Final report

IN3405 Bachelor project

Conversion and extension of system software for the Forze IV hydrogen race car

Authors:	*Student number:*	*Mentor Forze:*
Alexander van Gessel	1287974	P. Danneels
Mark Provo Kluit	1263099	
Jan Jaap Treurniet	1308351	*Mentor TU:*
		H.G. Gross
		Coordinator:
		B.R. Sodoyer

Contents

1 Introduction

As team members of the Hydrogen Racing Team Delft have participated in the development and building of the fourth generation Forze fuel cell powered racing vehicle. The vehicle was intended to compete in the Formula Student UK class 1A competition for alternative fuel formula cars[1]. The electronics in the car have been fully redesigned, and is based on Cortex-M0 and Cortex-M3 microcontrollers. This means the old software had to be renewed so that it could run on these new microcontrollers. The conversion and extension of this software is the subject of this bachelor project performed by the authors. This final report describes how we fulfilled this goal, the experience we acquired, and the extra work we did that was not part of the original assignment.

We explain the problem and the requirements in Section 2. Given the fact that this bachelor project involves an embedded system — a fuel cell powered racing vehicle — means that we thought it would be useful to perform a hazard analysis, which can thus be found in Section 2.5. The specifications of the drivers and the various ECU's (Electronic Control Units) can be found in Section 3. Designs of the API's of the drivers, the ECU's, and the CAN protocol can be found in Section 4. The quality and testing of the software is described in Section 5. The porting of the old software and writing of the new software, along with a description of our work on the fuel cell software — which was not part of the original assignment — is written in Section 6. The development proces and the problems we encountered is described in Section 7. Finally, the results, and conclusions and recommendations can be found in Section 8 and Section 9.

We would like to thank our mentor from the Forze team, Pieter Danneels, and our mentor from the TU Delft, Hans-Gerhard Gross, for their help and support during this project.

[1]http://www.formulastudent.com/class1a/aboutclass1a.aspx

2 Problem statement and analysis

This section describes the problem in Section 2.1. The techniques we have employed are described in Section 2.2. Section 2.3 describes the research we did at the beginning of the project. The requirements we gathered while working on the 'Plan of Approach' can be found in Section 2.4. A hazard analysis in Section 2.5 lists the causes of potential hazards and what can be done to mitigate these.

2.1 Problem statement

The Forze IV's design includes more than 10 boards of various types that contain microcontrollers of the families Cortex-M0 and Cortex-M3. An RTOS has been supplied, but for the operation of the vehicle, the RTOS needs to be configured, drivers for various hardware need to be written and the applications for these microcontrollers need to be written.

The goal is to deliver the software to make the vehicle ready to race and communicate through the telemetry node. The full original project description can be found in Appendix A.

2.2 Techniques

The code in this project is written in C. The Segger embOS is used as Real-Time Operating System for both the M0 and M3 boards, programmed using J-Links. The Segger emFile file system is used to write log files to a microSD card. IAR Embedded Workbench IDE has support for detecting MISRA C violations. This tool also has support for the RTOS used, so that it can show a list of running tasks and their stack usage. This is useful to detect possible stack overflows during testing.

2.3 Research

At the beginning of our bachelor project we did research on a topic that was relevant to our subject of this project; the system software of a racing vehicle. The topic of the research concerned the "usage of coding standards for embedded programming in automotive applications". We choose this topic because the project assignment required that all software that would be part of the racing vehicle would comply to the MISRA C coding guidelines where possible. The IDE tool we used, IAR Embedded Workbench IDE, has support for detecting violations of rules of the coding guidelines.

We concluded our research that the MISRA C coding guidelines could have a positive effect on the quality of the software, although the code changes to fix violations of certain MISRA C rules could, ironically, introduce new errors. We also researched AUTOSAR, which is becoming a de-facto standard in the automotive industry, but we concluded that it would be too extensive and cost too much time to base our software on AUTOSAR.

The research described in the 'Orientation report' can be found in Appendix B.

2.4 Requirements

In this paragraph we describe the requirements as given and gathered from interviews with the other team members involved with the hardware parts. These requirements have been taken from Section 2.4 and 2.6 of the document 'Plan of Approach', but have been classified according to MoSCoW method. Additionally, process requirements have been added, which were not explicitly mentioned in the original requirements, and various components of the assignment — as described in Section 2.4 of the document 'Plan of Approach' — have been added to the list of requirements below.

2.4.1 Quality requirements

Must have

- Throttle nodes and dashboard controller should be able to handle hardware and software exceptions, including illegal CAN messages.

Should have

- Drivers should be designed in such a way that the API is reuseable on different microcontrollers.

2.4.2 Functional requirements

Must have

- Configure the RTOS, Segger embOS for the ARM Cortex-M0 and Cortex-M3.
- Design and write the following drivers for the M0 and/or M3:

UART	Used on all M3 and M0 boards for configuration, debugging, retrieving data, etc.
CAN	Used by all M3 and M0 boards for communication
ADC	Used by some general-purpose and sensor nodes
DAC	To actuate the motor controllers by the throttle pot node
PWM	To control various hardware by the Fuel Cell Controller and to turn on/off the brake light by one of the wheel speed nodes
SPI	Used by the DAC driver on the M0 and various general-purpose nodes of the Fuel Cell

- Design and write the throttle node application, more specifically:
  - Two general-purpose nodes (throttle nodes) each read a throttle pot and send the values over the CAN2 bus. The values of the two throttle pots are compared so that the power can be cut if the values differ too much or are not within their expected range (See Figure 1)
  - One of the throttle nodes also uses the read value to actuate the motor controllers
- Design and write the dashboard controller application, more specifically:
  - The dashboard application reads the input from the buttons that are controlled by the driver, and gives status information about the fuel cell and other systems, via LED displays

Should have

- Dashboard controller must log data to a microSD card, more specifically:
  - Configure microSD card driver and logging system, using Segger emFile
- Design and write the software for the telemetry node, more specifically:
  - Being able to transmit sensor data
  - Use the JenNet network protocol stack to transmit the data over the 2.4 GHz band

Could have

- Design and write extra software for the telemetry node, more specifically:
  - Transmitting a part or the complete data log from the microSD card
  - Receiving commands that can be send over the CAN bus, so that the vehicle can be configured or commanded to perform a certain function

Would be nice to have

- Extend the throttle node software to have the ability to do launch control

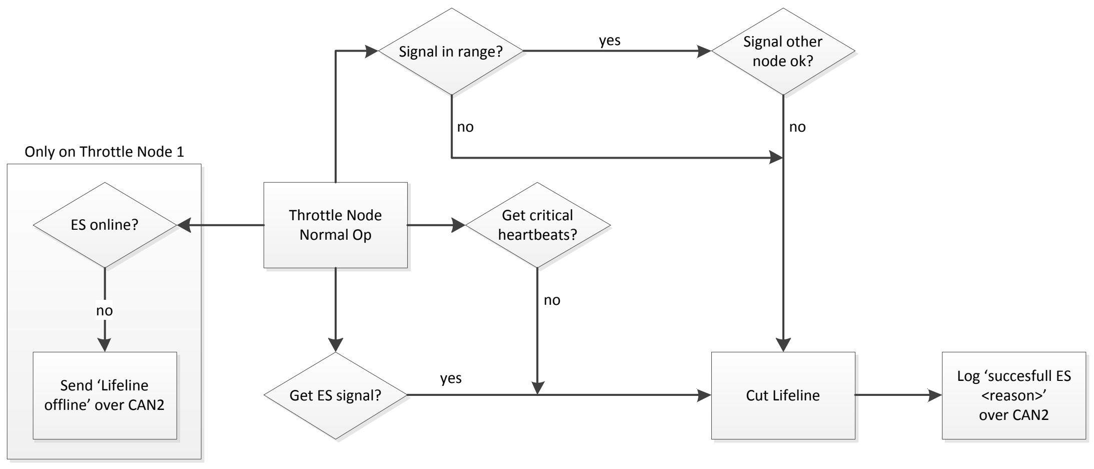

Figure 1: Throttle node software overview [1].

2.4.3 Platform requirements

Must have

- Drivers must work on both the Cortex-M0 (LPC11C14) and/or Cortex-M3 (LPC1768) microcontrollers, more specifically, following drivers must work on the M0:
  - ADC, DAC, SPI, UART, CAN, and PWM

  Following drivers must work on the M3:
  - ADC, SPI, UART, and CAN

2.4.4 Safety requirements

Must have

- Throttle nodes must cut the lifeline if the values of the two throttle pots read by the two throttle nodes differ too much, or are out of safe range.

Should have

- All software should comply with the MISRA C guidelines.

2.4.5 Process requirements

Must have

- Deadlines of various parts of the project must be met as described in Section 3.4.1 of the document 'Plan of Approach'.

2.5 Hazard Analysis

The nature of this bachelor project — creating software for embedded systems of the Forze IV, a racing vehicle using hydrogen as fuel — means people could get hurt if things go wrong. While a lot of potential problems can be attributed to hardware, some of the problems may be caused by the software. We have performed a hazard analysis to identify possible problems and how these can be mitigated.

2.5.1 Throttle nodes

Car accelerates when it gets powered on while standing still

- Causes:
  - Could be caused by an incorrectly configured minimum value of the throttle pot.
  - Even if the minimum value of the throttle pot is set to a correct value, the car can still accelerate if the motor controllers are not adjusted for the minimum analog value fed by one of the throttle nodes, which is 0.2 Volt. This is because the black box — provided by the organization of FSUK — requires the voltage to be between 0.2 and 4.8 Volt.
- Mitigations:
  - This hazard is mitigated by the motor controllers, which will refuse to work if a signal is fed by one of the throttle nodes while the car is being powered on.
  - Motor controllers need to be adjusted for the minimum voltage, which is 0.2 Volt.

Car does not accelerate linearly while increasing the throttle linearly

- Causes:
  - The ADC and DAC hardware may not provide sufficient linearity or resolution.
  - Software converting the digitized analog signal to the appropriate digital value — ready for conversion to an analog signal between 0.2 and 4.8 Volt — may cause loss of accuracy.
- Mitigations:
  - There are few things that can be done to mitigate this issue, other than choosing better hardware, which is in our case no option for the Forze IV. One thing that can be done is measuring the linearity so that it is at least known where the driver can expect any non-linearity.

2.5.2 Right rear wheel speed node

Brake light is severly dimmed or is not turned on at all while the driver is applying the brake or enabling regenerative braking

- Causes:
  - This could be caused if the brake pressure is not properly calculated by the right rear wheel speed node or if the dashboard controller does not send on time or not at all the correct CAN message when the driver presses one of the two buttons that are designated for regenerative braking.
- Mitigations:
  - The calculation of the brake pressure in software must be verified to be correct. Furthermore, the minimal brake pressure value must be determined during testing. The format of the CAN messages for regenerative braking must be established and one must verify that both the dashboard controller and the right rear wheel speed node adhere to it. In case the right rear wheel speed node uses PWM to turn on the brake light, one must verify that it applies the correct PWM signal.

2.5.3 Dashboard controller

Dashboard displays a speed that is much higher than the actual speed of the car

- Causes:
  - This could be caused if the revolutions per minute of a wheel is not properly measured or if this number is converted to KM/h in such a way that it results in an incorrect speed or just loss of accuracy. Another possible cause is that if the dashboard controller has multiple sources that it could use to calculate the actual speed, that it uses the measured speed of the wrong wheel or just does not take effects like slip into account.
- Mitigations:
  - It must be verified that the number of revolutions per minute of a wheel is measured correctly and that the conversion is done correctly and does not lead to loss of accuracy. To our current knowledge, only the speed of the right rear wheel is measured; this makes the problem of determining which wheel speed to use a non-issue, but also means that effects like slip of the rear wheels cannot be mitigated.

3 Specifications

This section describes the working of the drivers and the various ECU's in more detail than some of the requirements found in Section 2.4. The specifications of the drivers can be found in Section 3.1. The specifications of the ECU's in Section 3.2.

Key words signifying requirement levels The key words "MUST", "MUST NOT", "SHOULD", "SHOULD NOT", and "MAY", in this document are to be interpreted as described in RFC 2119.[2]

3.1 Drivers

Drivers for several hardware found in common MCU's need to be designed and written. These drivers must work on the Cortex-M0 (LPC11C14) and/or Cortex-M3 (LPC1768) microcontrollers, but should be MCU agnostic as much as possible. The platform requirements, found in Section 2.4.3, defines which drivers must work at least on which microcontrollers.

Additional drivers During the project, it became clear that it would be nice to create specifications and design API's for some reuseable software and some hardware found in the vehicle:

PID	General PID controller. Used by the fuel cell software.
MAX6675	The MAX6675 is a temperature sensor found on the PCB of the general-purpose nodes. The temperature that it measures can be received via the SPI bus. Used by the fuel cell software.
HLS-440	The HLS-400 is a device that measures the amount of hydrogen near the fuel cell and broadcasts this information on a CAN bus. Used by the throttle node; the throttle node must kill the lifeline if the amount of hydrogen detected by the HLS-400 is above 1%.
LED	The dashboard consists of a lot of LED's, connected to four LED drivers controlled via SPI. The software uses the LED driver to turn on and off the LED's.

