robustness when communicating. The changes made improved the memory usage and calculated the work load more accurately. This led to less resets and a more robust system. However, if more robustness would be needed in the future, the current operating system would need to be replaced.

Key words: Robust, Software, C, Bus, ECU, Memory

Contents

Preface V

Acknowledgment V

Glossary VI

1 Background 7

1.1 Purpose 7

1.2 Scope 7

1.3 Contributions 7

1.4 Reading guidance 8

2 Theory 9

2.1 BBM 9

2.1.1 Endianness 9

2.2 Software robustness 10

2.2.1 Robustness in code 10

2.2.2 Robustness during execution 11

2.2.3 Protection outside software 11

2.2.4 Genetic Algorithm 11

2.3 ISO 26262 11

2.4 MISRA C 12

2.5 Communication 12

2.5.1 J1587 12

2.5.2 CAN 13

3 Method 15

3.1 Analysis 15

3.1.1 Incorrect input 15

3.1.2 Internal Error 15

3.1.3 Memory shortage 15

3.1.4 CPU overload 16

3.2 Improvements 16

4 Result 17

4.1 MISRA C compliance 17

4.2 Analysis 17

4.2.1 Incorrect input 18

4.2.2 Internal Error 18

4.2.3 Memory shortage 18

4.2.4 CPU overload 20

4.3 Improvements 21

4.3.1 Incorrect input 21
4.3.2 Internal Error 21
4.3.3 Memory shortage 21
4.3.4 CPU overload 23
5 Discussion 25
5.1 Biological inspiration 25
5.2 Communication 25
5.3 Standard 25
5.4 Improvements 25
5.4.1 OS 26
5.5 Future work 26
6 Conclusion 28
References 29

Preface

In this master's thesis, analysis and improvement of the software in an electrical control unit has been performed. The work aims to improve the unit's robustness. The project has been conducted from January 2015 to June 2015 at Volvo Buses AB, Arendal, Sweden.

Acknowledgment

This master's thesis was supervised by Per Grönroos at the Embedded Control Systems Group in the Division of Electrical and Electronics Development at Volvo Buses AB.

The grading was performed by Univ. Lektor Krister Wolff at the Division of Vehicle Engineering and Autonomous Systems in the Department of Applied Mechanics at Chalmers University of Technology.

Glossary

ISO	International Organization for Standardization *Organization providing standards*
ANSI	American National Standards Institute *Organization providing standards*
MISRA	Motor Industry Software Reliability Association *Organization providing programming guidelines for automotive industry*
SAE	Society of Automotive Engineers *Organization providing standards for automotive industry*
KWP	Keyword Protocol *A communication protocol*
CAN	Controller Area Network *A communication protocol*
ECU	Electronic Control Unit *physical unit (circuit board, ports and plastic shell)*
BBM	Body-Builder Module *The ECU in primary focus of this project*
MID	Module Identification *Integer value unique for each ECU*
PID	Parameter Identification *Integer value unique for each parameter*
OS	Operating System *Resource managing system in computers*
CPU	Central Processing Unit *Computer part managing the computations*
ALU	Arithmetic Logic Unit *Computer part performing the computations*
ECC	Error Correcting Code *Code that can detect and correct bit errors*
KiB	Binary prefixes of bytes [1] *KiB is 1024 bytes while kB is 1000 bytes.*

1 Background

In modern vehicles, electronics and control modules continue to grow in importance. Real-time communication between these components is an essential cornerstone which was often placed before systems became as large as they are today. To no surprise have these old but working parts been left alone until stronger regulations or added functionality require change.

Volvo Buses is currently working with preparations to meet a new quality standard which within a near future will be in effect throughout Volvo AB. As part of this work, the code for control modules has gone through quality verification. The Embedded Control Systems group found a weakness in one of their modules' communication during this process, and hence desired a solution. The problem was detected though the use of two static code analysis tools and therefore the knowledge of a problem exist but it is not known exactly what it is, nor its severity. A short overview pointed in the direction of insufficient error handling combined with a weak internal structure.

An effect, believed to originate from these problems, is that updating software in the aftermarket sporadically fails. The current solution is to simply reprogram the unit again, but this is obviously a quick fix which is unwanted in the long run. In order to fix this problem and prevent future ones, the overall wish is a more robust code.

1.1 Purpose

The aim is to improve the robustness for receiving and transmitting data in the affected module. Thereby eliminating the current error and reduce the likelihood of new ones. As well as allow the code to comply with the new standard, ISO 26262, which defines functional safety for electronic and electrical equipment in automotives throughout its life cycle.

1.2 Scope

This project concerns a single person and has a time frame from January 2015 to June 2015 which also includes documentation and presentation. It focuses exclusively on the troublesome ECU, namely the BBM. Hence this is a single case study. Therefore, it will prove difficult to motivate that any result may be applied on a general basis. The purpose is to improve the existing codes robustness rather than rewriting it from scratch, something that would not possible fit into the time slot available.

1.3 Contributions

By conducting the study and writing this report the outcome should deliver a more reliable BBM to Volvo Buses AB as well as ideas of future improvement with accompanying motivations. For those who want to do similar improvements the method chapter should provide a general framework that is applicable for embedded software written in C, although the remaining chapters contain more case specific information.

1.4 Reading guidance

This thesis report has been divided into three major parts theory (2), method (3) and result (4) after which discussion and conclusion follows. Theory describes both basic information and a bit more detailed information. To understand method and result one should focus on BBM (2.1) and robustness (2.2) in the theory.

In all three main chapters (theory, method and result) are four different topics (Incorrect input, internal error, memory shortage and CPU overload) reoccurring. If one wish to focus on only one topic it is fine to skip the others, but the discussion still mix them somewhat together.

2 Theory

This chapter provides theoretical knowledge as well as facts and definitions needed to follow and motivate the methods used later.

2.1 BBM

The BBM is an ECU and thus consist of a circuit board with components and connectors for inputs, outputs and communication channels, encased in a shell of plastic. The concept is illustrated in Figure 2.1. The architecture used has a little-endian format and on it is a real-time OS installed which currently runs over twenty different threads, also denoted processes. There is hardware support for analogue and digital inputs and outputs in both high and low voltage. It has three communication channels J1587, J1939 and D-Bus on which it communicates with other ECUs.

