Czech Technical University in Prague
Faculty of Electrical Engineering

Diploma thesis

Evaluation of Safety and Security Properties of Automotive Motor Control Software

2014 **Michal Kreč**

Poděkování

Na tomto místě bych rád poděkoval své rodině za jejich neutuchající podporu během doby mého studia – bez nich bych tuto práci nikdy nezačal.

Také zde chci poděkovat svému vedoucímu diplomové práce – Michalu Sojkovi – za jeho vstřícnost a ochotu vždy poradit – bez něj bych tuto práci nidky nedokončil.

České vysoké učení technické v Praze
Fakulta elektrotechnická

Katedra kybernetiky

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student:	Bc. Michal K r e č
Studijní program:	Kybernetika a robotika (magisterský)
Obor:	Robotika
Název tématu:	Vyhodnocování bezpečnosti softwaru pro řízení motorů v automobilových aplikacích

Pokyny pro vypracování:

1. Seznamte se se softwarovým modulem eMotor firmy Infineon, který slouží k řízení elektrických BLDC motorů v automobilových aplikacích vyvíjených dle standardu AUTOSAR. Dále se seznamte s vývojovou sadou TriBoard a procesorem TC1798, pro které je eMotor vyvíjen.
2. Za pomoci prostředí Matlab/Simulink vytvořte testovací stolici pro software-in-the-loop a hardware-in-the-loop simulace. Změřte časové parametry eMotor softwaru (doba vykonávání jedné iterace regulátoru apod.).
3. Na testovací stolici implementujte testy pro ověření správnosti funkcí zajišťujících bezpečný provoz motoru (tzv. safety measures).
4. Integrujte eMotor s protokolem MaCAN (Message Authenticated CAN) a ověřte opět funkci safety measures za přítomnosti kybernetických útoků přes sběrnici CAN.
5. Výsledky přehledně zdokumentujte.

Seznam odborné literatury:

[1] Infineon: TC1798 32-bit single-chip microcontroller user's manual
[2] Infineon: MC-ISAR_AUDO_UM_EmoDriver documentation, Release V1.3

Vedoucí diplomové práce: Ing. Michal Sojka, Ph.D.

Platnost zadání: do konce letního semestru 2014/2015

L.S.

doc. Dr. Ing. Jan Kybic	prof. Ing. Pavel Ripka, CSc.
vedoucí katedry	**děkan**

V Praze dne 10. 1. 2014

Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Cybernetics

DIPLOMA THESIS ASSIGNMENT

Student:	Bc. Michal K r e č
Study programme:	Cybernetics and Robotics
Specialisation:	Robotics
Title of Diploma Thesis:	Evaluation of Safety and Security Properties of Automotive Motor Control Software

Guidelines:

1. Make yourself familiar with the eMotor software module developed by Infineon company in compliance with AUTOSAR standard, that is used to control electric BLDC motors in automotive applications. Next make yourself familiar with TriBoard development kit and TC1798 processor, for which is the eMotor developed.
2. Using the Matlab/Simulink environment create a testbed for software-in-the-loop and hardware-in-the-loop simulations. Measure time parameters of eMotor software (duration of execution of one controller iteration and similar).
3. Implement tests for verification of eMotor safety measures on the testbed.
4. Integrate eMotor with MaCAN protocol (Message Authenticated CAN) and verify again the functionality of the safety measures in the presence of cybernetic attacks via CAN bus.
5. Document the results.

Bibliography/Sources:

[1] Infineon: TC1798 32-bit single-chip microcontroller user's manual
[2] Infineon: MC-ISAR_AUDO_UM_EmoDriver documentation, Release V1.3

Diploma Thesis Supervisor: Ing. Michal Sojka, Ph.D.

Valid until: the end of the summer semester of academic year 2014/2015

L.S.

doc. Dr. Ing. Jan Kybic	prof. Ing. Pavel Ripka, CSc.
Head of Department	**Dean**

Prague, January 10, 2014

Prohlášení autora práce

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne ....

Podpis autora práce

Abstrakt

V automobilovém průmyslu bylo vždy důležité zaručit bezpečnost vyvíjených produktů a v poslední době je nutné zaručit i zabezpečení proti neoprávněné manipulaci. V současné době se tyto dvě aktivity řeší odděleně. V této práci se zabývám testováním řídícího softwaru v rámci architekrutry AUTOSAR z heldiska bezpečnosti i zabezpečení a snažím se ukázat, že toto oddělení není v některých fázích vývoje nutné. Za tímto účelem jsem vyvinul testovací stolice pro software-in-the-loop a hardware-in-the-loop simulaci, které se dají použít pro testování jak bezpečneosti tak zabezpečení. Jako příklad testovaného produktu používám softwarový modul pro řízení elektromotorů, v současnosti vyvíjený firmou Infineon, a provádím na něm množství testů pro ověření funkčnosti bezpečnostních opatření. Výslekdy naznačují, že s výjimkou několika drobných problémů, bezpečnostní opatření fungují správně.

Abstract

In automotive domain, it is paramount to ensure safety, and lately also security, of developed products. So far safety and security are handled separately. In this work we deal with safety and security testing and validation of control software in the AUTOSAR architecture and try to show that the separation of those two issues is not necessary. We do that by developing software-in-the-loop and hardware-in-the-loop testbeds and showing that they can be used for both safety and security testing. We use a SW module for control of electromotors, that is currently under development by Infineon Technologies, as an example of tested product and execute number of tests to verify correct functionality of implemented safety measures. The results show, that apart from few minor problems, the safety measures function correctly.

Contents

Terms and definitons

Here we describe terms and abbreviations used in this work. First and foremost we must define two crucial terms:

safety – freedom from the risk of causing death, injury or damage to the environment

security – resistance of the system against outside attacks

Simply put **safety** prevents the system from causing damage to the outside and **security** prevents the outside from causing damage to the system.