Except the LED driver, all of the drivers specified above must work on the Cortex-M0 (LPC11C14). The LED driver must work on the Cortext-M3 (LPC1768). The PID and HLS-440 drivers are not dependent on the hardware of the general-purpose nodes and should be useable on the Cortext-M3 (LPC1768) platform.

3.1.1 ADC

Functionality

- The ADC driver allows one to convert an analog signal — associated with a so-called 'channel' — to its digital representation.
- The meaning of a 'channel' is up to the microcontroller specific part of the driver and MAY correspond with certain pieces of the hardware of the microcontroller.
- The driver MUST provide an API to start a new conversion of a specific channel, and to return the last converted digital value of a specific channel.
- The LPC1768 and LPC11C14 specific part of the driver MAY provide an API to configure the hardware of the ADC unit.

[2] http://www.ietf.org/rfc/rfc2119.txt

- If the driver is requested to convert an analog signal of a specific channel, the driver MUST NOT accept a channel number not supported by the microcontroller and SHOULD raise an error.
- If requested to convert an analog signal of a specific channel, the driver MUST return a number that is in range of the supported ADC unit.
- If the client requests the digital value of the last conversion of a specific channel for which no analog signals have been converted yet, then the value returned by the API MUST be undefined. This means clients SHOULD NOT assume that the analog signal of a channel has been converted at least once.
- The driver MAY block for an indefinite amount of time if the client requests a new conversion of the analog signal of a specific channel.
- If the hardware provides multiple ADC units, then the driver SHOULD NOT block a request for the conversion of the analog signal of a specific channel if the conversion can be handled by a free ADC unit; that is, by an ADC unit not currently converting the analog signal of another channel.

Quality

- Response time should be as low as possible but MAY depend on scheduling by the RTOS.

3.1.2 DAC

Functional

- The DAC driver converts a digital value to an analog signal — associated with a so-called 'channel'.
- The meaning of a 'channel' is up to the microcontroller specific part of the driver and MAY correspond with certain pieces of the hardware of the microcontroller.
- The driver MUST provide an API to convert a digital value to an analog signal for a specific channel, and to disable and enable specific channels repeatedly throughout the lifetime of the application.
- The LPC1768 and LPC11C14 specific part of the driver MAY provide an API to configure the hardware of the DAC unit.
- Any request to convert a digital value for a specific channel MUST NOT be accepted while the channel is disabled and the driver SHOULD raise an error.
- The state of the analog signal of a disabled channel is defined by the microcontroller specific part of the driver. It can float or be pulled to a certain voltage level by the microcontroller implementation or hardware. The microcontroller specific part of the driver MAY provide an API to configure the voltage level for the 'enabled' and/or the 'disabled' state.
- The client SHOULD assume that each specific channel is enabled by default, but the client MUST NOT assume that a specific channel's analog signal is set to a certain voltage after startup or after the client has enabled a specific channel. The microcontroller specific part of the driver MUST make sure that each specific channel is enabled after startup.
- If the client requests the driver to convert a digital signal for a specific channel, the driver MUST NOT accept a channel number or a value representing the digital signal not supported by the targeted microcontroller and SHOULD raise an error.
- After the driver has converted a digital value to an analog signal for a specific channel, it MUST maintain that analog signal for an indefinite amount of time, until the client requests a conversion of a different digital value or disables the channel.
- The driver MAY block for an indefinite amount of time if the client requests a new conversion of the digital signal of a specific channel.

Quality

- Response time should be as low as possible but MAY depend on scheduling by the RTOS.

3.1.3 SPI

Functional

- The SPI driver sends, receives, or exchanges messages over the SPI bus.
- The driver MUST NOT not handle the possibility of multiple slave devices connected to a bus. This is because it would make the driver highly dependent on a certain PCB if it handles this possibility. It is up to the client to make sure only one of the slave devices is enabled.
- The driver MUST provide an API to send a message, to receive a message, or to exchange a message; that is, sending and receiving a message at the same time.
- A 'message' is a sequence of bits and its length MUST be defined by the microcontroller specific part of the driver.
- The driver MUST provide an API to configure the clock polarity, clock phase, and the frequency of the transmission.
- The driver MUST NOT accept a bus number that is not supported by the microcontroller specific part of the driver and SHOULD raise an error.
- The driver MAY block for an indefinite amount of time while sending and/or receiving a message.

Quality

- Response time should be as low as possible but MAY depend on scheduling by the RTOS and will depend on the frequency of the transmission and length of the message that is being sent or received.

3.1.4 UART

Functional

- The UART driver reads and writes characters over a UART channel.
- The driver MUST provide an API to read and write single characters over UART channels provided by the PCB.
- A 'message' is a sequence of characters with a length provided by the client.
- The driver SHOULD provide an API to write entire messages.
- The driver MAY block for an indefinite amount of time when reading or writing.
- The driver MAY provide an API to allow the client to determine whether reading or writing would cause the driver to block.

3.1.5 CAN

Functional

- The CAN driver sends and receives CAN messages over one or more CAN buses.
- The driver MUST provide a parameter in each part of its API to select the CAN bus to use.
- A ‘message’ is an 11-bit ID plus a 0-8 byte data-field.
- The driver MUST provide an API to send CAN messages.
- An ‘event’ is a method to inform tasks of I/O progress and is provided by the RTOS.
- The driver MUST provide a method for tasks to indicate which CAN ID’s they are interested in and which events they wish to receive to be informed of incoming messages.
- The driver MAY provide a method for tasks to de-register for these CAN ID’s.
- The driver MUST provide a method for tasks to retrieve CAN messages that have been received.
- The driver SHOULD NOT block when sending CAN messages.
- The driver MUST NOT block when receiving CAN messages.

3.1.6 PWM

Functional

- The PWM driver converts a digital value to a PWM signal with the given duty cycle.
- The driver MUST provide an API to set the duty cycle.
- The driver MUST be able to control at least 2 channels simultaneously.
- The driver SHOULD have a resolution of at least 1024.

3.1.7 PID

Functional

- The PID driver generically calculates output values based on setpoints and measurements.
- The driver MUST provide an API to set the P, I, and D values.
- The driver MUST provide an API to update the setpoint.
- The driver MUST provide an API to generate a new output value based on a new measurement.

3.1.8 MAX6675

Functional

- The MAX6675 driver receives the temperature measured by the MAX6675 chip over the SPI bus.

- The driver depends on the SPI driver and MUST internally initialize the SPI bus to which the MAX6675 chip is connected on the PCB of the general-purpose nodes. The driver MAY assume that the client does not interfer by using the specific SPI bus used by the driver.
- The driver MUST provide an API to read the latest measured temperature.
- Because the MAX6675 chip returns the temperature at the thermocouple's hot junction which is between 0 °C and 1023.75 °C, the API MUST NOT return a number — representing the temperature — less than zero.
- The driver MUST configure the clock polarity and clock phase according as specified in the datasheet of the MAX6675 chip. The frequency of the transmission MUST be configured to be less than the maximum frequency specified in the datasheet.

Quality

- Response time should be as low as possible but MAY depend on scheduling by the RTOS and will depend on the frequency of the transmission and length of the SPI message that it receives.
- The client SHOULD NOT request a reading within 250 milliseconds after the previous reading. This is because, according to the datasheet of the MAX6675 chip, a conversion takes at most 220 milliseconds.

3.1.9 LED

Functional

- The LED driver controls LED's via chips using SPI on the dashboard.
- The driver depends on the SPI driver and MUST internally initialize the required SPI bus. The driver MAY assume that the client does not interfer by using the specific SPI bus used by the driver.
- The driver MUST provide an API to turn on or off a single LED.
- The driver MUST provide an API to turn set all LED's to a certain value in one call.
- The driver MUST NOT turn on more than 16 LED's at the same time. This is because of hardware limitations of the PCB of the dashboard controller.

3.1.10 HLS-440

Functional

- The HLS-400 is a device which can measure the amount of hydrogen in the air and can report this in a specific format over the CAN bus.
- Receiving the CAN messages is not the responsibility of the driver.
- The driver MUST provide an API that allows the client to hand over the data of the CAN message — which is 8 bytes — in order to extract information out of it later.
- The driver MUST provide an API for the client to retrieve the hydrogen concentration, message counter, sensor status, and sensor CAN ID.
- The driver MUST NOT, for any of the components, return a number that is outside the range as specified in the datasheet of the HLS-400. For the hydrogen concentration, this range is from 0.0% to 4.4%.

3.2 ECU's

This section specifies the working of the various ECU's.

3.2.1 Throttle nodes

Functional

- The vehicle contains two throttle nodes.
- Each throttle node MUST regularly read its own throttle pot and perform several safety checks: whether the value of the throttle pot is within a predefined range, and within a certain predefined distance of the value of the other throttle pot, read by the other throttle node.
- A throttle node MUST send the read value of its throttle pot over the CAN bus, so that the other throttle node can compare the values of the two throttle pots.
- The primary throttle node MUST convert the digital value of its throttle pot to a corresponding analog signal between 0.2 Volt and 4.8 Volt and send this analog signal to the motor controllers if the digital value is valid according to the performed safety checks.
- Each throttle node MUST kill the lifeline if the digital value is invalid according to the performed safety checks.
- The primary throttle node MUST regularly check whether the lifeline is online or offline and MUST send a notification over the CAN bus when the status changes, and MAY regularly send a notification, even when the status has not changed.
- Each throttle node MUST regularly send its 'heartbeat' over the CAN bus.
- A 'heartbeat' is a CAN message with a CAN ID that is unique to the device that sent it.
- Each throttle node MUST continuously listen for heartbeats sent over the CAN bus by all devices.
- Each throttle node MUST, during the start-up phase, send a notification over the CAN bus that it has received the heartbeats from all devices once it has received the heartbeats of all of them. Each throttle node MUST, after its start-up phase, kill the lifeline if it has not received a heartbeat of some device within a certain time.
- Each throttle node MUST listen for command to kill or activate the lifeline, sent over the CAN bus.
- Each throttle node SHOULD broadcast over the CAN bus when it has tried to kill or activate the lifeline.
- Each throttle node MUST regularly measure the speed of its associated front wheel and broadcast this speed over the CAN bus.

3.2.2 Wheel speed nodes

Functional

- The vehicle contains two wheel speed nodes.
- Each wheel speed node MUST regularly send its 'heartbeat' over the CAN bus.
- A 'heartbeat' is a CAN message with a CAN ID that is unique to the device that sent it.

- Each wheel speed node MUST regularly measure the speed of its associated rear wheel and broadcast this speed over the CAN bus.
- The right rear wheel speed node MUST continuously listen for special messages sent over the CAN bus that indicate that the driver has enabled or disabled regenerative braking. Based on the CAN message indicating whether 0%, 50%, or 100% regenerative braking should be applied, the right rear wheel speed node MUST send the appropriate analog signal to the motor controllers.
- The right rear wheel speed node MUST regularly measure the brake pressure and turn on the brake light if the brake pressure is higher than a predefined minimum pressure or if the driver has enabled regenerative braking.

Quality

- The right rear wheel speed node SHOULD turn on or off the brake light as soon as possible once it has determined that the status of the brake light should change because of the measured brake pressure or the received CAN messages indicating the status of regenerative braking.

3.2.3 Dashboard controller

Functional

- The vehicle contains one dashboard controller.
- The dashboard controller MUST regularly send its 'heartbeat' over the CAN bus.
- A 'heartbeat' is a CAN message with a CAN ID that is unique to the device that sent it.
- The dashboard controller MUST display the speed in KM/h of the vehicle based on the speed of the four wheels received over the CAN bus.
- The dashboard controller MUST display the status of the electronics via the "Electronics OK" LED based on certain received CAN messages sent by the throttle nodes indicating that a throttle node has received all heartbeats or has detected that the lifeline is offline.
- The dashboard controller MUST turn off the "Electronics OK" LED if it has not received the heartbeat of one of the throttle nodes within a certain time, and SHOULD send a command to stop the fuel cell stack to the fuel cell controller over the CAN bus.
- The dashboard controller MUST display the status of the "Lifeline OK" LED based on certain received CAN messages sent by any of the throttle nodes.
- The dashboard controller SHOULD display the reason on the speed LED's when it receives a CAN message from any of the throttle nodes indicating that it has killed the lifeline.
- The dashboard controller MUST display whether the fuel cell stack is operating or not via the "Fuel cell running" LED and whether the fuel cell controller is standing by via the "Fuel cell standby" LED based on certain received CAN messages sent by the fuel cell controller.
- The dashboard controller MUST display the states of various components in the car using LED's: voltage of boostcaps, temperature of coolant (water), and the pressure and temperature of the hydrogen tank.
- The dashboard controller MUST measure whether the driver is pushing the "Activate lifeline" pushbutton and send a command to activate the lifeline over the CAN bus to the throttle nodes accordingly.