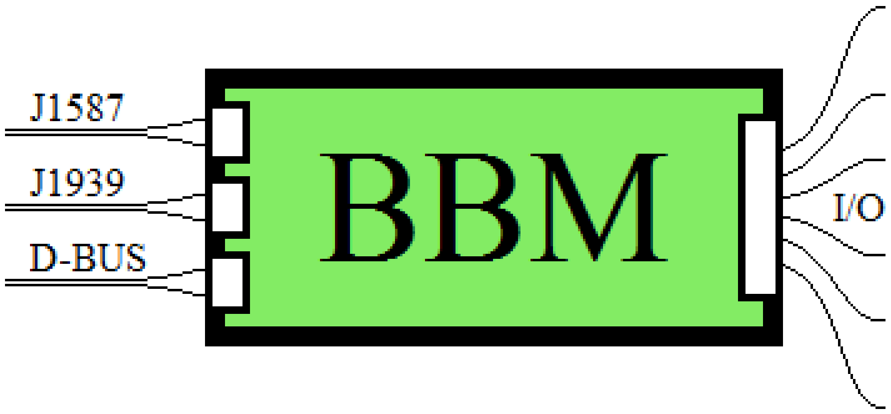

Figure 2.1 Illustration of the BBM's interfaces.

The BBM does not regulate many actuators, it instead specialize more on reading sensors and transmitting it to other ECUs. Since it is connected to three communication channels it also serves as a gateway for ECUs which would otherwise be unable to reach each other. When programming the BBM, one uses J1587 to transmit the new code.

2.1.1 Endianness

Big-endian and little-endian describe the two, by far, most common ways to arrange values which do not fit into a single byte. A big-endian machine store them with the big end first, also called most significant byte. Likewise does the little-endian principle place the least significant byte first. The concepts are more easily grasped by showing them as images, see Figure 2.2. Which way it is implemented is important when one shall transmit data on a channel, otherwise they may end up in the wrong order.

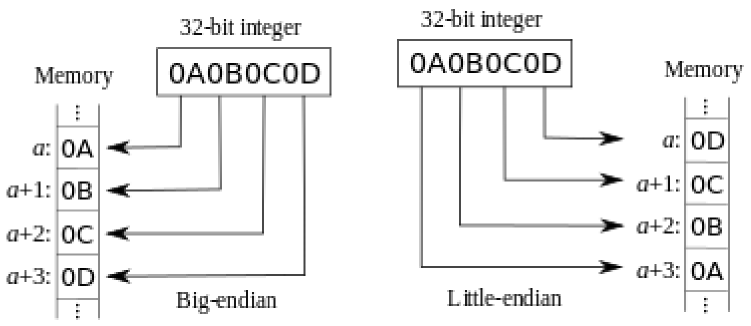

Figure 2.2 Both endianness concepts explained with an example. Source: [2].

2.2 Software robustness

Being robust means to have a strong constitution, i.e. resist perturbations. This translates to the ability to cope with errors during execution when considering software[3]. There are many types of errors and a generalized classification is needed, this can be done in different ways but here they are chosen to;

- Software induced and software solvable errors.
- Non-software induced but software solvable errors.
- Non-software induced and not software solvable errors.

The first group is often named programming errors, while the second is where the robustness truly acts and the third is rarely touched nor mentioned. Robustness should not be confused with resilience which is the ability to handle changes in an adaptive manner[4].

A descriptive comparison with biological systems is made by Sussman in[5] and here follows a brief reflection. Biological systems have a lot of robustness and are therefore suitable as an inspiration source. The most noticeable properties are redundancy and degeneracy. Redundancy in biological systems means that multiple sources provide the same service at an overcapacity such that if one fails, the system survives, e.g. only one lung or a diminishing food source. Degeneracy means that multiple sources provide different solutions to the same problem which have different operational range. Where they overlap, they shall be coherent, e.g. vision and hearing both provide localization information. A great thing with biological systems is that they are also adaptable which causes resilience.

2.2.1 Robustness in code

All software robustness is of course built into the written code, but there is one kind of robustness that is not visible in the completed, compiled code. That is the resistance to errors while programming, something that can be prevented by writing structured and predictable code. The structured part is most often covered by enforcing that all individuals involved follow a common coding standard. To make the code predictable usually takes a bit more effort. One way to increase predictability is to use MISRA C coding standard (described in Chapter 2.4).

2.2.2 Robustness during execution

Given that the functionality has been implemented as intended, i.e. no programming errors remain. Then there still exists a group of errors that occurs when one goes outside normal usage.

2.2.2.1 Incorrect input

Whenever there is an interface, there are almost always more potential inputs than there is valid. A robust system should be able to identify and handle unwanted input. The occurrence of such an event is rather common and may origin from a counterpart, human or machine, which is incorrectly using the interface or communicating with a third party on it.

2.2.2.2 Internal error

Internal errors are faults occurring when reading from memory or performing arithmetic in the ALU. These are rarely occurring and the average computer does not implement any general countermeasures against it. Places where countermeasures are taken are in life critical systems such as spacecraft but also where it can have large economic impact such as a miscalculation in a bank's system or cause downtime for a large data center[6].

2.2.2.3 Memory shortage and CPU overload

Memory shortage as well as CPU overload occurs when the software hits a hardware boundary. In the first case the space is no longer adequate to store all information and in the later the calculations can no longer be made in real-time. A robust system should be able to manage both situations, when they occur as well as working proactive to reduce the probability to reach them.

2.2.3 Protection outside software

There are many potential errors originating from other sources, e.g. low voltage or incorrect version of software or hardware. These might be detected but never corrected by the software itself. Hence managing of such problems is needed elsewhere.

2.2.4 Genetic Algorithm

A genetic algorithm is an optimization method that is inspired by biological systems. It utilizes the concepts of selection, mutation and crossover which are among the most recognized parts of the evolution theory. Genetic algorithms are in many ways an applied method. The greatest benefit is that it does not require any simplification of the problem. One only needs to be able to gather and weight all properties of a solution into a single value, denoted "fitness", to use it. A weakness is that a genetic algorithm does not guarantee it will find the best solution but instead finds a near-optimal solution fast, something that is often desired in real life problems. Other strong points are that it can make easy use of sampled data and be applied to almost any problems[7].