Other than that, this document uses mostly terminology defined in ISO 26262 [1], definitions of most important terms are listed below. For the full list of terms please see the standard.

anomaly – condition that deviates from expectations, based, for example, on requirements specifications, design documents, user documents, standards, or on experience

detected fault – **fault** whose presence is detected within a prescribed time by a safety mechanism that prevents the fault from being latent

error – discrepancy between a computed, observed or measured value or condition, and the true, specified, or theoretically correct value or condition

failure – termination of the ability of an element, to perform a function as required

fault – abnormal condition that can cause an element or an item to fail

functional safety – absence of unreasonable risk due to hazards caused by malfunctioning behaviour of E/E systems

safety measure – activity or technical solution to avoid or control **systematic failures** and to detect random hardware failures or control random hardware failures, or mitigate their harmful effects

systematic failure – **failure**, related in a deterministic way to a certain cause, that can only be eliminated by a change of the design or of the manufacturing process, operational procedures, documentation or other relevant factors

testing – process of planning, preparing, and operating or exercising an item or an element to verify that it satisfies specified requirements, to detect **anomalies**, and to create confidence in its behaviour

List of abbreviations:

ADC	– analog to digital converter/conversion
AUTOSAR	– automotive open system architecture
BC	– block commutation
BLDC	– brush-less direct current (motor)
CAN	– controller area network
FOC	– field oriented control
PMSM	– permanent magnet synchronous motor

1. Introduction

If a builder build a house for some one, and does not construct it properly, and the house which he built fall in and kill its owner, then that builder shall be put to death.

—Code of Hammurabi, law 229, ∼1772 BC

The quotation above shows us, that the notion, that professionals should be responsible for the quality of their work, is as old as mankind itself. This concept is, of course, part of modern law as well and, together with professional pride and desire to deliver quality products, provides a strong incentive to test newly developed products to confirm that they meet all necessary criteria. The automotive industry is a prime example of this, because automobiles have one of the highest potentials to cause damage and harm, given their wide-spread use.

With the boom of electronic systems in cars came also the need to ensure their safety, i.e. reduce the probability of their failure and severity of damage that can be caused by it as much as possible. This led to development of safety standards like ISO 26262 [2]. These safety standards provide detailed guidelines and procedures for development of safe electronic systems and respective software. All these procedures involve extensive testing in multiple stages of development and this is the main area of interest of this work.

Later, with the continuing integration of various electronic systems present in modern cars, that brought the ability to access most (if not all) on-board systems from a single entry point (or even remotely), emerged a whole new area of concern – security. It is no longer sufficient to prevent system faults and minimize their impact, but now we need to defend against intentional attacks or misguided efforts to "upgrade" the control software as well. At this point there were already security standards developed (e.g. ISO 15408 [3]), but they were developed for "normal" IT applications, not automotive applications, and therefore lack the necessary degree of consideration for safety features

Because safety is already well-known and well-established in the automotive industry but security is something new and imported from outside, the safety and security issues are usually solved separately. There are separate teams responsible for safety and security incorporation and testing of safety and security measures are usually separate processes. This leads to increased development costs, prolonged development time and possibly even to conflicts between safety and security measures, which may lead to decrease in their efficiency or even thwart their function altogether.

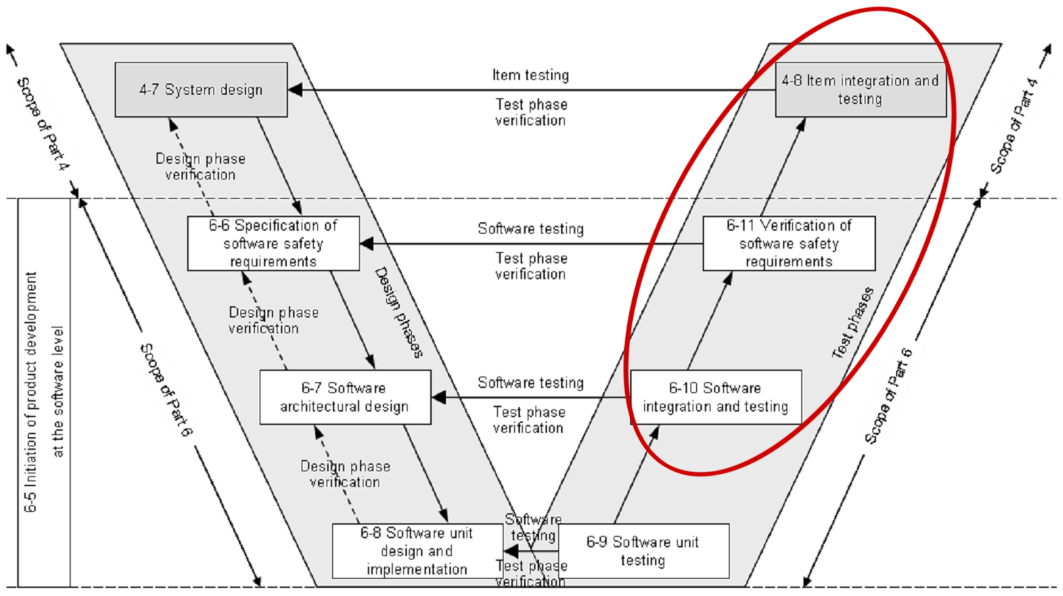
In this work we try to show, that such separation of safety and security is not necessary. We develop a reusable Matlab/Simulink-based testbeds for software- and hardware-in-the-loop simulations [4] for the automotive domain and show that it can be used to test both safety and security measures. We were given the opportunity to test an actual software module prototype, a complex device driver for AUTOSAR architecture named eMotor, currently being developed by Infineon Technologies, so we do have a practical example in this work. The eMotor is a software module for controlling several types of electric motors and is meant to run on TriCore TC1798 processor also developed by Infineon. For the purpose of security testing we integrate the eMotor software with the message authenticated protocol on the CAN bus [5] implemented[1] in our group[2] and use this external interface to execute attacks on the eMotor software.

ISO 26262, as well as number of other standards, suggests the so called V-model as a method to control product development and testing. The V-model describes recommended stages of product development and how they are connected, i.e what are their inputs and outputs and what must be fulfilled in order to advance from one to another. An illustration of the section of the V.model dealing with SW module development taken from ISO 26262 is shown in Figure 1.1, the red circle indicates stages covered in this work. We focus on testing, i.e. executing the evaluated software under different conditions and measuring important parameters, rather than on static analysis. We believe that testing is becoming more and more important, because increasing amount of software modules present in modern cars and interactions between them make the static analysis more difficult and less reliable.