- The dashboard controller MUST measure whether the driver is pushing the “Stop lifeline” pushbutton and send a command to kill the lifeline over the CAN bus to the throttle nodes accordingly.
- The dashboard controller MUST measure the status of the “Start/stop fuel cell” flipswitch and send a command to start or stop the operation of the fuel cell stack over the CAN bus to the fuel cell controller. The dashboard controller MUST also measure the status of the two other flipswitches to send commands to the fuel cell controller: prime/standby and cooldown/standby.
- The dashboard controller MUST measure the status of the two pushbuttons indicating whether the driver wants to apply 50% or 100% regenerative braking and send the status over the CAN bus to the right rear wheel speed node.
- The dashboard controller SHOULD request diagnostics from the motor controllers via the CAN3 bus and send this forward over the CAN2 bus.

3.2.4 Telemetry node

Functional

- The telemetry node sends various data over a wireless UART to a basestation in the pits.
- A ‘heartbeat’ is a CAN message with a CAN ID that is unique to the device that sent it.
- The telemetry node MUST regularly send its ‘heartbeat’ over the CAN bus.
- The telemetry node SHOULD send all relevant data to the basestation in the pits.
- The telemetry node MAY accept commands from the basestation in the pits.

4 Design

This section gives a detailed design description of the software developed in the project. We start with an overview of the system in Section 4.1. Section 4.2 shows the API's of the drivers we implemented. In Section 4.3 an list of the different ECU's in the car and their tasks can be found. Section 4.4 shows the CAN protocol used for communication between these ECU's via the CAN bus.

4.1 System overview

An overview of the system is shown in Figure 2. The fuel cell and dashboard controllers are boards containing an ARM Cortex-M3 microcontroller (LPC1768), which is a 32-bit MCU operating at up to 100 MHz. The ARM Cortex-M0 microcontroller (LPC11C14) operates at up to 50 MHz and is used by the other components; the general-purpose and sensor nodes.

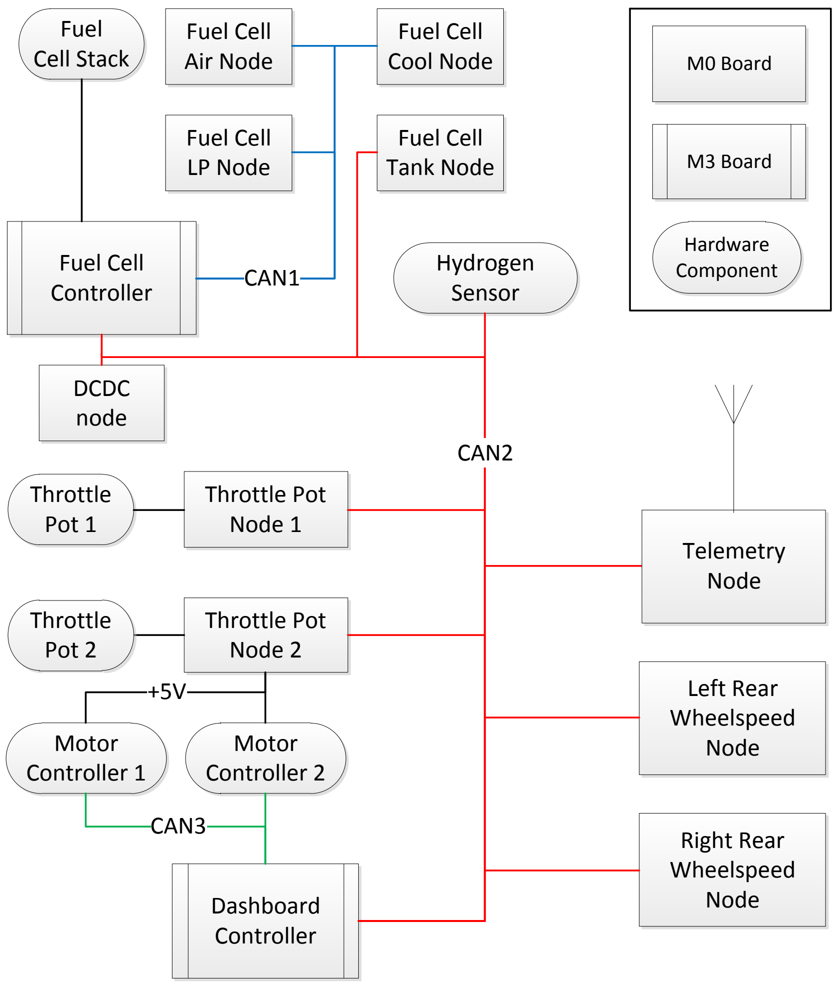

Figure 2: Overview of the system. M3 boards contain an ARM Cortex-M3 microcontroller. M0 boards are smaller and contain a Cortex-M0 microcontroller.

4.2 API's of drivers

This section shows the API's of the drivers we implemented.

PWM

```
/****************************************************************************\
 * Initialize the chosen PWM channel.
 * Must be called before any of the below functions can be used.
\****************************************************************************/
void pwm_init(unsigned int channel);

/****************************************************************************\
 * Set PWM value for the given channel. Value can be an integer between 0
 * and 65535.
\****************************************************************************/
void pwm_set(unsigned int channel, unsigned int value);
```

LED

```
/****************************************************************************\
 * LED Driver
 * This driver controls the LEDs connected to the four TLC5925 LED
 * controllers on the dashboard.
\****************************************************************************/

/****************************************************************************\
 * This function initializes the LED drivers.
\****************************************************************************/
void led_init(void);

/****************************************************************************\
 * Turn the given LED on or off. id can be 0 to 63. status should be 0 to
 * turn off, 1 to turn on.
\****************************************************************************/
void led_set(unsigned int id, unsigned int status);

/****************************************************************************\
 * Set status of all LEDs. Each bit in status sets the status of a led.
\****************************************************************************/
void led_set_all(unsigned long long status);

/****************************************************************************\
 * Get the status of the given LED.
\****************************************************************************/
unsigned int led_get_status(unsigned int id);

/****************************************************************************\
 * Get status of all LEDs.
\****************************************************************************/
unsigned long long led_get_status_all(void);
```

ADC

```
/****************************************************************************\
 * Initialize the ADC.
 * Must be called before any of the below functions can be used.
\****************************************************************************/
void adc_init(void);

/****************************************************************************\
 * Return a new value converted by the ADC. This function blocks until
 * the ADC has completed the conversion.
 * The maximum delay is:
 * <channels active> * <clocks per conversion> / <clockspeed of ADC>
 * M0 example: 3 * 11 / 4 MHz = 8.25 us
\****************************************************************************/
unsigned int adc_read_new_value(unsigned int channel);

/****************************************************************************\
 * Retrieves the last value converted for this channel.
 * This value is *not* guaranteed to be recent, or even to be converted at
 * all.
 * This function is entirely dependent on adc_read_new_value for updates.
\****************************************************************************/
unsigned int adc_get_last_value(unsigned int channel);
```

SPI

```
/****************************************************************************\
 * Initialize the SPI bus.
 * CPOL and CPHA defines the clock polarity and phase with respect to
 * the data. The configuration of the polarity and phase depends on what
 * the slave expects. You can usually find the information for this in the
 * datasheet of the slave device.
 *
 * CPOL:
 * 0 means clock is low between frames and high during frames
 * 1 means clock is high between frames and low during frames
 *
 * CPHA:
 * 0 means data lines are stable during first change of clock and switch
 * during second change of clock
 * 1 means data lines change during first change of clock and are stable
 * during second change of clock
 *          _   _
 * CLOCK: _/ \_/ \_
 *
 * CPHA = 0:  ___
 * DATA:  ___<___>_
 * CPHA = 1:___
 * DATA:  _<___>___
\****************************************************************************/
void spi_init(unsigned int bus, unsigned int cpol, unsigned int cpha);

/****************************************************************************\
 * Send an SPI message over the bus.
\****************************************************************************/
void spi_send_message(unsigned int bus, struct spi_message message);
```

```
/***************************************************************************\
 * Receive an SPI message sent by a slave over the bus.
\***************************************************************************/
struct spi_message spi_receive_message(unsigned int bus);

/***************************************************************************\
 * Send and receive an SPI message received and sent, respectively, by
 * a slave over the bus.
\***************************************************************************/
struct spi_message spi_exchange_messages(unsigned int bus, struct spi_message
message);
```

DAC

```
/***************************************************************************\
 * Initialize the DAC.
 * All channels are disabled by default. Before a digital value can be
 * converted to an analog signal for a channel, it first needs to be enabled.
 * Must be called before any of the below functions can be used.
\***************************************************************************/
void dac_init(void);

/***************************************************************************\
 * Set a new digital value to be converted to an analog signal by the
 * DAC unit.
 * The channel must not have been disabled or else an error might
 * potentially be raised.
\***************************************************************************/
void dac_write_new_value(unsigned int channel, unsigned int digital_signal);

/***************************************************************************\
 * Enable the channel.
\***************************************************************************/
void dac_enable_channel(unsigned int channel);

/***************************************************************************\
 * Disable the channel. After calling this function, any call to
 * dac_write_new_value() will potentially raise an error until the channel
 * is enabled again.
\***************************************************************************/
void dac_disable_channel(unsigned int channel);

/***************************************************************************\
 * Return whether the channel's analog output is set to a certain voltage
 * level by a call to dac_write_new_value().
 * If the channel has no analog value, then the channel either has been
 * disabled or enabled but dac_write_new_value() hasn't been called yet.
 * In this case one should not trust the value returned by
 * dac_get_value(); the implementation may either let the analog output
 * float or pull it towards some voltage level.
\***************************************************************************/
unsigned int dac_has_analog_value(unsigned int channel);

/***************************************************************************\
 * Get the current digital value of the DAC unit.
 * Preferably one should call dac_has_analog_value() first and if it returns
\***************************************************************************/
unsigned int dac_get_value(unsigned int channel);
```

TS

```
/***************************************************************************\
 * Initialize the thermo couple driver.
 * Must be called before any of the other functions can be called.
\***************************************************************************/
void ts_init(void);

/***************************************************************************\
 * Read a value from the thermocouple and returns it. Should not be called
 * with a frequency higher then 4 Hz, because conversion can take up to
 * 220 ms.
\***************************************************************************/
unsigned int ts_read_new_value(void);
```

HLS-440

```
/* Data structure to make it easier to extract specific information
 * from the body of the HLS-440's CAN message.
 */
typedef union {
    struct {
        /* byte 0 */
        unsigned int h2_concentration : 8u;

        /* byte 1 */
        unsigned int protection_value_1 : 8u;

        /* byte 2 */
        unsigned int : 2u;
        unsigned int ack_upper_limit : 3u;
        unsigned int ack_lower_limit : 3u;

        /* byte 3 */
        unsigned int crc16_byte_1 : 8u;

        /* byte 4 */
        unsigned int protection_value_2 : 2u;
        unsigned int sensor_can_id : 3u;
        unsigned int : 3u;

        /* byte 5 */
        unsigned int message_counter : 8u;

        /* byte 6 */
        unsigned int : 1u;
        unsigned int sensor_status : 2u;
        unsigned int part_number : 5u;
        /* byte 7 */
        unsigned int crc16_byte_2 : 8u;
    } read;
    struct {
        unsigned long long value : 64u;
    } write;
} hls440_message;
```

CAN

```
/*****************************************************************************\
 * This function initializes the CAN interfaces.
 * It must be called before any of the other functions can be used.
\*****************************************************************************/

void can_init(unsigned long bitrate1, unsigned long bitrate2);
/*****************************************************************************\
 * Add an event for the given CAN bus and CAN id. The event will be signaled
 * when a message with the specified id is received.
 * Mask 0x7FF (all bits) means only that specific id.
 * Mask 0x0 (no bits) means catch-all.
 * Mask 0x0 (no bits) means catch-all.
 * A message only triggers one event. Narrower masks precede wider ones.
 * Adding an id/mask pair that already exists is considered an error.
 * Multiple received messages are not guaranteed to generate more than 1 event.
 * Multiple received messages with the same id are overwritten by the latest message.
\*****************************************************************************/
void can_add_event(unsigned int bus, unsigned int id, unsigned int mask, char event, OS_TASK* task);

/*****************************************************************************\
 * Remove the given event for the given CAN id.
\*****************************************************************************/
void can_remove_event(unsigned int bus, unsigned int id, unsigned int mask, char event);

/*****************************************************************************\
 * Send a message on the specified CAN bus.
\*****************************************************************************/
void can_send(unsigned int bus, unsigned int id, char *data, unsigned count);

/*****************************************************************************\
 * Read the last received message for the given CAN id.
 * id points to the integer matching the id from the event.
 * Due to masking, the actual id read can differ, so it gets written into *id.
 * When no message can be found, UINT_MAX is returned.
 * Multiple received messages are not guaranteed to generate more than 1 event.
 * Multiple received messages with the same id are overwritten by the latest message.
 *
 * The recommended way to call can_read() is this:
 *
 * event = OS_WaitEvent(my_event);
 *
 * if (event & my_event) {
 *     can_id = can_id_block;
 *     can_mask = can_id_mask;
 *
 *     while (can_read(CAN_BUS_ID, &can_id, can_mask, data) != UINT_MAX) {
 *         -- Do something here with the data
 *         can_id = can_id_block;
 *     }
 * }
 * Note: the statement can_id = can_id_block at the end of the while loop is
 * not necessary if can_mask is equal to CAN_MASK_ALL, because then can_id
 * will always be the same.
\*****************************************************************************/
unsigned can_read(unsigned int bus, unsigned int *id, unsigned int mask, char *data);
```