2.3 ISO 26262

ISO 26262 is a quality standard and one of the main reasons to why this thesis were initialized. A description of ISO 26262 can be found in the beginning of ISO 26262

[8], here follows the most vital points. ISO 26262 is a standard named “Road vehicles - Functional safety” and regards safety in systems comprised of electrical, electronic and software components. In contrast to its more general parent standard, IEC 61508, ISO 26262 is tailored for series production of passenger cars. In addition to providing requirements and processes for development, it also involves validation and supplier relations.

2.4 MISRA C

MISRA C is a set of guidelines for ANSI C aimed to improve reliability and portability in safety critical embedded software. It emphasizes avoidance of undefined or implementation defined behavior. The methodology is to restrict usage, down to a subset of the ANSI C which is predictable and safe, thereby reducing mistakes while programming [9].

The guidelines are divided into short and simple rules which in most cases are easy to check, e.g. one should never pass the minimum value of an integer to the absolute function, since the result is undefined. MISRA is especially tuned for embedded software and because of this it also bans certain coding methods e.g. the use of a variable to specify length of an array. Something which is very common to do when programming for general purpose machines like personal computers but could cause problems in embedded software where memory is limited and the programmer always should know how large an array can be.

It is generally impossible to comply with all rules due to use of external software and compact data storage. For this reason one are allowed to deviate from the rules, given that a written motivation accompany the deviation. Deviations are allowed to be made on case to case basis but it is strongly advised to have general motivations with group or even company-wide effect.

Since the set of rules are large, it is not recommended to manually find all these faults. A static code analysis tool, if well constructed, finds most of them and give a messages that makes it clear to the programmer what should be changed. However, the change must still be made manually, due to the tools inability to determine the intended functionality with absolute certainty. It is also stated that old code do not need to be rewritten unless a change is made, but it is still advised to do.

2.5 Communication

The communication protocols one need to know in order to be able to interpret the transmission between units encountered in this project.

2.5.1 J1587

J1587 is a protocol standard for serial communication of diagnostics provided and maintained by SAE. The role fulfilled by this protocol can also be done with KWP which is a competing standard by ISO.

Information included in the J1587 document defines how and what data should be sent[10]. Recommendations for the frequency of and circumstances in which messages are transmitted are also included.

The protocol operates bytewise and first position always contains the senders MID followed by one or more PIDs. The example in Table 2.1 first has a PID with a single data byte followed by a second with variable size and hence also includes length information. The J1587 leaves some flexibility by letting one specific PID indicate that the coming information is undefined by the protocol, thus allow for company specific messages. It also omits how the checksum should be calculated.

Table 2.1 An example message for J1587.

MID	PID	Data	PID	Length	Data 1	data 2	Data 3	Checksum
143	12	1	230	3	0	15	255	147

2.5.1.1 J1708

J1587 only handles the data structure and is therefore complemented with the SAE J1708 which handles the message basic structure. Per the J1708 definitions shall each communicated message consist of a MID character, one or more PID characters with accompanying data, finish with a one byte checksum and have a specified maximum message length which in most cases cannot be exceeded[11].

The message structure must also follow other definitions in J1708, which defines bit and byte timing and structure. These specifications are a modified version of the EIA RS-485 protocol.

2.5.1.2 RS-485

RS-485 official name is TIA-485 in its current release but the characteristics remain unchanged. The RS-485 uses two wires, if the "+" wire is a certain threshold larger than the "−" wire the state is one. The opposite situation results in a zero. If the wires are within the threshold the state is undefined. RS-485 uses a start and stop bit.

2.5.2 CAN

CAN is a protocol that use message IDs as priority information. A low numbered ID infers a high priority message. All units begin sending their messages simultaneously but only the one with highest priority get its message out. This is caused by the value zero have dominance over one on the channel. Combined with everyone else's bitwise reading of the channel, which will mismatch theirs broadcasting at least once while the priority bits are sent, causing them to become silent. A side effect of this construction is that every ID must be unique.

When receiving messages on a CAN channel, they are conventionally pre-processed by a CAN controller, which is dedicated hardware on the circuit board[12]. When the message has been parsed it triggers an interrupt in the processor, which allowing it to be used elsewhere. This part also handles the silencing when a more prioritized sender is detected. A final function of this dedicated hardware is that it adds and removes extra bits between too many consecutive bits of equal state. This is called "bit stuffing" and the inserted bit is of opposite state and serves as a clock synchronizer.

Like RS-485 and consequently J1587, CAN uses two wires. However, it is with a significantly different strategy. The idle state of CAN have both the "+" and "−" wire at the same voltage level. During a transmission the wires' voltages diverge to represent a dominant bit (0), while a recessive bit (1) is the same as the idle state. It is possible to have a single-wire CAN, but the two-wire solution removes some noise issues e.g. inductive spikes and electrical fields' interference[13].

2.5.2.1 J1939

The CAN protocol is not a complete standard, but needs to be complemented with other standards to be implementable. J1939 is such a standard and adds, among other things, that the message will be sent with little endian.

2.5.2.2 D-Bus

"D-Bus" is a name used at Volvo AB and is not a standard nor does it have any connection with the D-Bus standard related to Linux. In fact, it is actually a Volcano Network Architect, which is a complementing standard to CAN in the same regards as J1939. The noticeable difference is that it uses a big endian format, forcing a little endian machine to reverse the byte order of the data it wants to send. Notice that this is not equivalent to reverse the complete message, but the change depend on if the originating data is 2 byte, 4 byte, etc. long. Hence this cannot be taken care of by the CAN controller.

3 Method

The first action to be done was to write the code MISRA C compliant. This was done with the use of a static code analysis tool and chosen because each adjustment only has a narrow effect on the code and preferably no effect on the program. But more importantly because of one of the side effects of rewriting code, one familiarize with the code. Such approach allows for progress to be made while constructing a mental image of the concept used when writing the code. A mental image, if rather accurate, is something that improves the speed at which one can make later analysis and modifications. It is also vital for any kind of prediction, which is the basis for modifications.

3.1 Analysis

The second step was an extensive analysis of the code. It was divided into parts by type of errors. The concept of this analysis should be applicable to any ECU or similar system even though the resulting outcome and following actions can diverge greatly.

3.1.1 Incorrect input

There are two common ways of testing for incorrect input. One way being to follow the execution flow and try to find ways in which the program may but should not act. This way of analysis suffers from the same problem as when writing the original code, the human doing this may make mistakes. Faults in this type of analysis is often cause by tangled and complex logic. The alternative is to do a fuzz-test, in which one sends random, invalid or unexpected data.