Safety and security testing and their unification is also the topic of European SESAMO (SEcurity and SAfety MOdelling) project[3], to which this work also contributes.

This work is structured as follows. In Chapter 2 we give an overview of Infineon's eMotor driver. Chapter 3 describes the dynamic model of a PMSM motor that we use in our simulations. Chapter 4 describes software- and hardware-in-the-loop testbeds developed for this project. The results of the experiments run on those testbeds are given in Chapter 5 and we conclude with Chapter 6.

[1] `https://github.com/CTU-IIG/macan`
[2] Industrial Informatics Group, DCE FEE CTU
[3] `http://sesamo-project.eu/`



Scope of Part 4
4-7 System design
Item testing
Test phase verification
4-8 Item integration and testing
Scope of Part 4
Design phase verification
6-6 Specification of software safety requirements
Software testing
Test phase verification
6-11 Verification of software safety requirements
6-5 Initiation of product development at the software level
Design phase verification
Design phases
Test phases
Scope of Part 6
6-7 Software architectural design
Software testing
Test phase verification
6-10 Software integration and testing
Scope of Part 6
Design phase verification
6-8 Software unit design and implementation
Software testing
Test phase verification
6-9 Software unit testing

Figure 1.1.: V model for SW development from ISO 26262 and our place in it

2. eMotor driver prototype

The eMotor driver is a software module – more specifically a complex device driver – for the AUTOSAR software architecture [6], currently being developed by Infineon Technologies. Its place in the AUTOSAR architecture is shown in Figure 2.1. It is meant to control electric Permanent Magnet Synchronous Motors (PMSM) and Brushless Direct Current motors (BLDC). It implements two control algorithms: Field Oriented Control (FOC) for PMSMs and Block Commutation (BC) for BLDC motors, providing torque control using PI controller(s). Motors are controlled by generating a PWM signal for the inverter. BLDC motors use 1-phase PWM signal and PMSM use 3-phase PWM signal. In this work, we consider only PMSM motors and therefore only FOC, because that is the new algorithm being developed, the BC comes from previous products.

The eMotor requires measurement of electrical current in the controlled motor (see Section 2.1). It supports several methods and sensor types for determining the position of the motor shaft, including a sensorless option (Section 2.2). The driver is also equipped with a prototype implementation of several safety measures that should detect hazardous states. For example, the generated PWM signal is also read back in order to validate the function of the PWM unit. More details can be found in Section 2.3.

The eMotor has an interrupt based design, where the actual control algorithm is executed in a hardware interrupt handler, which is invoked at the end of the analog-to-digital conversion (ADC) for current measurement [7].

The eMotor is designed to run on the 32-bit TriCore TC1798 microcontroller [8] developed by Infineon in compliance with the ISO 26262 [2]. It was supplied together with ISO 26262 compliant TASKING TriCore VX-toolset[1], which contains C/C++ compiler for TriCore series microcontrollers as well as all other necessary tools (IDE, assembler. linker, etc.). Its configuration is handled by Tresos Studion by ElectroBit[2], which is a configuration management tool designed specifically for AUTOSAR architecture.

2.1. Current sensor configurations

The eMotor driver supports following current measurement configurations:

1. **Two phase parallel** – in this configuration, currents of two phases (i_a and i_b) are measured using two ADCs that measure simultaneously. The current through the third phase i_c is calculated from equation $i_a + i_b + i_c = 0$.

[1]`http://www.tasking.com/products/tricore/`
[2]`http://automotive.elektrobit.com/ecu/eb-tresos-studio`

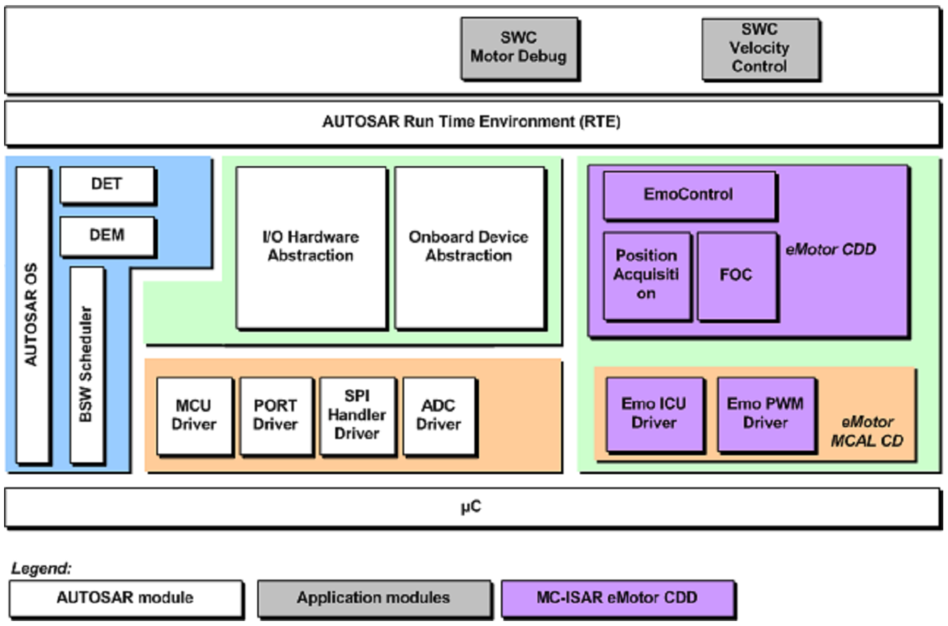


Figure 2.1.: AUTSOAR architecture with eMotor

2. **Two phase sequential** – in this configuration two phases are measured using only one ADC. One phase is measured, then second one and then the first one again, averaging the result, to estimate the value of the first current at the time of the second phase measurement. Then the equation $i_a + i_b + i_c = 0$ is used to calculate the current through the third phase.

3. **Three phase measure** – in this configuration all three phases are measured with two parallel ADCs using the combination of the previous two configurations. One ADC is used to measure i_a, then both ADCs measure i_b and i_c and finally i_a is measured again and averaged with the first measurement. The equation $i_a + i_b + i_c = 0$ can be than used to check for anomalous behaviour.

4. **DC link measurement** – this configuration is used with the Block Commutation algorithm.