UART

```
/*****************************************************************************\
 * This function initializes this UART interface.
 * It must be called before any of the other functions can be used.
\*****************************************************************************/
void UART0_Init(unsigned long baud);

/*****************************************************************************\
 * Read a single character from the UART channel.
\*****************************************************************************/
char UART0_Getchar(void);

/*****************************************************************************\
 * Write a single character to the UART channel.
\*****************************************************************************/
void UART0_Putchar(char cdata);

/*****************************************************************************\
 * Write the specified number of characters to the UART channel.
\*****************************************************************************/
int UART0_Write(const char *data, int length);

/*****************************************************************************\
 * Returns whether calling UART_Getchar would return immediately instead of blocking.
\*****************************************************************************/
unsigned char UART0_RxAvailable(void);

/*****************************************************************************\
 * Flushes the read and write buffers.
\*****************************************************************************/
void UART0_Flush(void);
```

4.3 Tasks of ECU's

This sections describes which tasks run on the various ECU's and what each task does. The functionality of the fuel cell nodes is not described here, because when we started working on this parts, we did not have enough time left to write them down as much in detail as the other parts of the system.

4.3.1 Dashboard controller

The dashboard controller presents information to the driver and communicates requests from the driver to other parts of the system. A listing of the tasks that run on the dashboard controller can be seen in Table 1.

Like all other ECU's, the dashboard controller sends a 'heartbeat' — over the CAN2 bus — 10 times per second. This heartbeat is mainly meant for the two throttle nodes, but is also received by the fuel cell controller.

In the task "Receive CAN" the dashboard controller listens for a lot of CAN messages like messages indicating the status of the lifeline, heartbeats of some ECU's, and sensor values. It turns on the "Electronics OK" LED when it receives a message from one of the throttle nodes that it got all the heartbeats, and it turns the LED off when it has not received the heartbeats from the two throttle nodes within a certain time. It does something similar with the "Fuel cell running" and "Fuel cell standby" LED's depending on the messages it receives from the fuel cell controller. The task updates the LED's of the hydrogen tank pressure and hydrogen tank temperature based on messages sent by the hydrogen tank node. The 32 LED's indicating the voltage of the boostcaps are updated when it receives the message containing the voltage level sent by the DCDC node. The LED's of the coolant (water) temperature are updated when receiving the appropriate message from the cool node.

As a debugging feature, the "Receive CAN" task starts and retriggers several timers when certain CAN messages arrive. For example, if the dashboard controller receives a "Successful activate lifeline" message from one of the throttle nodes, then it expects a "Lifeline online" message within 250 milliseconds. If this does not happen, a timer expires, resulting in executing a certain function where the developer can place a breakpoint. These timers do not have any influence on the behaviour of the car and should only be used to detect the absence of certain expected CAN messages.

The task "Measure Switches" measures the state of some pushbuttons and flipswitches 20 times per second. Two pushbuttons are used to activate or kill the lifeline. If it measures that both pushbuttons are pushed by the driver at the same time, it does nothing. If the electronics are OK, it sends a CAN message to activate the lifeline if the driver pushes the corresponding pushbutton. The three flipswitches are used to start/stop the fuel cell, to send a command to cool down, or to prime.

The tasks "Display Speed" and "Update Speed Display" are used to update the speed shown on the speed display, found between the two rows of boostcap LED's. The first task is used to calculate the actual speed to display to the driver, based on the measured speed of the four wheels. The other task then continuously updates and refreshes the first, second, or third digit of the LED display. Because the LED display hardware can only show one digit at a time, the "Update Speed Display" needs to cycle with a very high frequency.

The "Knight Rider" task is a task that turns on all the LED's in a specific sequence for a couple of seconds when the PCB is powered on. The "MC Diagnostics" task requests diagnostic data from the motor controllers over the CAN3 bus, processes it and sends it on via the CAN2 bus.

Task	Type	(average) frequency	Misc
Send Heartbeat	periodic	10 Hz	
Receive CAN	sporadic	-	
Measure Switches	periodic	20 Hz	
Display Speed	aperiodic	15 Hz	
Update Speed Display	periodic	500 Hz	
Knight Rider	startup	-	
MC Diagnostics	periodic	1 Hz	

Table 1: Tasks running on the dashboard controller

4.3.2 Telemetry node

The telemetry node is used to gather telemetry and send this over the air via the JenNet protocol by communicating with a chip via UART. A listing of the tasks that run on the telemetry node can be seen in Table 2.

The telemetry node send its 'heartbeat' over the CAN bus 10 times per second in the "Send Heartbeat" task.

The "Send Telemetry" task sends content of a buffer — prepared by the "Gather Telemetry" task — via UART to the chip that will transmit the data over air. It does this 10 times per second. The "Gather Telemetry" task just listens for certain messages that it receives over the CAN bus and adds this to the buffer.

Task	Type	(average) frequency	Misc
Send Heartbeat	periodic	10 Hz	
Send Telemetry	periodic	10 Hz	
Gather Telemetry	aperiodic	-	

Table 2: Tasks running on the telemetry node

4.3.3 Throttle node

The two throttle nodes measure the amount of throttle that the driver applies and send an appropriate analog signal to the motor controllers. They also measure the wheel speed of the front wheels and perform several functions relevant to the lifeline. A listing of the tasks that run on the two throttle nodes can be seen in Table 3.

Just like the dashboard controller, the throttle nodes send their 'heartbeat' over the CAN bus 10 times per second. The "Receive Heartbeats" task listens for heartbeats from the ECU's connected to the CAN2 bus. It expects to receive a heartbeat from each ECU within a certain time after the last received heartbeat from that same ECU. It kills the lifeline if a heartbeat is not received in time or not at all.

The "Lifeline Status" task — which runs only on throttle node 1 — measures whether the lifeline is online or offline and send a notification over the CAN bus when the status changes. The "Process ES" task activates or kills the lifeline if it receives a corresponding CAN message or if signaled by another task.

The "Throttle Control" task measures the throttle pot that is associated with the throttle node. The throttle nodes perform some safety checks, including comparing the measured values from the two throttle pots. If the measured values are not valid, the throttle nodes try to kill the lifeline. If the measured values pass the safety checks, only throttle node 1 then send the appropriate analog signal to the motor controllers.

In the "Wheel Speed" task, each throttle node regularly measures the speed of their associated front wheel and broadcast this information over the CAN bus.

Task	Type	(average) frequency	Misc
Receive Heartbeats	aperiodic	100 Hz	
Send Heartbeat	periodic	10 Hz	
Throttle Control	periodic	100 Hz	Controlling MC by TN1 only
Lifeline Status	periodic	10 Hz	TN1 only
Process ES	sporadic	-	
Wheel Speed	periodic	5 Hz	

Table 3: Tasks running on the throttle nodes

4.3.4 Wheel speed node

The two wheel speed nodes measure the speed of the rear wheels. The right rear wheel speed also turns on the brake light if it measures that a certain amount of brake pressure is being applied or if the driver is requesting regenerative braking. The right rear wheel speed is also responsible for send an analog signal to the motor controllers for regenerative braking. A listing of the tasks that run on the two wheel speed nodes can be seen in Table 4.

The wheel speed nodes send their ‘heartbeat’ over the CAN bus 10 times per second in the “Send Heartbeat” task. In the “Wheel Speed” task, each wheel speed node regularly measures the speed of their associated rear wheel and broadcast this information over the CAN bus.

The other three tasks are only performed by the right rear wheel speed node. In the “Brake Pressure” task, the right rear wheel speed node measures the brake pressure 100 times per second. The task can then send a signal to the “Brake Light” task if the pressure becomes higher or lower than the minimum brake pressure. The “Regen Braking” task listens for CAN messages sent by the dashboard controller indicating whether the driver wants to apply 0%, 50%, or 100% regenerative braking. Because the dashboard controller’s “Measure Switches” task has a frequency of 20 Hz, this task, in practice, also has a frequency of 20 Hz. When this condition changes, the task signals the “Brake Light” task, which can then turn on or off the brake light. The “Regen Braking” task also sends an appropriate analog signal that indicates the requested amount of regenerative braking to the motor controllers.

Task	Type	(average) frequency	Misc
Send Heartbeat	periodic	10 Hz	
Wheel Speed	periodic	5 Hz	
Brake Pressure	periodic	100 Hz	Right only
Regen Braking	aperiodic	20 Hz	Right only
Brake Light	sporadic	-	Right only

Table 4: Tasks running on the wheel speed nodes

4.3.5 DCDC node

The DCDC node performs some functions that control and monitor the DCDC hardware that can be found in the vehicle. A listing of the tasks that run on the DCDC node can be seen in Table 5.

The DCDC node send its ‘heartbeat’ over the CAN bus 10 times per second in the “Send Heartbeat” task. In the “Receive Heartbeat” task it listens on the CAN bus for the heartbeat of the fuel cell controller.

The “Receive Current” task listens for messages — sent by the fuel cell controller — containing a current on the CAN bus and converts it to an analog value on the DAC.

The “Send Values” task sends the current and voltage in and out values read from the DCDC converter over the CAN bus.

The "Enter Standby" task sets the Required Current on the DAC to the standby current if it does not receive the heartbeat of the fuel cell controller within a certain time.

Task	Type	(average) frequency	Misc
Send Heartbeat	periodic	10 Hz	
Receive Heartbeat	aperiodic	10 Hz	
Receive Current	aperiodic	4 Hz	
Send Values	periodic	10 Hz	
Enter Standby	sporadic	-	

Table 5: Tasks running on the DCDC node

4.3.6 Fuel cell controller

The fuel cell controller reads the stack voltage and controls the other nodes in the fuel cell. A listing of the tasks that run on the fuel cell controller node can be seen in Table 6.

The "Voltage Monitoring" continously reads out the voltage for every cell in the stack, and calculates the average of all cells.

Using the "CAN Receive" task this controller receives sensor values and other messages from nodes in the fuel cell, and commands from the dashboard.

The "Diagnostics" task checks if all sensor values are in range, and takes action if one of them is not. Depending on the sensor and how far it is out of range, this causes a shutdown of the fuel cell, an emergency shutdown of the fuel cell or a warning to the driver.

Real control of the fuel cell is done by the "Run Control" task. This task periodically sets the actuators in the fuel cell, and keeps the power from the fuel cell at the level needed to keep the boostcaps as full as possible.

The "Statemachine" task controls the state machine.

A blink LED on the controller flashes with a frequency depending on the current state, controlled by the "Blinky" task.

The command shell used for some diagnostic information and debugging runs in the "Command" task.

Task	Type	(average) frequency	Misc
Send Heartbeat	periodic	10 Hz	
Receive Heartbeats	aperiodic	60 Hz	
Voltage Monitoring	periodic	4 Hz	
CAN Receive	aperiodic	150 Hz	
Diagnostics	periodic	10 Hz	
Run Control	periodic	4 Hz	
Statemachine	sporadic	-	
Blinky	periodic	8/2/1 Hz	
Command	sporadic	-	

Table 6: Tasks running on the fuel cell controller

4.3.7 Hydrogen tank node

The hydrogen tank node monitors the hydrogen tank and calculates an approximate mass flow to detect leaks upstream of the fuel cell. A listing of the tasks that run on this node can be seen in Table 7.

The hydrogen tank node send its ‘heartbeat’ over the CAN bus 10 times per second in the “Send Heartbeat” task. In the “Receive Heartbeat” task it listens on the CAN bus for the heartbeat of the fuel cell controller.

The “Check Massflow” task measures the pressure and temperature of the tank and uses a one-minute history to calculate the average mass flow over the past minute. It sends each of these values over the CAN bus and sends the “kill lifeline” command to close the tank if any of them is out of their safe range.

The “Coolant Temperature” task measures the temperature of the coolant at the inlet of the stack and sends this over the CAN bus.

Task	Type	(average) frequency	Misc
Send Heartbeat	periodic	10 Hz	
Receive Heartbeat	aperiodic	10 Hz	
Check Massflow	periodic	1 Hz	
Coolant Temperature	periodic	4 Hz	

Table 7: Tasks running on the hydrogen tank node

4.3.8 Hydrogen low pressure node

The hydrogen low pressure node measures the flow, pressure and temperature of the hydrogen in the low pressure system and controls the speed of the recirculation pump and the state of the inlet and purge solenoids. A listing of the tasks that run on this node can be seen in Table 8.