On J1587 the messages have both a great variety in values and length, resulting in a huge amount of invalid messages that can be sent. When testing for rather unusual events, quite harsh restrictions are needed on the randomized part in order to get an acceptable time. To choose those restrictions, one needs good insight in the protocol and at such stage it becomes more efficient to follow the execution flow instead. A restricted fuzz-test could still be made after following the flow to verify that it was done correctly.

3.1.2 Internal Error

An analysis of the amount of internal errors should be considered greatly before conducted. This is because one can expect roughly 1 to 5 bit errors per year per MiB (converted from [6]). Hence it may be highly advantageous to extrapolate bit error frequencies from earlier studies.

3.1.3 Memory shortage

To get a clear view of the memory usage an overall measure was needed. This must be collected at multiple times in order to ensure that a representative value has been found. The measure should also be sampled with a few different speeds to ensure one does not miss events that occur with regular intervals. The distribution of memory among different processes also needed to be mapped. That information points toward the region of greatest improvement potential. These values give a current status view and can be used for proactive actions.

The behavior displayed when more and more memory is occupied, to the limit when no more memory is available holds great value since it is telling what countermeasures is implemented and how effective they are. This behavior can be provoked by temporary adding a function which in a controlled manner allocate more and more memory.

3.1.4 CPU overload

To determine the CPU load when the ECU is working with normal load, a function was added to broadcast CPU load on request. The load was calculated with Equation (3.1).

$$LOAD = 1 - \frac{Idle\ counter}{Theoretical\ max} \tag{3.1}$$

Where the idle counter is calculated in an infinite while loop with lowest priority and its theoretical maximum is the crystal frequency divided by number of clock cycles per idle loop. The counter and maximum must be valid for the same time span. This can be done in two ways; first is to scale the maximum with the time since last reading dynamically, while the other is to periodically perform a reading. Here the periodic approach was used. The counter should be protected against overflow, which for the chosen approach can be done with a reset of the counter at each reading. If not enough load can be simulated in the laboratory by requests alone, a dummy function will be added to increase stress on the CPU.

The load measurement cannot be used to observe the behavior during overload since it would be kept constant at its maximum. In this case the observation must rely on input response. How does the increase in work affect the response time, does it skip some messages and if it does skip some, which ones? For this purpose the request sent must have distinct responses and be logged with a timestamp.

3.2 Improvements

The actions for improvements were chosen based on the results from the analysis. Hence it cannot be fully described nor motivated until after the result of the analysis has been presented. This way, the solution becomes more tailored for the BBM but less general.

4 Result

This chapter covers the result from both the code rewriting to comply with MISRA C and robustness analysis. It also covers the attempted improvements and their results, due to these being a direct response to the analysis result.

4.1 MISRA C compliance

The effect brought on by rewriting the code MISRA C compliant had no influence on the BBM in a functionality perspective. Additionally, the change was admissible when viewed from a resource usage perspective.

The changes in the code consisted primarily of three things. The first was the replacement of pointer arithmetic to index offsets. Since its effect is clearer as well as easier to check for boundary conditions. The second was to eliminate all implicit cast between different types, which occur when changing type (e.g. int to byte) without stating it. This was mostly fixed by making the cast explicitly, indicating that the cast was a conscious action. The last of the main concerns was the interfaces towards external sources. Variables and functions defined in external code often used unconventional names and some could not be traced because they are indirectly declared.

A noticeable occurrence was a complaint from the static code analysis tool regarding potentially uninitialized variables, where they clearly were initialized. Suspicions are that the static code analysis tool used could not assert that a flag ensured that they were initialized before use. One would need a second tool to confirm this, unfortunately none was available. There was previously one competing tool available but its license was discontinued due to the superfluous of having two tools that perform the same task. A third alternative tool was earlier considered but had already been deemed insufficient.

4.2 Analysis

The analysis result will be presented in the same subfields used in Chapter 3.1 but first some results and notices of general nature will be put forth. Throughout the code one encounter some deprecated legacy code. The problem is that it is quite nested and therefore rather difficult to remove. The compiler does detect unused code on its own and hence most legacy code does not affect the final program, but it slows down the compile time and makes the effect of the code harder to interpret.

Another notice is that the BBM restart very fast. It is completely rebooted within 500 ms but can even perform most operations, such as message transceiving, faster than that. This does not affect robustness but it improves availability. Additionally it makes a reset harder to detect, if one does not request the reset status regularly it is possible to have a reset without noticing it. The most obvious indication is that the BBM forgets its state of whether or not all broadcasts has been disabled and would start transmitting again.

Out of the different subfields, memory shortage was the most imminent problem, followed on a safe distance by CPU load. Both internal error and incorrect input carry little importance when compared to these problems. Provoking a failed software update also showed that memory shortage was the cause of the failure.

4.2.1 Incorrect input

From following the flow of how an input message is handled, no flaws were detected. The code appears to cover exactly what is specified in the specification [10], rejecting neither more nor less inputs. However, the code for the communication channels does not take into account if the data of a parameter is valid since that would require knowledge of each specific parameter, which the it leaves to the specific functionality governing that parameter.

To increase the confidence of the analysis a fuzz test was performed. For total random tests, almost all cases tested that the checksum was working. When appending a checksum to the random message to ensure that it always passed, it would almost exclusively verify that the length was correct. In a few cases it actually managed to get through the checksum and length check, but that only proved that the normal case was working as intended.

One small deviation was found, but its effect was negligible. When sending a message which length is known, the length check is not performed. This allows one to manually append the checksum and add an arbitrary amount of zeros, up to the message limit. The only change the zeros cause was that the channel did not return to its default undefined state until after the zeros end.

4.2.2 Internal Error

Due to the low chance of occurrence no tests were conducted. The risk of a few errors each year is not a big problem since most of those errors would misinterpret a program instruction, a pointer or variable data. In both of the first cases the likelihood of a fatal error is prominent and lead to a reset. A reset is unwanted but taking into account the rare occurrence and fast reboot, it remains the most cost effective solution. For the case with a corrupted variable, the BBM would interpret the data as if it was noisy, since the BBM’s most common behavior is to send data periodically and such variable would be corrected at next update. A risk of getting a subtle memory corruption still exists but should be considered rare even among the events of error.