2.2. Position acquisition

The eMotor driver supports following motor shaft angle acquisition methods:

1. **Resolver** – with the PMSM a resolver can be used to acquire position of the motor shaft and it can be used either directly, by connecting the resolver to the TriBoard so that clock signal generation and position resolving are done by the TriBoard, or with external resolver-to-digital converter (AD2S1200).

2. **Encoder** – a standard encoder can be used to measure PMSM shaft angle, but it has to be backed by a Hall sensor to provide position before the first zero crossing happens.

3. **Hall** – a Hall sensor can be used to determine commutation state for BLDC motors.

4. **Sensorless** – the eMotor driver provides a sensorless option to calculate shaft position based on internal motor model and current measurements. This can be used for both PMSMs and BLDC motors. Note that when this option is not used for position acquisition it can be used for validation of the actual measurement.

2.3. Safety measures

The eMotor driver prototype implements four main mechanisms to detect anomalous, potentially dangerous, situations. They serve only to detect anomalous situation, responses to the errors were not yet implemented in the version available to us. These features are optional and can be turned off via compile-time configuration. The four safety measures are:

1. **Current validation** – this safety measure calculates the sum of all phase currents and checks it against a given limit (theoretically it should be zero, but practically one should allow for measurement and numerical inaccuracy) and if it is outside of limits an error is reported. This safety measure is meant for PMSMs only and it only works with three phase current measurement, because in other modes the third current is calculated from the other two, so the sum is always zero.

   Besides the above check, there is an option to set upper and lower limits for each phase current, and if the current stays outside of these bounds for a given period of time a different error is reported. Note that this check cannot be disabled via configuration.

2. **Position validation** – when an actual sensor is used for position acquisition the mathematical model of motor, implemented in eMotor driver for sensorless mode, can be used to detect anomalous behaviour. If enabled, the motor shaft position is both read by the sensor and calculated by the model and if these two values differ more than a predefined threshold an error is reported. Note that for the first roughly 10 seconds of eMotor driver execution, the offset between the model and the sensor is being calibrated. During this time no position error can be reported.

3. **PWM diagnostics** – this safety measure reads back the physical PWM signal generated by the controller (an appropriate physical connection outside of the CPU must be established) in order to verify the correct function of the PWM generating hardware and/or connecting wiring (based on where the measurement is taken from). If the read duty cycle differs from the expected one by more than predefined threshold an error is reported.

4. **Memory validation** – with this feature enabled a complete copy of eMotor configuration parameters (motor and controller parameters, safety thresholds, etc.) is stored in the memory and before every major operation (measurements, control action calculation, etc.) the working and backup copies are compared by calculating a checksum with CRC32 algorithm and in case of a mismatch an error is reported. This enables detecting HW memory faults as well as unauthorized manipulation of eMotor parameters.

3. PMSM motor model

In this work a Simulink model of a Permanent magnet synchronous motor (PMSM) is used instead of an actual motor in both SW-in-the-loop and HW-in-the-loop test methods. This allows us to see the influence of hardware on the eMotor behaviour, because both the software and the motor model are the same in both test methods. This also allows greater flexibility in terms of tested motor parameters as well as greater control over the simulated faults and the ability to run SW-in-the-loop tests not in real time speed in exchange for minor simplification of things like friction and increased computational requirements.

The model was downloaded from MATLAB Central[1]. It directly implements differential equations describing a simplified PMSM in the d-q plane:

$$u_d = Ri_d + s\lambda_d - \omega_r \lambda_q \tag{3.1}$$

$$u_q = Ri_q + s\lambda_q - \omega_r \lambda_d \tag{3.2}$$

$$T_e = 3P\left[\lambda_{af} i_q + (L_d - L_q)\, i_d i_q\right]/2 \tag{3.3}$$

where

$$\lambda_q = L_q i_q$$
$$\lambda_d = L_d i_d + \lambda_{af}$$

To convert the voltages and currents from phase values (abc) to d-q plane and vice versa the Park's transformation and its inverse is are implemented in the model. Park's transformation is defined for voltages $u_{a,b,c}$ as:

$$\begin{bmatrix} u_q \\ u_d \\ u_0 \end{bmatrix} = 2/3 \begin{bmatrix} \cos\theta & \cos(\theta - 2\pi/3) & \cos(\theta + 2\pi/3) \\ \sin\theta & \sin(\theta - 2\pi/3) & \sin(\theta + 2\pi/3) \\ 1/2 & 1/2 & 1/2 \end{bmatrix} \begin{bmatrix} u_a \\ u_b \\ u_c \end{bmatrix}, \tag{3.4}$$

where θ is rotor angle w.r.t phase A.

Inverse Park's transformation is then defined as:

$$\begin{bmatrix} u_a \\ u_b \\ u_c \end{bmatrix} = \begin{bmatrix} \cos\theta & \sin\theta & 1 \\ \cos(\theta - 2\pi/3) & \sin(\theta - 2\pi/3) & 1 \\ \cos(\theta + 2\pi/3) & \sin(\theta + 2\pi/3) & 1 \end{bmatrix} \begin{bmatrix} u_q \\ u_d \\ u_0 \end{bmatrix}. \tag{3.5}$$

The same transformations apply for currents [9],[10].

[1]`http://www.mathworks.com/matlabcentral/fileexchange/38804-pmsm-simulation`.

Used symbols

u_d, u_q – d, q axis voltages [V]

u_a, u_b, u_c – a, b, b phase voltages [V]

i_d, i_q – d, q axis currents [A]

i_a, i_b, i_c – a, b, b phase currents [A]

s – derivative operator

P – number of pole pairs

R – stator resistance [Ω]

L_d, L_q – d, q axis inductances [H]

ω_r – rotor speed [rad/s]

T_e – electric torque [Nm]

$\lambda_a f$ – mutual flux [Wb]

$\lambda_d, lambda_q$ – d, q axis flux linkage [Wb]

4. Developed testbeds

This chapter describes the two testbeds developed as part of this work – the software- and hardware-in-the-loop testbeds.

The SW-in-the-loop testbed runs completely on a PC, simulating both the controlled motor and eMotor software. It is used for functionality testing of eMotor software and its safety functions and integration testing of said functions. This testbed does not contain the MaCAN module.