The hydrogen low pressure node send its ‘heartbeat’ over the CAN bus 10 times per second in the “Send Heartbeat” task. In the “Receive Heartbeat” task it listens on the CAN bus for the heartbeat of the fuel cell controller.

The “Purge Hydrogen” and “Set Flow” tasks control the purge valve and the recirculation pump in response to CAN messages.

The “Read Flow”, “Read Pressure” and “Read Temperature” tasks read the mass flow into the low pressure system and the pressure and temperature inside it and send these values on the CAN bus.

The “Process Stateswitch” and “Receive Stateswitch” tasks control the statemachine.

Task	Type	(average) frequency	Misc
Send Heartbeat	periodic	10 Hz	
Receive Heartbeat	aperiodic	10 Hz	
Purge Hydrogen	sporadic	-	
Set Flow	sporadic	-	
Read Flow	periodic	4 Hz	
Read Pressure	periodic	4 Hz	
Read Temperature	periodic	4 Hz	
Process Stateswitch	sporadic	-	
Receive Stateswitch	sporadic	-	

Table 8: Tasks running on the hydrogen low pressure node

4.3.9 Air node

The air node controls the airflow, measures its temperature and controls the pressure of the low pressure hydrogen system based on the air pressure. A listing of the tasks that run on this node can be seen in Table 9.

The air node send its 'heartbeat' over the CAN bus 10 times per second in the "Send Heartbeat" task. In the "Receive Heartbeat" task it listens on the CAN bus for the heartbeat of the fuel cell controller.

The "Pressure Regulator" task measures the air pressure and updates the hydrogen pressure setpoint to be equal to the air pressure plus 20 kPa and sends these values on the CAN bus.

The "Flow Setpoint" task controls the air mass flow setpoint based on CAN messages from the Fuel Cell Controller.

The "Air Flow" task measures the air mass flow and controls the air compressor with this value and the setpoint using PID. The mass flow is also sent over the CAN bus.

The "Air Temperature" task measures the air temperature at the stack inlet and sends this on the CAN bus.

The "Process Stateswitch" task controls the statemachine.

Task	Type	(average) frequency	Misc
Send Heartbeat	periodic	10 Hz	
Receive Heartbeat	aperiodic	10 Hz	
Pressure Regulator	periodic	5 Hz	
Flow Setpoint	sporadic	-	
Air Flow	periodic	100 Hz	
Air Temperature	periodic	4 Hz	
Process Stateswitch	sporadic	-	

Table 9: Tasks running on the air node

4.3.10 Cool node

The "Coolant Temperature" and "Massflow Coolant" tasks reads the coolant temperature at the outlet of the stack and the coolant mass flow and sends them via the CAN bus. A listing of the tasks that run on the cool node can be seen in Table 10.

This temperature is also used by the "Cool" task to control the fan on the radiator via the DAC. In running mode, the fan is turned off until the coolant temperature at the outlet of the stack is below 55°C, and fully turned on when the temperature is above 65 °C. Between these temperatures, the fan speed is increased linearly. When the node is in Cooldown mode, the fan is turned on as long as needed to get the temperature below 30 °C.

The "Process Stateswitch" and "Receive Stateswitch" tasks control the statemachine by commands sent by the Fuel Cell controller over CAN.

Task	Type	(average) frequency	Misc
Send Heartbeat	periodic	10 Hz	
Receive Heartbeat	aperiodic	10 Hz	
Coolant Temperature	periodic	4 Hz	
Massflow Coolant	periodic	4 Hz	
Cool	periodic	1 Hz	
Process Stateswitch	sporadic	-	
Receive Stateswitch	sporadic	-	

Table 10: Tasks running on the cool node

4.4 CAN protocol

All nodes in the vehicle communicate with each other using several CAN (Controller Area Network) buses. The CAN protocol defines messages of up to 8 bytes long, with an 11 bit message ID. The ID also determines the priority of a message: a lower ID means a higher priority. A listing of all the CAN messages can be seen in Table 11.

We decided to use the following conventions:

- All CAN ID's are unique over all CAN buses found in the system. In other words, the ID's are not related in any way with the bus from which they originate.
- Every message category has one or more blocks of 16 IDs that can be used for messages within this category.
- Every message type (sensor value etc) has its own ID in one of these blocks.

The CAN ID of the HLS-440 hydrogen sensor is set fixed to 0x640; this means that all messages with a lower priority must have higher ID's. Only the message that is used to kill the lifeline and the heartbeats have a higher priority than these sensor values. Throttle and wheel speed values have the next highest priority, because the software that sends and receives these kind of messages must have a very short response time. After this, sensor and command messages from the fuel cell follow. The lowest priority goes to the motor controller diagnostic values, because these values are purely diagnostic and are used for logging only.

ID	Type	Description	Value
0x500	Command	Kill lifeline	-
0x584	Heartbeat	Throttle node 1	-
0x585	Heartbeat	Throttle node 2	-
0x590	Heartbeat	Dashboard controller	-
0x591	Heartbeat	Telemetry node	-
0x592	Heartbeat	Left rear wheelspeed node	-
0x593	Heartbeat	Right rear wheelspeed node	-
0x594	Heartbeat	DCDC node	-
0x5c0	Heartbeat	Fuel cell controller	-
0x5c1	Heartbeat	FC Air node	-
0x5c2	Heartbeat	FC Tank node	-
0x5c3	Heartbeat	FC H2 low pressure node	-
0x5c4	Heartbeat	FC Cool node	-
0x640	Sensor	HLS-440 Hydrogen Sensor	See API
0x650	Command	Activate lifeline	-
0x661	Status	Successful Emergency shutdown	1 byte char (reason)
0x662	Status	Successful Activate lifeline	-

0x663	Status	Lifeline offline	-
0x664	Status	Lifeline online	-
0x665	Status	Got all heartbeats	-
0x6a1	Sensor	Throttle pot 1	2 byte unsigned short
0x6a2	Sensor	Throttle pot 2	2 byte unsigned short
0x6d1	Sensor	Left front wheel speed	4 byte unsigned int (RPM)
0x6d2	Sensor	Right front wheel speed	4 byte unsigned int (RPM)
0x6d3	Sensor	Left rear wheel speed	4 byte unsigned int (RPM)
0x6d4	Sensor	Right rear wheel speed	4 byte unsigned int (RPM)
0x701	Command	Regenerative braking 0%	-
0x702	Command	Regenerative braking 50%	-
0x703	Command	Regenerative braking 100%	-
0x731	Setpoint	Air mass flux setpoint	8 byte double (SL/min)
0x732	Setpoint	Mass flux h2 setpoint	4 byte unsigned int (I (5V/1024))
0x733	Setpoint	Hydrogen pressure setpoint	-
0x737	Command	FC Purge	4 byte unsigned int (ms)
0x741	Setpoint	Current required	8 byte double (A)
0x747	Command	FC H2 low pressure mode	4 byte unsigned int (mode)
0x748	Command	FC Air mode	4 byte unsigned int (mode)
0x749	Command	FC Cool mode	4 byte unsigned int (mode)
0x74b	Command	FC Start	-
0x74c	Command	FC Stop	-
0x74d	Command	FC Standby	-
0x74e	Command	FC Prime	-
0x74f	Command	FC Cooldown	-
0x751	Status	FC started	-
0x752	Status	FC stopped	-
0x753	Status	FC in standby	-
0x761	Status	Air/H2 pressure diff too high	-
0x762	Status	Air pressure sensor inoperative	-
0x770	Sensor	H2 pressure	8 byte double (bar)
0x771	Sensor	H2 temperature	8 byte double (°C)
0x772	Sensor	H2 mass flux	8 byte double (SL/min)
0x773	Sensor	Air mass flux	8 byte double (SL/min)
0x774	Sensor	Air pressure	8 byte double (bar)
0x775	Sensor	Air temperature	8 byte double (°C)
0x77a	Sensor	Coolant in temperature	8 byte double (°C)
0x77b	Sensor	Coolant out temperature	8 byte double (°C)
0x77c	Sensor	Coolant mass flux	8 byte double (SL/min)
0x780	Sensor	H2 tank pressure	8 byte double (bar)
0x781	Sensor	H2 tank temperature	8 byte double (°C)
0x784	Sensor	DCDC current in	8 byte double (A)
0x785	Sensor	DCDC voltage in	8 byte double (V)
0x786	Sensor	DCDC current out	8 byte double (A)
0x787	Sensor	DCDC voltage out	8 byte double (V)
0x788	Logging	Average cell voltage	8 byte double (V)
0x789	Logging	Cell voltage factor	8 byte double (-)
0x7d0	Other	Motorcontroller left receive	-
0x7d1	Other	Motorcontroller left send	-
0x7d2	Other	Motorcontroller right receive	-
0x7d3	Other	Motorcontroller right send	-
0x7e0	Sensor	MC Left PWM	1 byte char
0x7e1	Sensor	MC Right PWM	1 byte char
0x7e2	Sensor	MC Left Rotation	1 byte char
0x7e3	Sensor	MC Right Rotation	1 byte char

0x7e4	Sensor	MC Left Motor temperature	1 byte char
0x7e5	Sensor	MC Right Motor temperature	1 byte char
0x7e6	Sensor	MC Left Controller temperature	1 byte char
0x7e7	Sensor	MC Right Controller temperature	1 byte char
0x7e8	Sensor	MC Left FET high-side temperature	1 byte char
0x7e9	Sensor	MC Right FET high-side temperature	1 byte char
0x7ea	Sensor	MC Left FET low-side temperature	1 byte char
0x7eb	Sensor	MC Right FET low-side temperature	1 byte char
0x7ec	Sensor	MC Left error code	2 byte unsigned short
0x7ed	Sensor	MC Right error code	2 byte unsigned short
0x7ee	Sensor	MC Left current percentage	1 byte char
0x7ef	Sensor	MC Right current percentage	1 byte char

Table 11: List of all the CAN messages

5 Software quality and testing

In this section we explain how we assure good software quality, and how we test the software to be sure everything works as required. In Section 5.1 we describe our what kind of tests we will perform. In Section 5.2 the system used for software documentation. In Section 5.3 we discuss MISRA compliance. Section 5.4 sets our software metrics goals.

5.1 Testing

The kind of tests we will execute are the following:

- Manual unit testing of drivers and ECU functionality on development boards and final PCB hardware using a digital oscilloscope, multimeters, and a special device that can monitor CAN messages that get broadcasted over a CAN bus.
- Checking for possible stack overflows using the IAR Embedded Workbench IDE
- Using Rate Monotonic Analysis to determine whether a set of tasks on an ECU that might result in a processor utilization that is too high
- Testing integration of the various ECU's by taking the vehicle for several test runs

5.2 Documentation

Good documentation for software is essential for software maintainability. If future programmers want to fix bugs, make changes or re-use code in other project, they will need an explanation of how the software we wrote works, and why it was implemented this way.

Our software documentation consists of two important parts:

- This report functions as a source of 'global' documentation. The design of the full systems can be found here, including argumentation why certain decisions were made.
- The specification of the driver API's can be found in the driver header files. All functions are described here with an explanation of what their parameters and return value is.

5.3 MISRA C

MISRA C is used as required by the project description and to detect code that could be the source of possible bugs. However, Segger embOS — the RTOS we used — is known to be not MISRA C compliant and did lead to many warnings that could only be removed by changing code in the the RTOS itself; something that was obviously impossible. Our research showed that some rules of the coding guidelines can be interpreted in multiple ways and code changes to remove warnings may introduce new bugs. However, we did follow more of the rules that deal less with the syntax and more with the semantics. For example, we did not use dynamic memory allocation or dangerous pointer arithmetic. Following these rules proved to be more useful than, for example, adding an 'u' character after every unsigned integer literal.

5.4 Software metrics

The obvious things to look at when quantifying software is lines of code, modules, functions and combinations like average lines of code per function. Other things that can be measured are parameter counts, maximum nesting depth and cyclomatic complexity.

With regard to embedded software, things like the maximum stack size of a task are relevant.

6 Implementation

In this section we give some details about the actual implementation we did, according to the designs as found in Section 4. Sections 6.1 and 6.2 explain which software was actually ported and which was written new. Section 6.3 explains the method used to determine which priority tasks should have. Section 6.4 explains which software that was not part of the original assignment.

6.1 Porting old software

Our initial expectation was that a lot of software could be ported from the Forze III applications. However, already during the design phase it became clear that only a few small parts could be re-used. Because the new boards use a new family of microcontrollers, the directly hardware-related parts were outdated. The API's of the old drivers were not as good as we wanted them, if present at all.

Most of the application software was not reusable too. The fuel cell control application was very simple in the Forze III, and only had to send some CAN commands and check some CAN diagnostic messages. The dashboard application was running on a PDA, where the dashboard controller in the Forze IV was directly connected to LED's, LED displays and hardware buttons.