4.2.3 Memory shortage

The BBM only has the single most basic proactive measure against memory shortage, which is to reuse freed memory, and even in this case it is limited to a predetermined range within the released buffer (a set of consecutive memory positions) fit and may only occur after the buffer has been explicitly released. When provoking a memory shortage the BBM did not respond with any actions until the memory was full, at which it did a full reset.

The pool of memory is governed by an OS which allow for multiple allocations of 8 distinct predetermined buffer sizes. For a requested size, the OS returns the smallest predetermined size that is larger or equal to the requested one. The predetermined buffer sizes are declared in a configuration file before compiling. When returning a memory buffer the OS makes it available for use again, but only for the same predetermined size. Hence the OS guarantee that there will not be any gaps in the pool with the tradeoff that it becomes fragmented.

There are two sources that cause a memory allocation to consume more memory than requested in the given implementation. The first is a mismatch between requested and given buffer sizes, causing some of the accessed memory to become unutilized. The other is a header of 12 bytes which is claimed next to each allocated buffer. The data held in the header is shown in Table 4.1. First two bytes contain a fix known value, if any other value would be present, a memory corruption has occurred. Byte 2 through 5 is a fast access pointer which is only valid for freed buffers and points to the next free buffer of the same predetermined length, if available. Next two bytes contains the requested length, followed by the index of the predetermined length at which the buffer is fixated. The last three bytes contain the number of the process that currently owns the buffer, should receive the buffer (only when the buffer is in transmission between two processes) and the final byte is the creator. A buffer is considered free if both owner and receiver is 0.

Table 4.1 Header information.

Byte	**Description**	**Value**
0	Write protection	0xEE
1	Write protection	0xEE
2	Fast access pointer, lowest byte	Any
3	Fast access pointer, middle lower byte	Any
4	Fast access pointer, middle higher byte	Any
5	Fast access pointer, highest byte	0
6	Requested length, lower byte	Any
7	Requested length, Higher byte	Length < max size
8	Predetermined length index (shifted two bits left)	(0 to 7) * 4
9	Owning process	0 to # processes
10	Receiving process	0 to # processes
11	Creating process	1 to # processes

The BBM has an 8 KiB (minus 2 B) memory pool in which a programmable limit allows for 6 KiB to be used. The overall memory usage could be divided into two values, one for current usage and one for all memory that has been fragmented until now. The fragmented memory is an important measure since it represents a combination of all the past peeks since reboot. Hence this is the one measure which needs to be minimized in order to be as robust as possible.

The data was sampled with three different intervals; 500 ms, 5000 ms and finally 537 ms. One short and one long duration were used together to determine if there were any different time scale trends. The third time contained a high prime number (3*179) and was used to detect regularly occurring events, which would always occur between the samples for the previous times. The behavior was similar regardless of which time was used.

Upon observation of a few memory snapshots it was clear that the three processes used to interface each communication channel represented a large proportion of the allocated memory. This was expected since they must allocate space making it possible to receive the largest message possible. A supervising process stood for a noticeable amount of the remaining allocated memory. It did so by allocating a lot of 2 B buffers when verifying that all other threads were active.

The average memory usage ranges from 3 KiB to 3.5 KiB and fragmented memory of about 4.5 KiB as shown in Figure 4.1. The reason for declaring the average value in a range arises from the memory's strong correlation to load on the communication channels.

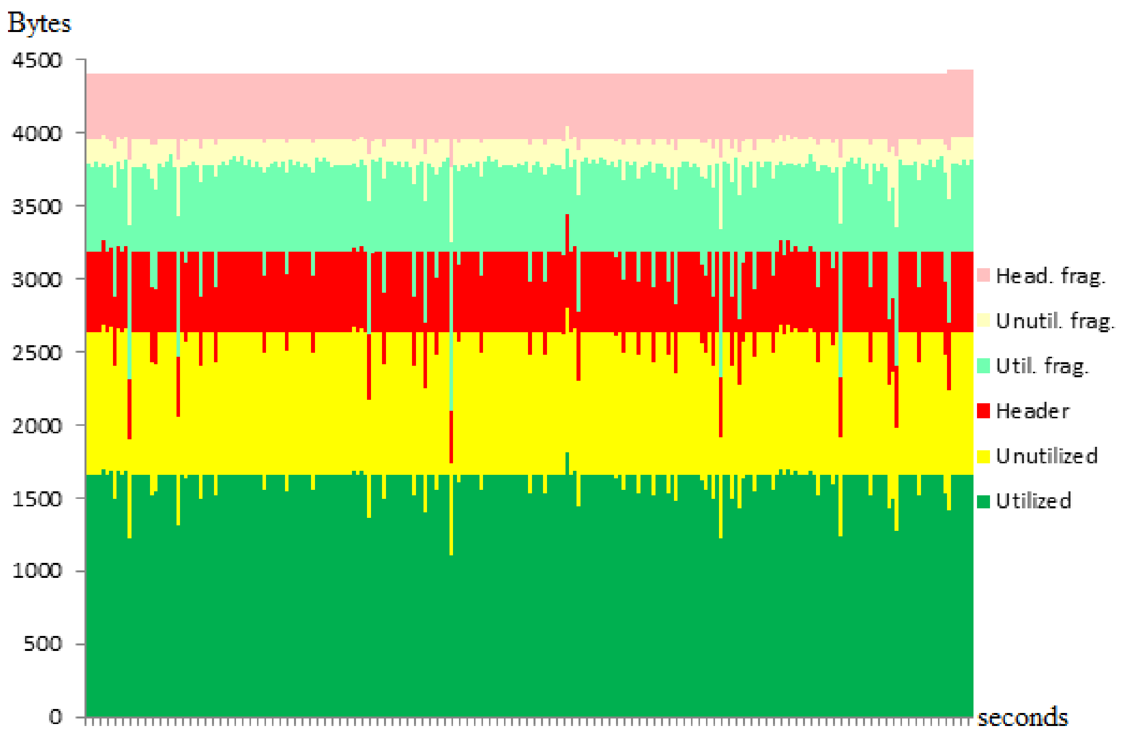

Figure 4.1 Memory pool, sampled twice a second, in laboratory.