The HW-in-the-loop testbed utilizes actual hardware that the eMotor is intended to run on – TriCore TC1798 microcontroller. Only the controlled motor is simulated in a PC. This testbed contains the MaCAN module and is in fact used for integration testing of said module, as well as for SW/HW interaction and subsystem functionality testing.

We use R2012b version of Matlab/Simulink for all Simulink models used in this work [11].

4.1. Software-in-the-loop testbed

The SW-in-the-loop testbed is a Simulink model (see Figure 4.1), that contains the PMSM model described in Section 3, S-function block with the eMotor code and blocks that simulate various faults. It is designed to work with all current measurements methods except DC link (we focus on FOC rather than on BC) and with all position sensors. The sensorless mode is currently not supported. The eMotor S-function reads shaft angle and 3-phase currents from the motor model and outputs a vector of three PWM duty cycle values, which is converted to voltage in order to act as an input for the motor model. There is also a standard Simulink PI controller block, serving as speed controller. In other words, it controls the eMotor's torque input in order to follow reference speed.

4.1.1. eMotor block

The eMotor block shown in Figure 4.1 is implemented as a C MEX S-function. This means that the block is implemented in C language. It comprises of Infineon's eMotor driver code and Simulink interface code. The eMotor code is modified so that instead of accessing the hardware (e.g. ADC and PWM peripherals) directly it uses Simulink interfaces to read inputs and write outputs. Moreover, the eMotor code is invoked from S-function callback functions rather than by hardware interrupts. This allows us to simulate as much of eMotor behaviour as possible.

The Simulink glue code comprises of Simulink callback functions. The `mdlInitialize-Conditions` callback initializes the eMotor and enables the motor control. The actual

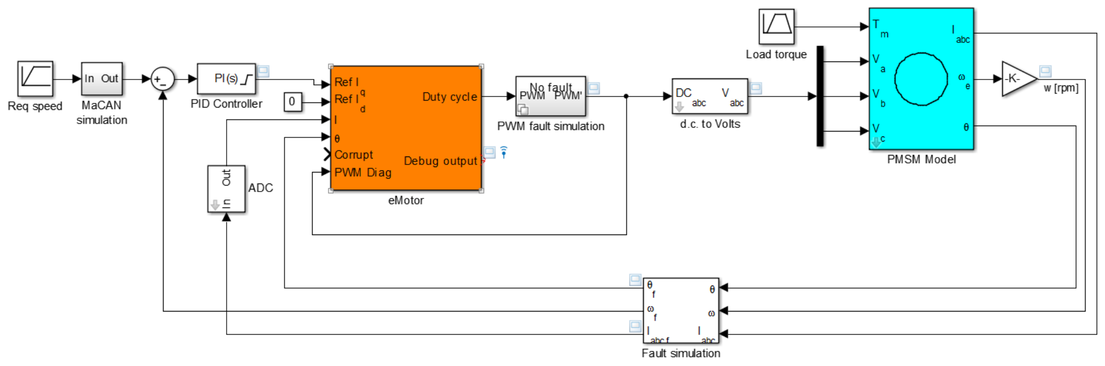


Figure 4.1.: SW-in-the-loop testbed

control algorithm is called in the `mdlOutputs` callback, together with the code that simulates the current measurement and position acquisition and setting of PI controller set point. This callback is called with S-function block sampling frequency, which was set to 20 kHz, the recommended value for eMotor control frequency.

The S-function was compiled with Microsoft C compiler from Visual Studio 10. The eMotor files had to be compiled with this compiler as well, which resulted in the need to modify them in some minor way. This was done by utilising C preprocessor and having separate code modules for Simulink and for TriBoard targets, allowing the same source files to be used for both targets. It was also necessary to handle numerous direct register read/writes in code. In order to minimize the differences between the Simulink and TriBoard targets a dedicated memory area is allocated in the S-function and register addresses are modified to point to that memory. The actual written values are usually of no interest because in most cases they control low level hardware behaviour irrelevant in Simulink simulation. Register reads are, when needed, replaced by setting the local variables directly to the values from Simulink input ports.

4.1.2. Fault simulation

The testbed naturally supports simulation of sensor and/or control faults. This is accomplished by variant subsystem blocks inserted between the motor model and the eMotor block. These subsystems allow for different behaviour based on the value of a control variable. We introduced several *fault locations* in the Simulink model where faults can be simulated. The locations are: current measurement, position measurement and PWM diagnosis. At each location we are able to simulate several *fault types* (e.g. additive or multiplicative) and for each fault location there is a vector of start and stop times (e.g. the fault can be active at times 1–5 and 7–9 s). Additionally, for some fault types, their magnitude can altered. Different fault types are described below.

Faults types common to all fault locations

1. **No fault** – no fault is introduced in this setting.
2. **Additive error** – a constant amount, specified by the fault magnitude is added to the signal.
3. **Multiplicative error** – the signal is multiplied by amount specified by the magnitude.

Fault types specific to current measurement

1. **Short circuit** – the current signal is set to zero.

Fault types specific to position measurement

1. **Stuck** – the position is held at the value it had at start of the fault.
2. **Slipping** – maximum rate at which can the position change is limited to the fault magnitude.

Fault types specific to PWM diagnosis

1. **Wire break** – the PWM duty cycle is set to 0.
2. **Min** – the PWM duty cycle cannot drop below fault magnitude.
3. **Max** – the PWM duty cycle cannot rise above fault magnitude.

4.2. Hardware-in-the-loop testbed

In a hardware-in-the-loop simulation, the tested software runs in real time on real hardware. In our case the eMotor driver runs on Infineon's TriBoard.

Our HW-in-the-loop testbed is depicted in Figure 4.2. The TriBoard[1] is connected to a PC via a PCI-based Humusoft MF624 I/O card[2]. PMSM motor model and fault simulation are run on the PC and the I/O card is used to connect the simulated motor with the board.