Eventually, in only two parts of the software substantial amounts of code could be re-used:

- The CAN driver uses the old implementation to send and receive CAN messages. The API however, was fully redesigned and rewritten.
- The UART driver was almost completely compatible with our requirements, so it was reused nearly unchanged. We also reused the command shell structure for testing and debugging.

6.2 Writing new software

As stated in Section 6.1, porting of software was hard or impossible for most of the software. That means a lot of new software had to be written from scratch. This gave us the advantage of being free in chosing how we wanted to design and implement these parts, but at the same time meant we had more work than we expected.

6.3 Rate Monotonic Analysis

Rate Monotonic Analysis was used to assign fixed priorities in such a way that the set of tasks on an ECU would be schedulable — as described by Liu and Layland [3]. In other words, that the processor utilization of the set of tasks is lower than the least upper bound — which depends on how many tasks a set contains — so that all real-time tasks will always meet their deadlines.

The analysis makes a number of assumptions and does not handle aperiodic tasks, sporadic tasks, task synchronization, I/O blocking, priority inversion, non-zero task switching times, interrupts, and non-preemptable code segments. Even though the basic analysis has some drawbacks, we used RMA as a guideline to determine which priorities to assign. We applied RMA with the following two additional considerations:

1. For two tasks that have the same period, the task that is considered to be more 'critical' is given a higher priority.
2. As suggested by the article [4] and Lehoczky et al. [2], all the tasks are given harmonic frequencies if possible so that the least upper bound for the set of tasks is 100%. If harmonic frequencies are not

possible, the average bound for a randomly generated set of tasks is shown to be 88% by Lehoczky et al. [2], which improves our confidence that an ECU will be able to schedule all real-time tasks.

6.4 Written software not part of original project description

The following parts of the final software were not part of the original project assignment or the design we made:

- **SPI driver**. Because the hardware designs were not finished in time, we did not know we needed SPI for the DAC, thermocouple and LED driver controllers. When it became clear all these controllers needed this protocol, we decided the best way to handle this was to write a separate SPI driver and let the other drivers use this driver.
- **LED driver**. Like the SPI driver, we had not enough details about the hardware to know the need for a LED driver. However, the SPI driver made the LED driver fairly simple to implement.
- **Fuel cell control software**. When starting the project, we knew this software had to be written. However, this software was not part of the project assignment and we were told that another team would write this software, therefore we did not take the process of designing and writing this software into account for the project planning. More details about the reason we designed and wrote this software can be found in Section 7.3.

7 Development process

In this section we describe the development process that has taken place during the project. We describe the process in Section 7.1. Section 7.2 gives an overview of problems encountered during this process. A separate Section 7.3 is written about the fuel cell control software. Section 7.4 is about the FSUK event we participated in with the car.

7.1 Process description

The first days of the project we were busy setting up a workplace, getting to know the other team members we had to work together with and installing and configuring the development environment. After we were finished with this we started working on the orientation report and plan of approach.

The next step was designing and implementing the various drivers. At the same time, more and more hardware prototypes became available, so we could start testing these boards and try out the drivers on the target hardware and help the hardware designers to fix bugs in their boards.

When the drivers were finished, we started with the design and implementation of the application software that had to be written for the dashboard, throttle nodes, and telemetry node. About a week before the scheduled end of the project, it became clear that no other team members would be available for the development of the fuel cell control software (see Section 7.3 for more information). To get the car working in time we started working on this parts of the software too.

This meant a lot of work had to be done in the last week of the project. We worked every possible hour and tried to finish the software in time. When we had to leave to the UK we did not have a working car, and the team could not get it to work during the event (see also Section 7.4). After the event we continued working to get the car ready to drive. And with good results: at August 16th we did a successful FIA world record attemt for hydrogen vehicles at the Beatrixlaan in The Hague!

The final step was writing this report. Although we planned to work on the report during the development process, the large amount of work took all the time we had, so the report had to be written in the last weeks before the presentation could be given.

7.2 Encountered problems

During the development of the software, we encountered some problems:

- The specifications of the various systems in the car that had to be controlled were not as detailed as needed, so we needed a lot of time to speak to the people who designed and built these parts, and had to wait for the answers to our questions.
- Our development hardware had some bugs, so we had to figure out wat was wrong with them and even grab the soldering iron to fix these bugs.
- Because the target hardware was fully newly designed, we had to test whether the hardware really did what it needed to do. Several problems were found in the designs of the PCB's of the hardware and it costed a lot of time waiting for new revisions of the boards. This also meant we could not test our software on the target hardware for a long time, so debugging of the software on the target hardware started very late.

7.3 Fuel cell software

Writing the fuel cell controller software was not a part of the original assignment as described by Forze. Another team promised to write this software, but the board could not contact them anymore when the software really had to be written. Because the team really wanted to finish the vehicle in time, we decided to throw a little extra effort and started designing and writing the fuel cell software.

However, we started with this software only two weeks before the FSUK competition would start. We worked really hard to finish the software in time, but unfortunately we did not finish it in time. After this, the code had become a real mess, so we decided to rewrite the fuel cell control software after FSUK. With the extra knowledge we had developed during these weeks, this rewrite was done surprisingly fast: within two days of work the software was rewritten and the fuel cell was running for some minutes for the first time.

A big issue during the development of the fuel cell controller was domain knowledge: knowledge about the working of a fuel cell appeared to be necessary to write the software to control it. We solved this by working closely together with the fuel cell engineer from the team, and a lot of work and study to expand our own knowledge.

7.4 Formula Student UK event

Our main activity at the competition at Silverstone Circuit was getting the fuel cell software to work. Unfortunately, some mistakes were made when connecting components, so hardware was damaged and we lost valuable time fixing hardware and waiting for hardware to be fixed. We could not get the car working to compete in the dynamic events.

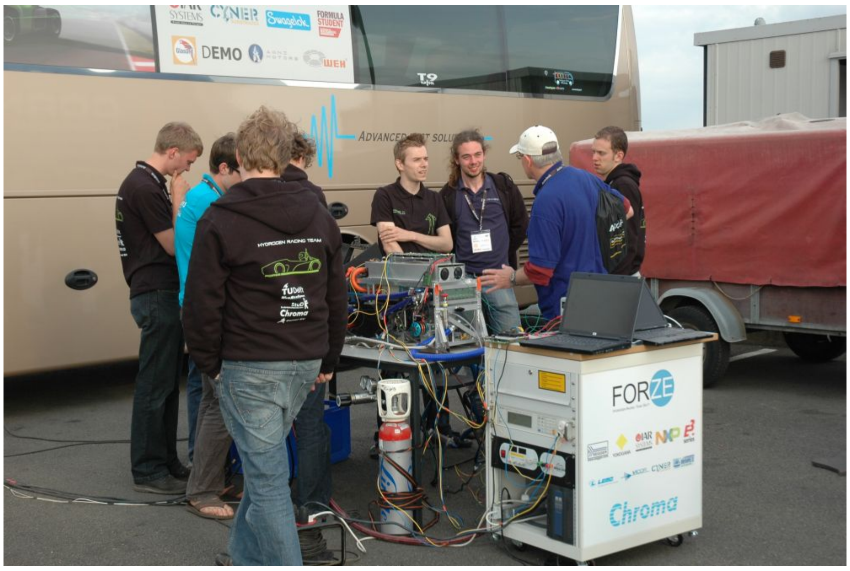
Figure 3: Fuel cell testing at Formula Student UK Competition.

We also had some problems calibrating the two throttle nodes and comparing the values, so after the competition we decided our safety measures are good enough that it allowed us to remove one of the nodes and control the throttle with only one node.

Although we could not show the car driving, everyone was very enthousiastic about the first Formula Student hydrogen fuel cell car. Even the scrutineers and judges were impressed by the quality of the car we built and help was offered from all sides as everyone really wanted to see our car driving.

8 Results

In this section we describe the results of this project. We give an overview of which requirements are met and which not in Section 8.1, and give the results of the execution of the test plan in Section 8.2.

8.1 Implementation of requirements and specifications

There is no HLS-440 driver per se, as much as there is a structure describing what each bit of the CAN messages it sends means. As this 'driver' is used in only two places and the lack of functions requires the client to add only three simple lines of code, this not a terrible trade-off.

Logging to SD card has only been partially implemented. The filesystem driver works and a dummy logger is functional on the development boards, but as there was no SD card connector present on target hardware, implementation of the actual logging tasks was delayed in favor of more immediate requirements.

The double throttle node system was eventually removed. There were problems calibrating the nodes and eventually the hardware was required for a different node, as we ran out of spare PCB's.

8.1.1 Requirements

Of the two quality requirements found in Section 2.4.1, most of the API's of the drivers are resuable on the Cortex-M0 and Cortex-M3 microcontrollers. ECU's have been implemented in such a way that unexpected CAN messages are simply ignored or an error will be raised during debugging. However, most code does not check that CAN messages have a payload with the expected number of bytes. The software simply copies the expected number of bytes from the buffer to a variable. This should only result in incorrect values, rather than a crash. Fortunately, the design of the CAN protocol describes the length of the data of each CAN message.

All items of "Must have" of the functional requirements found in Section 2.4.2 have been achieved. Of the "Should have" only the items concerning the telemetry node has been achieved. The items of "Could have" and "Want have" have not been achieved due to having been assigned a lower priority because the whole team needed to get the vehicle ready to drive on time for FSUK.

All platform requirements in Section 2.4.3 and the items of "Must have" of the safety requirements in Section 2.4.4 have been achieved. Most of the deadlines — as specified in the process requirements in Section 2.4.5 — have not been achieved because some drivers required more time for testing, or required the availability of the final PCB's of the general-purpose nodes hardware. Perhaps these deadlines were too tight to begin with.

8.1.2 Specifications

Most of the items of the specifications of the drivers in Section 3.1 have been implemented. The DAC driver does not provide an API to configure the voltage level for the 'enabled' and 'disabled' states. (This is a 'MAY', however) The SPI driver does not allow one to set the frequency of the transmission; this is currently hardcoded to 4 MHz. The HLS-440 driver does not provide an API to get some data from the message sent by the HLS-440 device. It does, however, provide a data structure that makes it easy enough to extract the data.

The items of the specifications of the ECU's in Section 3.2 have been implemented. The item that specifies that the dashboard controller should request diagnostics from the motor controllers was implemented after FSUK.

8.2 Execution of tests

All individual PCBs and drivers were tested extensively by hand and several subsystems — the throttle nodes and the fuel cell — were tested as a whole, but full integration testing was not possible until the wiring harness and final assembly were finished, which did not happen until FSUK.

We could not write and execute unit test cases — as is done in typical non-embedded software development — for the drivers because the drivers interface with the hardware and sometimes because the hardware could not be found on the development boards, but only on the specific PCB's — produced by the electrical department. One example is the DAC driver for the Cortex-M0, which requires a specific chip that can only be found on the PCB's of the general-purpose nodes.

The testing we did do was a lot of checking with the oscilloscope that analog signals on certain pins were as expected. Furthermore, we used a special device that could read broadcasted messages on the CAN bus and display the ID, value and frequency of each message. This device was very useful to verify the functionality of the CAN driver and that the several ECU's sent the correct messages with the correct frequencies.

In the beginning, the hardware problems of the development boards and a missing robust CAN bus, sometimes withholded us from proper testing. However, later these problems were fixed. Another problem was that the final hardware only became available very late during the project or sometimes was not even available; the dashboard was finished by other team members very late and there were a lot of electrical problems with many boards, which meant that we had to wait until the electrical department had fixed these or produced new boards, before we could continue testing certain drivers or ECU's.

8.2.1 Stack overflow testing

The IDE we used has support for the Segger embOS RTOS; it can show a list of tasks, their state, and their stack usage. The debugging build of the RTOS can monitor the stack usage and detect stack overflows, although not reliably. We hit some stack overflows during testing; we remedied a stack overflow by doubling the stack, monitoring the stack usage of the task that caused the stack overflow, and then gradually reducing the stack again a couple of times until there was a safe margin between the stack usage and the maximum available stack.

8.2.2 Rate Monotonic Analysis

RMA was mostly used to assign the fixed priorities of the real-time tasks, but was also used to see whether the microcontrollers would be able to handle the CPU load. Because we measured how long the microcontroller needed to execute a certain code segment — which uses system calls from the RTOS — the processor utilization time is only an estimate and not a guarantee. The processor utilization of the set of tasks appeared to be no problem; we had more trouble with the memory usage of the tasks on the Cortex-M0 because of the severe memory limitation of this specific microcontroller.

9 Conclusions and recommendations

In this final section, our conclusion about this project can be found. We evaluate what could have been done better in Section 9.1, look at the final results in Section 9.2 and what we have learned in the last months in Section 9.3. In 9.4 we give some recommendations for the team to build an even better car next year.