It is a noticeable difference in memory when comparing tests in rig and readings from bus, about 1 KiB. Another strong difference exists between programming phase and normal execution. Execution in buses requires more memory than laboratory tests due to more ECUs being connected and communicating. The programming phase requires more memory than normal phase since it is the normal execution with an extension added on top. Hence the worst case scenario is programming in buses. Since the ECUs are programmed before mounted this do not occur in production, but it does when testing as well as when performing software updates on aftermarket. This was also determined to be the cause for the sporadically failures detected.

4.2.4 CPU overload

The analysis regarding load but not overload revealed that there already was an existing idle loop at the lowest prioritized process, but it suffered from remains of legacy code. These remains caused the idle loop to take 26 clock cycles, except for once every 60 000 iterations in which it took an additional 24 cycles. This is not something wrong, but it complicates things unnecessarily. Combined with a theoretical maximum set to a value 8% off the mark resulted in a rather inaccurate load estimate. Regardless of what the estimate's value show, it still indicates trends. A temporary function which increased the CPU load upon request was added. The CPU load as a function of time between increased load requests is listed in Table 4.2.

Table 4.2 Median CPU load.

Request time interval [ms]	**Off (∞)**	**1000**	**800**	**600**	**400**	**340**	**320**	**310**	**300**
CPU load [%]	40	63	63	74	86	86	86	Reset	Reset

The somewhat blocky result can be explained by the fact that the load measure was calculated once each second. Hence the most frequent amount of function calls per load calculation becomes the rounded value of calculation interval divided by request interval.

When exposed to an overload situation, the BBM discover that it cannot keep real-time execution and therefore restarts within 500 ms. In this process it loses the request it was currently processing in the failing thread but still have time to finish most of the others requests with higher priority. At the restart it also loses the current messages sent over the different communication channels. The strong point with this implementation is that while operational, the BBM always operate in real-time. The downside is that the BBM may never do calculations that take 500 ms or more to complete. The reset is ECU wide and no attempts to kill the specific thread or any other subset are made prior to reset.

4.3 Improvements

This chapter presents the solutions that have been attempted, combining both angle of attack and result. Since the chosen method may fail due to limits set by external software or likewise, it may become an iterative process and therefore cannot be divided in a clear way.

4.3.1 Incorrect input

No further actions in this topic were deemed necessary. One could potentially add the length check even to those messages whose length is known.

4.3.2 Internal Error

The sacrifice of memory and CPU load or cost of upgrading hardware were deemed higher than the benefit of correcting internal errors and therefore no further actions were taken. If one wished to protect the equipment from this type of problems, the counter would be to use a redundant system. Common solutions are ECC in form of either triple modular redundancy or Hamming code. Hamming code uses the smallest number of parity bits needed and can be used for both memory access and transmission while triple modular redundancy is faster in memory reading but slower in transmission, in addition is the theory also applicable for calculations[14, 15].

4.3.3 Memory shortage

To decrease the risk of running out of memory the simplest solution is to increase the pool size. Since the memory pool does not compete with anything else in terms of space, its programmable limit was raised to the hard limit. This alone solved the current problem with sporadic failures on aftermarket.

The second step was to change the buffer sizes to minimize the fragmented memory. A handmade suggestion was constructed for reference purpose by looking at a snapshot of the memory pool's current buffer sizes' requested lengths and amounts. The same type of data, although in larger quantity, where used in a genetic algorithm to get a (near) optimal solution. For heavy load in the laboratory the average memory usage decreased to somewhere between 2 KiB and 2.5 KiB while the fragmented memory ended up in the range from 3.5 KiB to 4 KiB, as shown in Figure 4.2.

The supervisor was rewritten to iterate through the processes, asking one at a time if it was alive, rather than all at the same time, this led to the need of a single 2 B buffer allocation, compared to one for each process. The most noticeable improvement is the reduction in header which again is 12 B for each buffer, further reducing the fragmented memory usage by about 0.3 KiB.

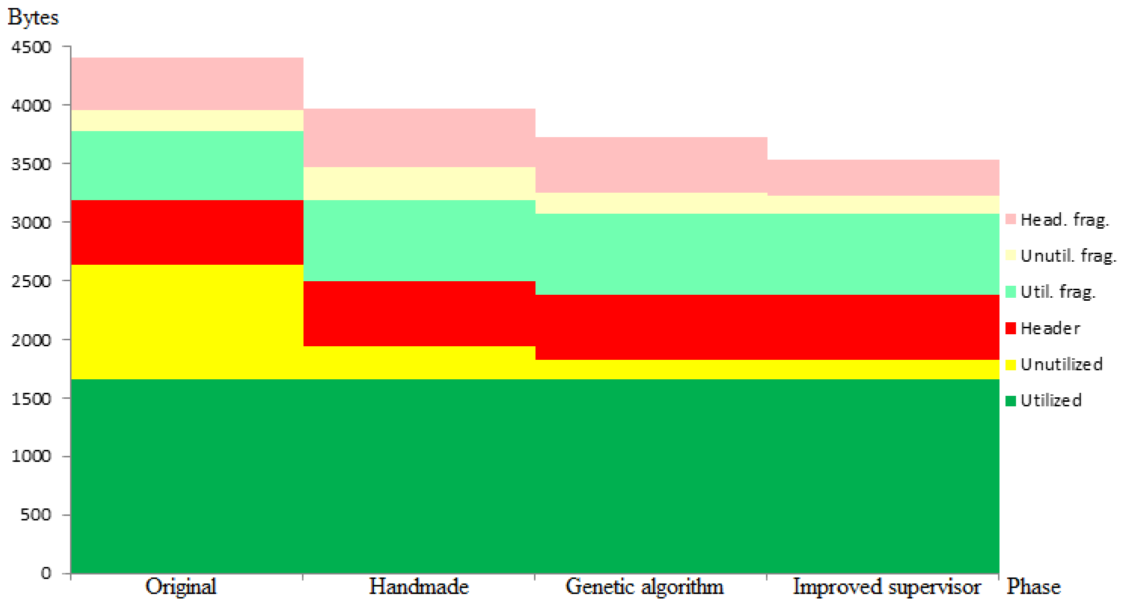

Figure 4.2 Memory Improvement, different code with same settings, in laboratory.

In order to give the BBM an alternative reaction to a full memory, some of the messages were rewritten to be discarded when not enough memory was available rather than forcing the whole BBM to reset. This was implemented by checking the buffer allocation for a null pointer return and then take appropriate actions. If the message was an important one, there might be no escape but to reset. At least now, the choice to reset is clearly a conscious one, compared to a side effect.