For real-time simulation, we use so called external simulation mode. In this mode C code is generated from the Simulink model, the code is compiled and run in real-time on the host. Simulink provides only the user interface, i.e. it is possible to tweak model parameters at run time or to see graphs of various signals. In our group[3] a custom code generation target [12] was developed and we run all simulations on Linux with PREEMPT RT patches. With such a setup, we can run the simulation without a

[1]Infineon Order Nr. KIT_TC1798_SK
[2]`http://www.humusoft.cz/produkty/datacq/mf624/`
[3]Industrial Informatics Group, DCE FEE CTU

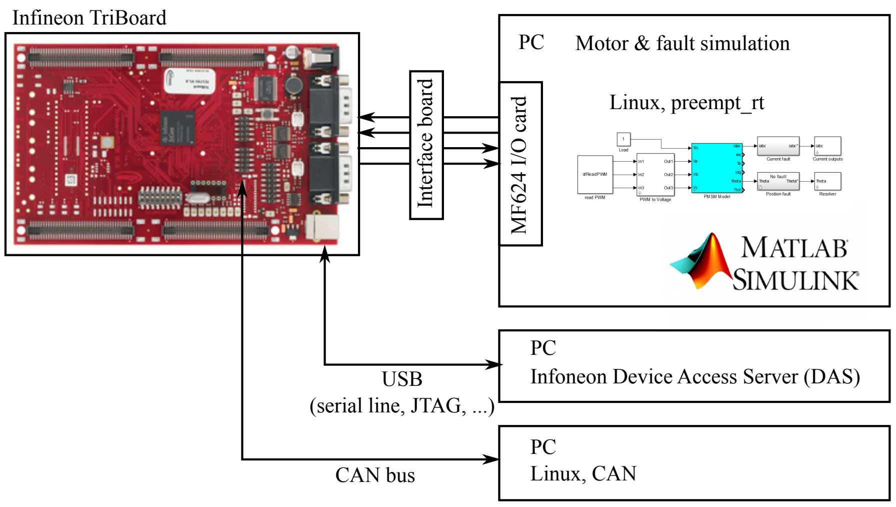


Figure 4.2.: Block diagram of the HW-in-the-loop testbed

deadline miss at 20 kHz sampling frequency. This is the same frequency at which eMotor driver runs on TriBoard.

The computer with CAN interface card was added later, in order to test the influence of CAN communication and message authentication on the eMotor's safety and time properties. This computer was used only in tests covered in Section 5.3.5.

4.2.1. Humusoft MF624 I/O card

The MF624 is a PCI expansion card designed for interconnection of a PC and real world signals. It features 8 channel 14 bit A/D converter, 8 channel 14 bit D/A converter, 8 bit digital input port, 8 bit digital output port, 4 quadrature encoder inputs and 5 timers/counters as well as fully 32 bit architecture. Only D/A converter, digital input and timers are used in the HW-in-the-loop testbed. Specifications are as follows [13].

A/D converter

Resolution:	14 bit
Number of channels:	8 single ended
Conversion time:	1.6 µs single channel
	3.7 µs 8 channels
Input range:	± 10 V
Input protection:	± 18 V
Input impedance:	$> 10^{10}$ Ohm

D/A Converter

Resolution:	14 bit
Number of channels:	8
Output range:	±10 V
Settling time:	max. 31 µs
Slew rate:	10 V/ µs
Output current:	min. ±10 mA
Differential nonlinearity:	±1 LSB

Digital Inputs

Number of bits:	8
Input signal level:	TTL
Logic 0:	0.8 V max.
Logic 1:	2.0 V min.

Digital Outputs

Number of bits:	8
Output signal level:	TTL
Logic 0:	0.8 V max. @ 24 mA (sink)
Logic 1:	2.0 V min. @ 15 mA (source)

Counters/Timers

Number of channels:	5, 4 of them available on I/O connector, one for internal use
Resolution:	32 bits
Clock frequency:	50 MHz
Triggering:	software, external
Clock source:	internal, prescalers, external
Inputs:	TTL, Schmitt triggers
Outputs:	TTL

4.2.2. Humusoft MF624-related Simulink blocks

Several Simulink blocks and corresponding S-functions were written in order to access the used I/O card from Simulink under Linux. These blocks are described in this section.

Analog Input

This block reads specified analog input channels of the MF624 card. This S-function has no inputs and number of outputs is equal to the number of selected channels. Output ranges from -10 to 10 and is equal to measured voltage in Volts.

Analog output

This block controls an analog output channel. The controlled channel is selected with the block's parameter. It has one input, values higher than 9.9988 result in output value of 9.9988 V, values lower than -10 result in output value of $-10\,V$, all values in between result in same output value $\pm$ 1/2 LSB. Values are sent to the card in every time step this block is called (more precisely its `mdlOutputs` function is called). It has no outputs.

Digital Output

This block sets value of a digital output bit specified by its parameter. It has one input and no outputs. Input values lower than or equal to 0.5 result in logical 0, inputs higher than 0.5 result in logical 1.

Digital Input

This block reads specified bit of a digital input and produces output value 0 if the input signal was logical 0 and output value 1 if the input signal wan logical 1.

Read PWM

This block is used to read PWM duty cycles of 3-phase PWM utilizing MF624's timers. Three channels of PWM should be connected to timers input 0, 1 and 2, which are used to gate corresponding timers. The internal timer – number 4 – is running continuously and time counted by those timers between two simulation steps is measured. The duty cycle of phase A (B and C respectively) is then calculated as a ratio of the value counted by timer 0 (1 and 2 respectively) and timer 4. Programming limitations unfortunately do not allow simultaneous read of all timers, so values higher than 1 can occur. These are not filtered in this block, but can be afterwards.

4.2.3. Simulink model

The model used for HW-in-the-loop simulation contains the model of the motor described in Section 3, read PWM block described above, one subsystem called Current outputs that converts phase current values to signals fed to Analog output blocks and measured by TriBoard, one subsystem called Resolver, that imitates resolver function and, of course, fault simulation. The fault simulation was modified to allow fault control from the TriBoard by means of a logical signal that is controlled by eMotor software and read by MF624's digital input. A preset fault is active whenever is this signal in logical 1. This allows precise time measurement of fault response times, because the precise time of fault initiation can be saved alongside the time measurements. Faults used here have the same types as those used in SW-in-the-loop testbed (see Section 4.1) with the exception of PWM fault which is not implemented, because it requires a HW implementation rather than SW one.