9.1 Evaluation of development proces

When looking at the complete development process, we can say it was far from optimal. The specifications available at the start were insufficient, and during the process we decided to write a lot more software than we had initially planned. Of course, this had consequences for the amount of time available for testing: this was very much limited. Also, the pressure on the programmers in the final phase was very high because of the deadlines of the roll-out and FSUK, which resulted in bungled work and quite complex code.

9.2 Evaluation of results

When looking at the final results of the project, we can be satisfied. The main goal of getting the Forze IV ready to drive has been achieved. The quality of the software, according to the Software Improvement Group, is average. This means future team members have a foundation to build on, but also that there is a lot to improve in the future.

9.3 Acquired experience

Before we started working on this project, none of us had substantial experience with embedded programming using a RTOS, other than one course on Embedded Programming in the second year. After the project we know what it is to build a fairly complex embedded system, distributed on about 10 different boards spread across a race car.

Because the hardware was developed in-house and we also had to write the software to test it working close together with electrical engineering students, we learned a lot about measuring, testing, and debugging hardware and software that works closely with that hardware.

Another interesting point is the working of a hydrogen fuel cell. To be able to design and write the control software for this system requires a lot of domain knowledge about hydrogen fuel cells. We acquired this knowledge by interviewing team members who had built the fuel cell.

We also aquired experience on non-technical aspects: being a member of a student race team is a learning experience. We learned how to organize and support events where the car had to be driven or to be showed to interested visitors. We learned to keep an eye on safety during these events, while working with highly flammable hydrogen gas.

9.4 Recommendations

We have the following recommendations for the team to make the software development process for next year's vehicle more easy than this year:

- The hardware design should be more simple. For example, 5 separate boards to control just the fuel cell makes software development and debugging a lot harder than just one or two larger boards.

- The hardware designs should be available more early, before the software design starts, to make a realistic estimate of how much work has to be done to develop software to control it.
- The hardware should be more robust and fool-proof. Too much hardware was damaged by just small mistakes during testing.
- It would be a good idea to have some advice from more experienced embedded programmers during the design and development phases. Team members who have worked on previous cars could be asked to have a few meetings where the current programmers can show their designs and implementations and receive feedback.
- The software design and development should start more early. Because the team works with volunteers, it is possible people quit with few ways to force them to finish their work. When more time is available, other programmers can be found to complete their jobs.

References

[1] Hydrogen Racing Team Delft. Risk assessment Greenchoice Forze. 2011.

[2] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *Real Time Systems Symposium, 1989., Proceedings.*, pages 166–171. IEEE, 1989.

[3] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973.

[4] D. Stewart and M. Barr. Rate monotonic scheduling. *Embedded Systems Programming*, page 79, 2002.

A Project description

Assignment description

Software Electronics Forze 4
2010-2011
Bachelor Project TI
Alexander van Gessel
Mark Provo Kluit

Introduction:

The hydrogen racing team delft is developing and building its 4th generation vehicle this year. It is intended to compete in the Formula Student UK class 1A competition for alternative fuel formula cars.

The Forze 4 is a hydrogen fuel cell powered vehicle build by TU Delft students from various faculties. More than 50 students are involved in this project from September 2010 until august 2011.

During previous seasons a modular electronics system has been developed and used with great success. This way the separate systems can be spread around the car to each perform its specific task. Many small boards also makes the system easier to debug, cheaper to replace and easier to adapt or expand. To achieve al the tasks in the Forze 4 vehicle 4 types of boards are used.

- Dashboard controller: reads input from driver, gives feedback to driver, does data logging
- Fuel cell controller: controls all fuel cell related tasks
- Sensor nodes: tiny board, reads out sensors, put data on CAN bus
- General purpose nodes: same as sensor nodes but also has some outputs to drive equipment
- Specific purpose nodes: telemetry node, inertial measurement node, throttle node,…

Last year 2 microcontrollers were used:

- ARM7 LPC2119 (small boards)
- ARM7 LPC2388 (big controller boards)

These hardware designs are updated to fit our needs and new controllers were selected:

- Cortex M3 LPC1768 microcontrollers for the dashboard and fuel cell controllers
- Cortex M0 LPC11C14 microcontrollers for the nodes

Because of this hardware update new software is needed, last year's software can be used as a guideline.

Assignment:

The assignment parts below should be completed in the order they are listed below and are essential for the good working of the vehicle.

Write a plan of approach for the project and a requirements plan clearly stating the different requirements and the consequences if a requirement is not met (in time).

Configure the operating system (delivered with support files for our controllers) to run on our hardware.

- Segger embOS (M3 and M0)

Configure, write drivers for

- UART (M3 and M0)
- CANbus (M3 and M0)
- ADC (M3 and M0)
- DAC, PWM (M3 and M0)

Configure SD card drivers and logging system

- Segger emFile (Only M3)

Telemetry system

- Write an application for the telemetry node pulling messages of the CAN bus and putting it on the UART connection to the wireless module.

The following assignment parts are on application level and also essential for the good working of the vehicle. This should be very easy if the previous assignments are completed successfully.

- Write the dashboard controller application:
  - Read inputs from switches and put this information on the CAN bus
  - Display essential information on the dashboard (status LEDs etc.)
- Write the throttle node application:
  - Accept emergency stop commands from other systems and cut car lifeline
  - Read out throttle position sensor and check if it is within limits
  - Compare throttle position sensor value with secondary sensor

If time is left the assignment can be expanded to for example:

- Configuration tool for telemetry node and sensor nodes
- Supporting the fuel cell application team
- Enabling two way communications on the telemetry system to change the vehicle settings from the paddock to save time during test sessions
- Enabling wireless download of the log file over the telemetry system

All software should comply with the MISRA guidelines wherever possible.

Extra info:

- We prefer the report to be written in English so future (international) team members may use it as a reference.
- The assignment can be changed by Greenchoice-Forze if they deem this necessary for the vehicle to be ready on time.
- The source code or parts of it is not to be published in the report or any other document without the written permission of the board of Stichting Formula Zero Team Delft.

Best regards,

[illegible]

Pieter Danneels

Chief Electronics Department
Greenchoice Forze
www.greenchoice-forze.com
p.danneels@greenchoice-forze.com
+31 (0)6 26567339

B Orientation report

Orientation report

IN3405 Bachelor project

Usage of coding standards for embedded programming in automotive applications

Authors:	*Student number:*	*Mentor Greenchoice Forze:*
Alexander van Gessel	1287974	P. Danneels
Mark Provo Kluit	1263099	
Jan Jaap Treurniet	1308351	*Mentor TU:*
		H.G. Gross
		Coordinator:
		B.R. Sodoyer

Contents

1 Introduction

This orientation report was written at the start of our bachelors project on the development of embedded software at Greenchoice Forze. The assignment of this project was to develop embedded software for a Formula Student 1A class hydrogen race car for the Formula Student UK 2011 competition. One of the requirements given was compliance to the MISRA C coding guidelines where possible.

Some research about the contents of these guidelines learned that writing compliant code would mean an extra effort, so the authors started wondering if these guidelines would really improve the quality of the product, if there would exist any other coding standards that could be used, what the differences are between these coding standards and if there are any results available of using them.

In this report we will give an overview of existing guidelines and standards for embedded software development, based on research in the existing literature. In Section 2 we explain why one would use a coding standard, and how coding standards can be used. In Section 3 we describe the MISRA C standard, discuss it's drawbacks and present any results from research to the effectiveness of these guidelines. In Section 4 we do the same for AUTOSAR. In Section 5 we briefly discuss some other coding guidelines and standards we found. Our conclusion can be found in Section 6.

2 About coding standards and guidelines

The concept that certain programming constructs can be detrimental to code quality is nothing new, with the 'goto' statement being questioned as early as the 1960s [5]. Embedded software is of particular concern, as it tends to have higher reliability requirements, and is usually written in C, which is notorious for having a large amount of unsafe constructs [9, 17].

This means it makes sense to avoid these constructs in an embedded software project, which can be done in different ways. A body of rules restricting the use of certain features of a programming language — and specifying the ways other features should be used — is known as a *coding standard*. Another option is to formally describe a *subset* of the C language, which lacks most of the constructs that one deems harmful [8]. Such a subset can then be extended with features that make it easier to write safe programs than it would be in C. The resulting language can be called a *safe dialect* [12].

Separately from coding standards, one can use *formal methods* for the specification and design of the system. These ensure that, if the specifications are properly implemented, (part of) the system is provably correct — there also exist methods to prove the implementation, but these are very time-consuming. The problem with *formal methods* is that using them takes a great deal of time and that, while they are widely believed to improve code quality, the empirical proof for this is lacking [16]. Another problem is that a system that is completely correct, is not necessarily a safe system. If the specifications are wrong, catastrophic failure can still occur, as the Ariane 5 flight 501 shows [18].

It is commonly assumed that writing safe software is more expensive than normal software, but there are known cases where the safe software had lower development costs. It has been suggested that the effort expended designing the system can reduce complexity, and a simpler system tends to be safer, more reliable and easier to implement [18].

3 MISRA C

In this section we we will discuss the MISRA C guidelines. In Paragraph 3.1 we will describe the MISRA C guidelines, and in Paragraph 3.2 we give an overview of it's drawbacks. In Paragraph 3.3 we discuss MISRA compliance, and Paragraph 3.4 describes results of research to the effectiveness of compliance.

3.1 What is MISRA C?

The full description of MISRA C is *Guidelines for the use of the C language in critical systems*. MISRA C is a development standard for the C programming language. The standard has been developed by MISRA (the Motor Industry Software Reliability Association), originally for use in the automotive industry. Nowadays MISRA C is also used by many developers in other industries that have a safety-critical component [7].

The current version, MISRA-C:2004 consists of 141 rules. 121 of these rules are marked as *required* and 20 are *advisory*. The rules are divided in 21 groups, for example: [19]

- Language extensions
- Documentation
- Control flow
- Runtime failures

3.2 Drawbacks

There is a lot of literature containing drawbacks of the MISRA C guidelines. We will mention the most important below.

MISRA C contains rules that have several possible interpretations [15]. This can cause programmers to choose the wrong interpretation, and not implement the correct solution. This multi-interpretability also causes problems when checking compliance. We will discuss this in Paragraph 3.3.

Les Hatton [9] compares MISRA C:1998 and MISRA C:2004. He finds that the signal-to-noise-ratio (the number of MISRA violations that actually cause software faults) of MISRA C:2004 is, although higher than in MISRA C:1998 [7], still very low. Also, he states the fixing of MISRA violations is likely to lead to more faults because every code change has a non-zero probability of introducing new errors, a phenomenon noticed by E. Adams as early as in 1984 [1].

Figure 1 shows transgression rates for MISRA C rules in different software projects. For some rules, transgression rates are very high, while it is unlikely that these projects will contain this much faults. This illustrates the high signal-to-noise ratio as described above.

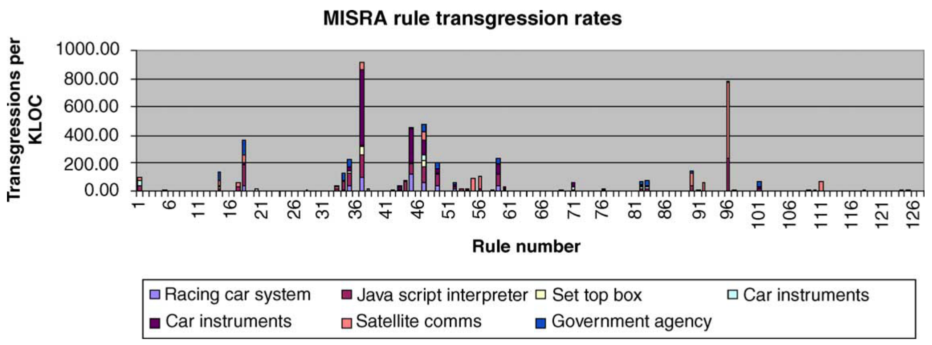

Figure 1: Transgression rates for MISRA C:1998 rules [7].

3.3 Compliance testing

A formal MISRA certification process does not exist. Any company can claim MISRA compliance for their code. Compliance to most of the MISRA guidelines can be checked automatically using static code analysis, and several tools that exist that claim to check conformance to these guidelines.

A comparison of MISRA C testing tools by Parker et al. [15] show big differences between the tested tools. One of the causes for differences is the interpretation of MISRA rules by the tool vendors. This might cause problems in situations where a company tests its code using a tool and claims compliance based of the results, while another tool might still find violations.

3.4 Results of MISRA C compliance

Boogert and Moonen [3] performed an empirical study about the effectiveness of following MISRA rules on the TV on Mobile project, looking for the correlation between violation of MISRA rules and faults in software. They did this by analysis of the source code (including history) and problem reports for the code. This study had a remarkable result: only 18 out of the 72 rules investigated had a positive influence on the number of faults. For 25 rules they found no correlation between the rule and the number of faults, and 29 rules even had a negative influence on the number of faults. Of course, these results are valid only for a certain project, but they show it is important to be careful when selecting code guidelines, and not just blindly implement all rules.