The current memory handler uses a 12 bytes header, as shown in Table 4.1. This amount to approximately 20% of the accessed memory and therefore one would benefit from minimizing it. When considering what is stored in the header, one starts to question its usefulness. Beginning at the end of the header, byte 11, the creator is saved, this information should not be needed for anything else but debugging. Moving over to owner and receiver, There will not be a receiver and an owner at the same time. Hence, they should be able to share one single space. In 3 bits in byte 8 lies a most vital piece of information, the buffer size index, which without one would not be able to know where the buffer ends and next start. The requested length, which may be of great value while debugging, holds nothing desirable at normal execution. The fast index pointer certainly speed up allocation but when one consider that the BBM is under heavier memory constraints than CPU, one cannot justify the speed improvement at the cost of 4 B header it takes. The remaining bytes protect against writing beyond the buffer size. Although useful, the effect can be condensed into the remaining 5 bits of the current byte 8. The 3 bits holding the length information should probably be shifted up to spread the valid possibilities and the 5 write protection bits should then be constructed in a way that they exclude the values 0 and 0xFF, since they are very commonly occurring. In summary should the header be possible to shrink from 12 B to 2 B thus reducing the total memory used with one sixth. The new version of the header is shown in Table 4.3. However, this improvement cannot be done due to it requiring modifications in the OS, which is a bought software without source code.

Table 4.3 Improved header information.

Byte	Description	Values
0	Length and write protection	(0 to 7) * 32 + (1 to 30)
1	Owner	0 to # processes

To remove the problematic memory fragmentation entirely one would need another way to handle the memory, i.e. another OS. There are quite a few solutions which shift some of the constraints. Three of these were considered, one that rearrange the buffers and two that merge the freed space. The rearranging method always uses the minimum amount of bytes needed but since it may move a buffer, its pointer must be considered volatile. For the OS to be able to change the pointer, it must be stored too (preferably last in the pool). The volatile property results in that each read or write occasion would take longer time. The two other methods both merge freed space by checking a freeing buffers adjacent space. What separates them is the method of reallocation. The first uses fixed buffer sizes with an internal relation such that they always will be able to fill the freed space completely. Easiest logic is where each buffer size, including header, is a multiple with a power of two of the other buffers. A side effect would be that the amount of accessed but unused bytes would increase. The last solution assumes that all buffers are temporary and hence a small unallocable space would be of no problem, since it soon will be merged again. It still allows for allocation of permanent buffers, such as hand shaking signals and alike, by allocating them elsewhere (potentially last in the pool). The problems arising in such a solution is that there will be a minimum size of the free space due to length information which needs to be saved and if the buffers are allocated in an unfortunate order, the allocation might temporary become larger than the current solution.

4.3.4 CPU overload

The idle loop was changed to take 10 clock cycles. Hence the crystal frequency becomes evenly divisible by the number of iterations per second. This change does not decrease the load of the CPU but it simplifies the load calculation and makes it more accurate. Combined with a correctly calculated theoretical maximum number of iterations it revealed that the load normally stays around 30% in laboratory tests with heavy message transmission.

An improvement from the current solution when detecting CPU overload would be if the BBM only restarted the responsible process rather than the entire software. For most of the current processes, this would be quickly implemented. However there are some key processes for which this would require a total restructure. The reason behind this is that when a process can reset at any time, it impose constrains on the way in which the processes can be written. It may never hold states or expect another process to register itself at startup. A saving property is that this can be implemented on only a selection of the processes. This was not implemented due to one last critical fact, one need to clear the process's memory and reset its program pointer which is properties known only to the OS.

If one ever desire to calculate a more demanding calculation, which exceeds the limitation of 500 ms. One would need to add a new process with priority second to last (last being the idle loop) and avoid checking it. The problems arising by doing so is that it cannot be verified against lockups and if requests were to arrive faster than resolved. Its queue would result in a memory reservation, almost equivalent to a memory leak. Finally, one would not be able to keep real-time execution since the queue length is unknown and the request may be handled in seconds or minutes. Due to the nature of the BBM's purpose it is unlikely that functionality similar to this would ever be desired.

5 Discussion

Here follows a discussion regarding the different steps and results in the project. It also contains personal thoughts and ideas of how to proceed if more time were available.

5.1 Biological inspiration

Drawing inspiration from biological systems was mentioned in theory, Chapter 2.2. Yet no such occurrence is suggested among the viable improvements. This stems from the fact that both redundancy and degeneracy creates robustness at the cost of resources, something that is an unnecessary expense from a corporate's point of view and hence should be removed whenever possible. When such robustness does occur, it is most often due to law requirements, safety criticality or in cases where improved performance can be directly observed, e.g. a checksum appended to messages sent over a channel guard against transmission errors.

Another strong point with biological systems are their resilience. Due to the nature of the BBM's tasks it is not something applicable, storing and sending messages are very deterministic. However if one would want to apply redundancy and degeneracy to an ECU it is important to be aware of that some of the more complex methods like neural networks lack proper analysis methods. One should also note that an adaptive construction diverge each individual. This makes it harder to provide support for all active products since such feature may mask other faults like worn down hardware and alike. It would most likely also be blamed for a lot of problems, related or not. However, a certain amount of machine learning at the right place can do wonders.

5.2 Communication

The method and result chapters nearly don't touch the subject of communication protocol, even though it is explained in theory. The reason for this is that it is an underlying knowledge which is necessary to have when conducting the analysis, but it would not affect the analysis method. The result is already case dependent regardless of which communication protocol being used. Hence it is bundled into the rest of the case dependencies and not mentioned separately.

5.3 Standard

The main point for software developers in ISO 26262 was that one should use a coding standard like MISRA C. Since MISRA C allows old code to remain if unchanged it infers that ISO 26262 was already reached in this aspect. This annihilated part of the purpose, but since changes were made in this project these was accompanied with a conversion to follow MISRA C.

5.4 Improvements

Although different improvements have been evaluated, not all are suitable for implementation while some other remain unknown. In the field of incorrect input an extra length check could be added, but since it does not add anything significant it is a task with low. Likewise do not any measures against internal error need to be taken. In regards of CPU load would thread-separated termination be desired since it adds robustness without much cost. Such an action should be implemented if the OS were changed to one that allowed for it. Calculations with long evaluation time, on the other hand, should be avoided since it opens up for new risks and vulnerabilities.