The model is compiled into an executable using Simulink Coder with custom code generation target – Linux with PREEMPT RT patch – and then run in external simulation mode, allowing live data viewing while maintaining real time properties.

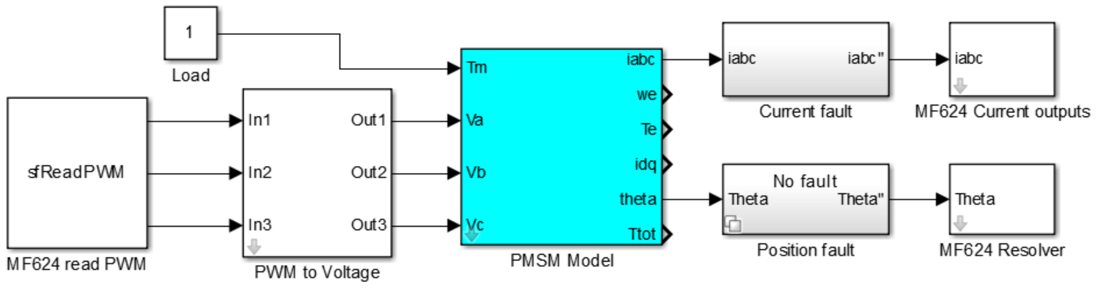


Figure 4.3.: HW-in-the-loop testbed

4.2.4. TriCore application

The eMotor demo application supplied by Infineon was taken as a basis for testing program. This demo application uses the eMotor driver and extends it with a PI speed controller and a simple command line interface accessed over virtual serial port. The user interface allows for switching the motor control on and off, setting the reference speed, reading from position sensors and calibrating them. We extended the application to allow for direct PWM control, safety measures diagnostics (see Section 2.3), time measurement control, fault control and transfer of measured values to a PC.

4.2.5. Interface board

A custom made interface board is used to connect the TriBoard with the MF624 card. It has two 40-pin connectors for MF624's X1 and X2 connectors and seven 16-pin connectors to connect to TriBoard. Connections realized by the interface board are described in Table 4.1 and the board itself can be seen in Figure 4.4.

4.2.6. CAN bus & message authentication

The eMotor driver was enhanced with Infineon's AUTOSAR compatible CAN driver [14] for the purposes of testing the effects of the CAN communication on the eMotor. At first only the effects of presence of a CAN traffic were tested, with 3 different ways of handling message reception (for details see Section 5.3.5). Then, the message authentication extension for CAN (MaCAN [5]) implemented in our [4] utilizing TriCore's Secure Hardware Extension (SHE) was integrated with the eMotor. The necessary *time server* and *key server* run on the PC with the CAN interface [15], together with simple application allowing secure motor control by means of an authenticated signal conveying the reference speed.

[4]Industrial Informatics Group, DCE FEE CTU

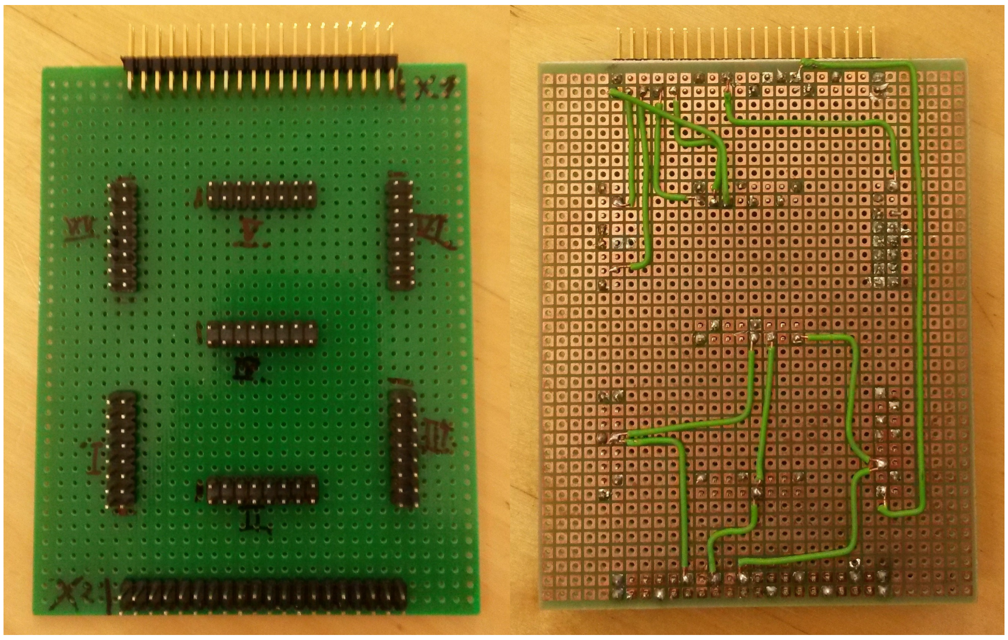


Figure 4.4.: Interface board

MF624 port	TB port	descr.
T0_IN	1.6	COUT60
T1_IN	6.12	COUT61
T2_IN	6.13	COUT62
DA4	AN0	resolver cos
DA3	AN16	resolver sin
DA0	AN2	I-A
DA1	AN18	I-B
DA2	AN4	I-C
T0_IN	4.0	PWM Diag
T1_IN	4.2	PWM Diag
T2_IN	4.3	PWM Diag
AGND	VAGND0	common ground
DIN2	1.4	fault control

Table 4.1.: MF624 to TriBoard connections

5. Experiments

This chapter documents executed tests and their results. It is divided into three sections. First we list testbed and eMotor parameters common in all experiments – motor configuration – and then software-in-the-loop and hardware-in-the-loop experiments – experiments are further grouped based on the focus of the tests.

5.1. Configuration parameters

Below are listed parameters used during the testing.

5.1.1. PMSM motor model parameters

Parameter names and units are kept the same as in original model, with the exception of fixing a typo in friction **vicious** gain → friction **viscous** gain.