4 AUTOSAR

In this section we will discuss the AUTOSAR architecture. In Paragraph 4.1 we will describe the AUTOSAR architecture, and in Paragraph 4.2 we give a brief overview of it's drawbacks. In Paragraph 4.3 we discuss AUTOSAR compliance and accreditation, and Paragraph 4.4 describes results of research to the effectiveness of compliance.

4.1 What is AUTOSAR?

AUTOSAR, standing for ***AUT**omotive **O**pen **S**ystem **AR**chitecture*, is an open and standardized architecture for automotive software [10]. It was developed by automotive companies such as BMW Group, Ford, GM, PSA, Toyota and Volkswagen. AUTOSAR tries to cope with the increasing functional scope by defining a software architecture which promotes modularity, scalability, and reusability [10] and by defining a methodology for a number of steps in the software design process.

AUTOSAR separates applications from the hardware-dependent system software via a RTE (Run Time Environment) (see Figure 3) and provides standardized interfaces for application software to communicate with each other. This increases modularity and reusability of application software in different ECU's (Electronic Control Units) [20]. The standard gives a layered software architecture for applications that should be used, and guidelines on how the components should be implemented, including a list of implementation rules for C code. Figure 2 shows an overview of AUTOSAR's separation of application software and AUTOSAR basic software. Figure 3 shows the layered architecture of the basic software components [4].

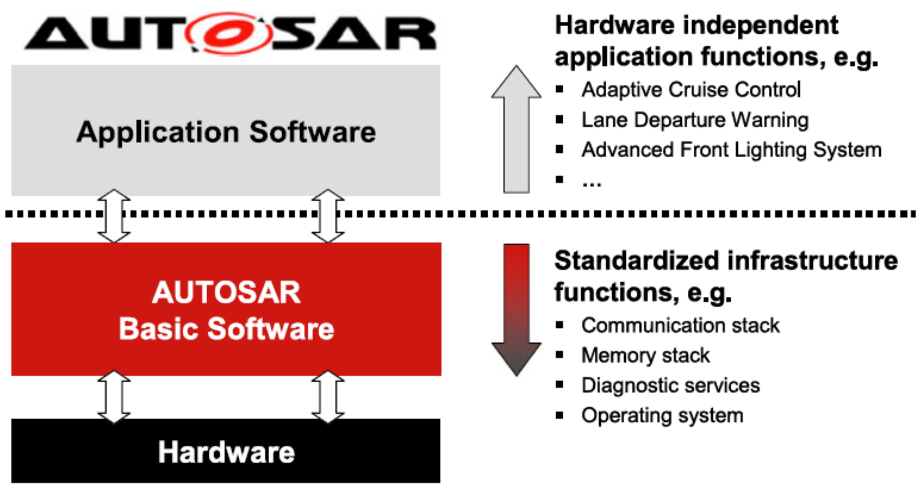

Figure 2: Separation of application software from infrastructure functions [4].

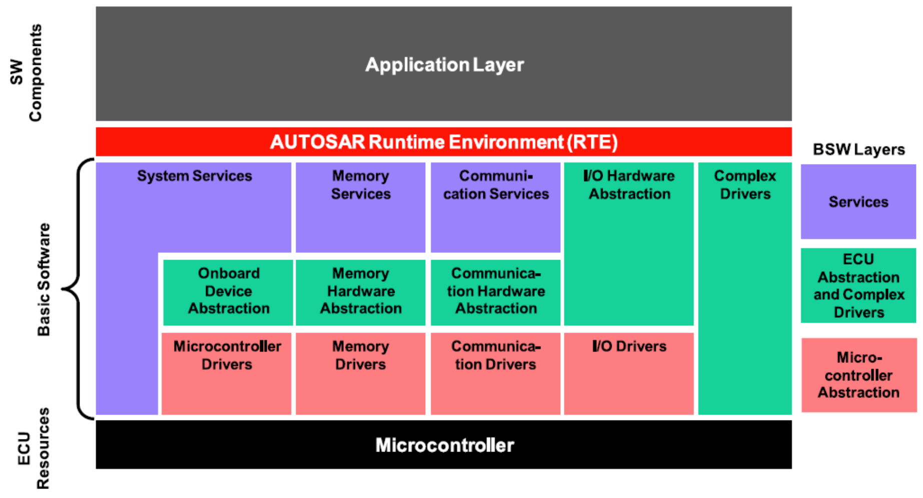

Figure 3: AUTOSAR's layered software architecture [4].

4.2 Drawbacks

One of the drawbacks of AUTOSAR is that it does not address timing problems, according to Espinoza et al. [6]. However, the TIMMO project — which stands for ***TIM**ing **MO**del* — attempts to address this issue by defining a timing framework which complements the AUTOSAR standard.

Espinoza et al. also claims that one of the identified deficiencies of AUTOSAR is the limited abstraction: "AUTOSAR covers only the lower two layers of the EAST-ADL[1] for the automotive domain developed in the ATESST project-layered architecture modeling proposal" [6].

4.3 Compliance and accreditation

Software can be tested for conformance against the AUTOSAR specifications by a CTA (Conformance Test Agency). The AUTOSAR organization provides conformance test specifications for CTAs (Conformance Test Agencies). AUTOSAR provides specifications and rules for CTAs and accreditation bodies for CTAs. Also, the possibility for self declaration of conformance exists for parts of the specifications where the certification process cannot be fulfilled because the conformance test process has not been finished, or where no CTA has been accredited yet.

4.4 Results

We have not found any information about the effectiveness of AUTOSAR so far with respect to the software architecture and methodology, although it has

[1]EAST-ADL is a modeling language for the architecture of electronic systems in automotives

been identified that AUTOSAR lacks a definition of a timing model. This may be attributed to the fact that AUTOSAR only exists since a couple of years. On a positive side note, AUTOSAR was created by large players in the automotive industry and is continuously being improved.

5 Other coding standards and language subsets

In this section we will briefly describe some other coding standards and language subsets we found during our research. We will not discuss them as much in depth as the MISRA C and AUTOSAR standards in the previous sections, but only give a short explanation and the reasons why we think they are not as much applicable for our project.

5.1 EC−−

EC−− is a subset of ISO C, constructed by Leslie Hatton [8]. The goal was to define a small number of rules to avoid faults, and to avoid any rules for which automated checking is impossible. The rules are based on fault occurrence rates found in other surveys. No effectiveness results or checking tools are available.

5.2 Cyclone

Cyclone is designed as a dialect of C, trying to prevent safety violations in programs. It uses a combination of static analysis and inserted run-time checks [12]. The goal is to provide a safety level equal to high-level languages like Java, but still accommodate the low-level programming style from C. Like EC−−, no effectiveness study has been performed and no checking tool is available.

5.3 JSF Air Vehicle C++ Coding Standards

The Joint Strike Fighter Air Vehicle Coding Standards are designed by Lockheed Martin for the development of C++ software for Air Vehicles, but also recommended for non-Air Vehicle C++ software [14].

The guidelines do not only contain hard coding rules, but also rules about optimization of programming code, for example Rule 216:

> The overall performance of a program is usually determined by a relatively small portion of code. This is often referred to as the *80-20 Rule* which states that 80% of the time is spent in only 20% of the code. Thus, design and coding decisions should be made from a safety and clarity perspective with efficiency as a secondary goal. Only after adequate profiling analysis has been performed (where the true bottlenecks have been identified) should attempts at optimization be made.

5.4 ISO Standards

We found the following ISO standards that have to do with software quality:

- **ISO/IEC 9126** – Software engineering - Product quality
- **ISO/IEC 25010** – Systems and software engineering - Systems and software Quality Requirements and Evaluation - System and software quality models
- **ISO/FDIS 26262** – Road vehicles - Functional safety

ISO 9126 defines a model for measuring software quality in six characteristics:

- Functionality
- Reliability
- Usability
- Efficiency
- Maintainability
- Portability

The standard also describes three types of metrics to be used when measuring quality: internal metrics, external metrics and quality in use metrics [13, 2].

ISO 25010 revises ISO 9126 and incorporates the same characteristics with some amendments, and more extensive information about testing whether the quality of a product is good sufficient [11].

ISO 26262 is a specific standard about the functional safety of road vehicles, providing a full safety life cycle for automotive products, and requirement for the validation of all steps to ensure the required safety level is achieved.

These standards are no coding standards or guidelines, but give a more conceptual approach to software quality and safety.

6 Conclusion

We found that MISRA, despite its drawbacks that are mainly rules with a very small effectiveness, can have a positive impact on the quality of software.

Also, we found that AUTOSAR is a promising architecture for automotive systems. The architecture is developed by large car manufacturers and their suppliers. The architecture covers more than just coding standards: it supplies a full model for the development of all components in an automotive product. However, not much specific information about the results and effectiveness is available in literature.

Other coding standards we found are experimental and have not proven to be working better (EC−−, Cyclone) or are not suitable for our project (JSV AV C++ Coding Standards) because they are designed for another language. The applicable ISO standards are mainly focused on general aspects of software quality, and not specifically on writing code.

The Formula Student rules still require us to follow the MISRA guidelines, but we will keep the results of this study in mind when developing, and use the newly gathered knowledge to make wise decisions about deviating from the MISRA guidelines.

The AUTOSAR architecture is too extensive to be fully implemented in this project, because it is designed for much larger systems. The modular approach, however, is also useful for a smaller system like the Forze 4, so we will try to use this in our project where possible.

References

[1] E. N. Adams. Optimizing preventive service of software products. *IBM Journal of Research and Development*, 28(1):2 –14, jan. 1984.

[2] W. Basalaj. Correlation between coding standards compliance and software quality. *IEE Seminar Digests*, 2005(11311):46–46, 2005.

[3] C. Boogerd and L. Moonen. Assessing the value of coding standards: An empirical study. In *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, pages 277 –286, 28 2008-oct. 4 2008.

[4] S. Bunzel. AUTOSAR - the Standardized Software Architecture. *Informatik Spektrum*, 34(1):79–83, 2011.

[5] E. W. Dijkstra. Letters to the editor: go to statement considered harmful. *Commun. ACM*, 11:147–148, March 1968.

[6] H. Espinoza, K. Richter, and S. Gérard. Evaluating MARTE in an Industry-Driven Environment: TIMMO's Challenges for AUTOSAR Timing Modeling. In *Conf. on Design, Automation and Test in Europe (DATE), MARTE Workshop*, 2008.

[7] L. Hatton. Safer language subsets: an overview and a case history, MISRA C. *Information and Software Technology*, 46(7):465 – 472, 2004.

[8] L. Hatton. EC−− a measurement based safer subset of ISO C suitable for embedded system development. *Information and Software Technology*, 47(3):181 – 187, 2005.

[9] L. Hatton. Language subsetting in an industrial context: A comparison of MISRA C 1998 and MISRA C 2004. *Information and Software Technology*, 49(5):475 – 482, 2007.

[10] H. Heinecke, K. Schnelle, H. Fennel, J. Bortolazzi, L. Lundh, J. Leflour, J. Maté, K. Nishikawa, and T. Scharnhorst. Automotive open system architecture-an industry-wide initiative to manage the complexity of emerging automotive e/e architectures. 2004.

[11] ISO, Geneva, Switzerland. Systems and software engineering - Systems and software Quality Requirements and Evaluation - System and software quality models, 2011.

[12] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. Cyclone: A Safe Dialect of C. In *Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference*, pages 275–288, Berkeley, CA, USA, 2002. USENIX Association.

[13] Y. Kanellopoulos, P. Antonellis, C. Makris, E. Theodoridis, C. Tjortjis, and N. Tsirakis. Code quality evaluation methodology using the iso/iec 9126 standard. *CoRR*, 2010. to appear.

[14] Lockheed Martin. Joint Strike Fighter Air Vehicle C++ Coding Standards for the System Development and Demonstration Program. Document Number 2RDU00001, Rev C, 2005.

[15] S. Parker. Comparison of MISRA C testing tools, 2001.

[16] S. L. Pfleeger and L. Hatton. Investigating the influence of formal methods. *Computer*, 30:33–43, February 1997.

[17] A. Sangiovanni-Vincentelli and M. Di Natale. Embedded system design for automotive applications. *Computer*, 40(10):42 –51, oct. 2007.

[18] M. J. Squair. Issues in the application of software safety standards. In *Proceedings of the 10th Australian workshop on Safety critical systems and software - Volume 55*, SCS '05, pages 13–26, Darlinghurst, Australia, Australia, 2006. Australian Computer Society, Inc.

[19] The Motor Industry Software Reliability Association. Guidelines for the Use of the C Language in Critical Systems, 2004.

[20] S. Voget and P. Favrais. How the concepts of the Automotive standard" AUTOSAR" are realized in new seamless tool-chains. In *Embedded Real Time Software and Systems conference (ERTS^2 2010), Toulouse, France*, 2010.