The largest topic of improvement was memory. The reason for this was because it was where the current problem originated from, meaning that some improvements only shift the balance between the memory and CPU usage. Those who use the approach of shifting balance are the different memory handling methods. A problem with determine if they are better suited for the BBM's case or not is that they need to be implemented, something which was outside the timeframe of this project. Since the current memory handling solution is rather simple and effective, although a bit wasteful in terms of header size, it is likely that optimizing it is a better approach. Other improvements aim to reduce the memory while leaving as small footprint as possible, such as optimizing the buffer sizes with a genetic algorithm and changing the supervisor. The favorable result of these is that their side effects are usually small.

One of the challenges in this project has been that the pre-existing code was quite extensive and contained protected parts. This caused many possible solutions to be rejected simply by the sheer work required or that important properties was unavailable. This is a problem which affects both proactive and reactive measures which would have needed to been included at the concept phase.

5.4.1 OS

When consider improvements, there exists a lot of limiting factors. Economy being the dominant one, otherwise robustness could be easily implemented on more powerful hardware. Focusing on more uncommon constraints, the largest factor has proven to be the OS, primarily because it fragments the memory and dedicates a rather large part of it to header information. Another reason is that it hides some information, like the program pointers.

In the current state of the BBM it is not advised to replace the OS. This is due to a change today is a comparison between "As is" and "cost saving" and not worth doing. The cost saving alternative would be to lower hardware specifications. This would lead to new design, test, contracts, software interface, etc. and involve a lot of people and cost money. Because of this, the payoff time would be very long. If one wait until the need for more memory and CPU becomes dire the motivation is reversed. The change of OS is then a comparison between "hardware upgrade" and "OS replacement". Even if the old cost saving alternative is exactly the same suggestion as this OS replacement, it now have all those old factors as savings instead of costs, since replacing the OS at this point result in an avoided hardware change. If one would start to look for a replacement it is advantageous to use an open-source alternative if available. This provides both an economical benefit as well as giving the freedom to study, extract or change properties of the system, although this should not normally be needed.

5.5 Future work

A continuation from this study should be to conduct an investigation of existing suitable OSs. Another interesting angle would be to determine which factors would motivate more redundancy, although not very technical, it would still prove very useful when discussing funding.

There are also things to take into account when developing the current code further. The MISRA C is not implemented throughout the entire code, but simply where changes have been made. Since making the complete code comply with MISRA C would take a considerable amount of time, it is recommended to only be compliant in the parts where changes are made, incrementally improving the code rather than doing it all at once.

6 Conclusion

The analysis provided some alarming result. In principle the only way the BBM had to handle errors was to reset. Measures were taken to reduce the probability to reach a reset, but no countermeasures other than resetting exist when hitting a limit. The cause for this was the OS which was the overall largest constraint, if one ignores the obvious one, economy. This was due to the information it hides from the programmer. A change of OS cannot be motivated now, but if a change would occur at a later time it is recommended to use an open source, if possible.

The BBM was detected to reach its memory limit. Hence the memory pool was increased and the buffers allocated within had their sizes adjusted to occupy as little space as possible. The supervising process was rewritten to reduce the peak usage. The CPUs load was not an issue but the calculation that estimated it was corrected. An improvement of handling CPU overload separate for each thread was suggested but not implemented due to OS limitations. Likewise were the internal errors only analyzed but not improved, in this case, the reason was economical. The input was the only part where no improvement was deemed necessary. Overall was redundancy sought but not implemented. This was due to the resource requiring nature, which is in direct contrast to an economic standpoint.

The coding standard MISRA C was applied where changes were made. This allow for fulfilling the aspects of ISO 26262 that was applicable.

References

[1] Wikipedia, "Binary prefix", 20 April 2015. Available: en.wikipedia.org/wiki/Binary_prefix. [Accessed 24 April 2015].

[2] Wikipedia, "Endianness", 29 March 2015. Available: en.wikipedia.org/wiki/Endianness. [Accessed 24 April 2015].

[3] Wikipedia, "Robustness", 5 March 2015. Available: en.wikipedia.org/wiki/Robustness. [Accessed 30 March 2015].

[4] G. Monti, "Resilience Engineering #1: Robust Vs. Resilient", Active Garage, 7 June 2011. Available: www.activegarage.com/series/resilience-engineering.

[5] G. J. Sussman, "Building Robust Systems", Massachusetts Institute of Technology Computer Science and Artifical Intelligence Laboratry, 13 January 2007. Available: www.csail.mit.edu.

[6] B. Schroeder, E. Pinheiro and W.-D. Weber, "DRAM Errors in the Wild: A Large-Scale Field Study", SIGMETRICS/Performance, 2009. Available: www.cs.toronto.edu/~bianca/.

[7] M. Wahde, "Biologically Inspired Optimization Methods: An Introduction", WIT Press, 2008.

[8] Technical Committee 22 Road vehicles, "ISO 26262-1:2011 Road vehicles -- Functional safety -- Part 1: Vocabulary", International Standardization Organization, 2011. Available: www.iso.org.

[9] MISRA, "Guidelines for the Use of the C Language in Critical Systems", Motor Industry Software Reliability Association, March 2013. Available: www.misra.org.uk.

[10] Truck And Bus Low Speed Communication Network Committee, "Electronic Data Interchange Between Microcomputer Systems in Heavy-Duty Vehicle Applications", Society of Automotive Engineers, 4 January 2013. Available: standards.sae.org/j1587_201301.

[11] Truck And Bus Low Speed Communication Network Committee, "Serial Data Communications Between Microcomputer Systems in Heavy-Duty Vehicle Applications", Society of Automotive Engineers, 9 December 2010. Available: standards.sae.org/j1708_201012.

[12] Wikipedia, "CAN bus", 28 April 2015. Available: en.wikipedia.org/wiki/CAN_bus. [Accessed 4 May 2015].

[13] Axiomatic Technologies Corporation, "What is CAN", 6 July 2006. Available: www.axiomatic.com/whatiscan.pdf. [Accessed 4 May 2015].

[14] Wikipedia, "Hamming code", 21 April 2015. Available: en.wikipedia.org/wiki/Hamming_code. [Accessed 28 April 2015].

[15] Wikipedia, "Triple modular redundancy", 27 April 2015. Available: en.wikipedia.org/wiki/Triple_modular_redundancy. [Accessed 28 April 2015].