Stator resistance:	$1.2\,\Omega$
Inductance L_d:	$0.013\,\mathrm{H}$
Inductance L_q:	$0.013\,\mathrm{H}$
Rotor flux constant λ_{af}:	$0.1546\,\mathrm{V\,rad^{-1}\,s}$
Moment of inertia J:	$0.00176\,\mathrm{kg\,m^2}$
Friction viscous gain B:	$0.00038818\,\mathrm{Nm\,rad^{-1}\,s}$
Number of poles P:	6

5.1.2. eMotor parameters

Because eMotor has dozens of parameters, only those, that are relevant to the measurements or crucial to the functionality are listed. Names of the parameters are left the same as they are in the ElectroBit Tresos tool, used to configure the eMotor.

EmoAlgorithm:	EMO_FOC
EmoCurrentMeasurement:	EMO_THREE_PHASE_MEASURE
EmoPolePairs:	3
EmoPhaseResistance:	1.2
EmoPhaseInductance:	0.013
EmoMaxSpeed:	16000
EmoMaxCurrentLimit:	50
EmoMinPulse:	0

EmoAdcTriggerVariance:	0
EmoIdCurrentPi	
EmoKp:	0.00132645
EmoKi:	0.5
EmoLimit:	1.0
EmoIqCurrentPi	
EmoKp:	0.00132645
EmoKi:	0.5
EmoLimit:	1.0
EmoCurrentPhaseA/B/C	
EmoCurrentGain:	-40
EmoShuntResistance:	0.01
EmoCurrentOffset:	2040
EmoLowerCurrentLimit:	-10
EmoUpperCurrentLimit:	10
EmoPaInputSensor:	EMO_PA_RESOLVER
EmoPaSpeedMeasureInterval:	998
EmoPaResIf:	EMO_PA_RES_GPTA_USED
EmoPwmPeriod:	4998
EmoPwmTimerOffset:	0
EmoPwmDeadTime:	1
EmoPt1FilterGain:	20
EmoPt1FilterTimeConst:	0.00005

5.2. Software-in-the-loop experiments

The SW-in-the-loop testbed can be used only for validation of properties that are not related to either hardware or real-time execution. SW-in-the-loop simulation is best used for verifying logical correctness of the executed software. In the test cases described in this chapter we simulated various faults and observed which errors are detected.

We automated the execution of test cases with a Matlab script that loads the test cases (see below) from another m-file, executes them and saves simulation results. For each test case type, fault locations, types, start times, stop times and magnitude can be set. Also reference speed for the PI controller and motor load can be specified independently for each test case, but for simplicity we used the same values for all test cases. As can be seen in Figure 5.1, reference speed starts at 0 and increases linearly to 3000 rpm between time 0.2 and 4 s. The motor load starts at 0.1 Nm, then it linearly increases to 5 Nm in time from 1 to 4 s and then linearly decreases back to 0.1 Nm from 8 to 10 s. All simulations described below had duration of 10 s.

In the figures below, we can see the results of simulations. The bottom part of each

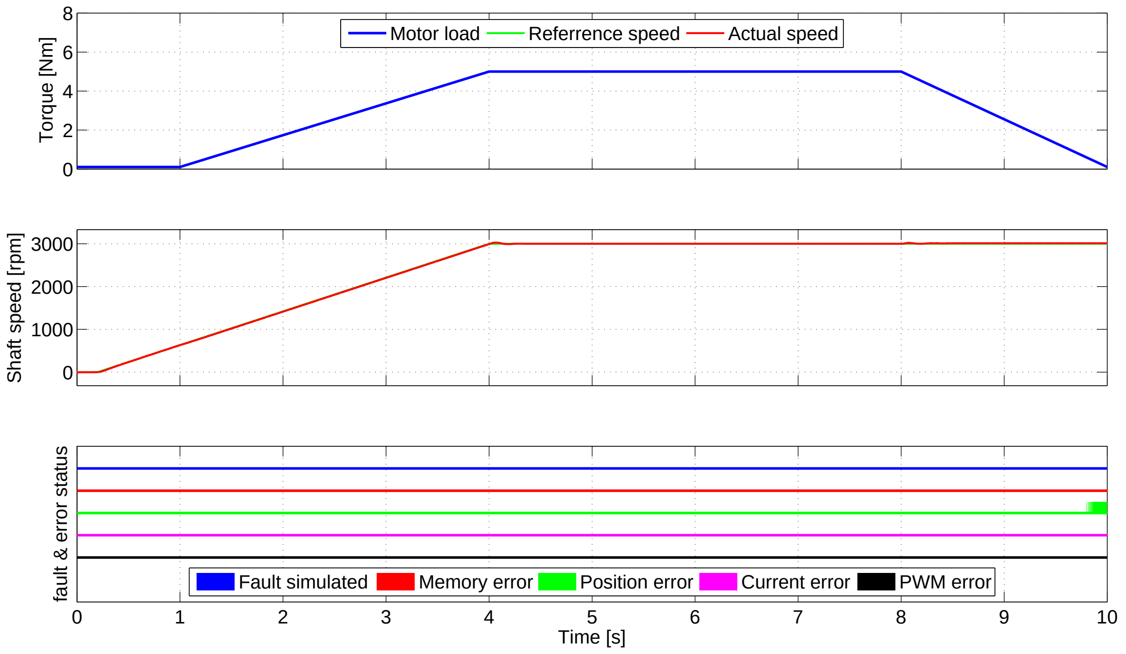


Figure 5.1.: Test case results when no fault is simulated.

figure shows the status of when the faults were simulated and when various errors were detected. Thin line means that no fault was simulated or no error was detected while thick line means the opposite. Colors denote different errors (see the legend in the figures).

5.2.1. No fault test case

In this test there are no faults simulated in order to test the normal operation and susceptibility to false errors.

In Figure 5.1 we can see a false position validation error occurring near the end of the simulation. This error is signalled when there is inconsistency between simulated motor and eMotor's internal motor model. This inconsistency only appears when the motor torque decreases and it causes the measured and calculated position to slowly diverge. The false error occurs when the two position differ more than a predefined threshold.

Because we cannot run tests with actual motor as of yet, we are unable to determine which model is correct or even whether we configured the eMotor properly. This being said, we think that the current version of the eMotor documentation is not clear enough with regards to the eMotor's motor model parameters (particularly definitions of used terms and vague description of filter parameters). It will be necessary to specify what are the valid operating conditions for the motor load and how to set the parameters of the eMotor's internal motor model.

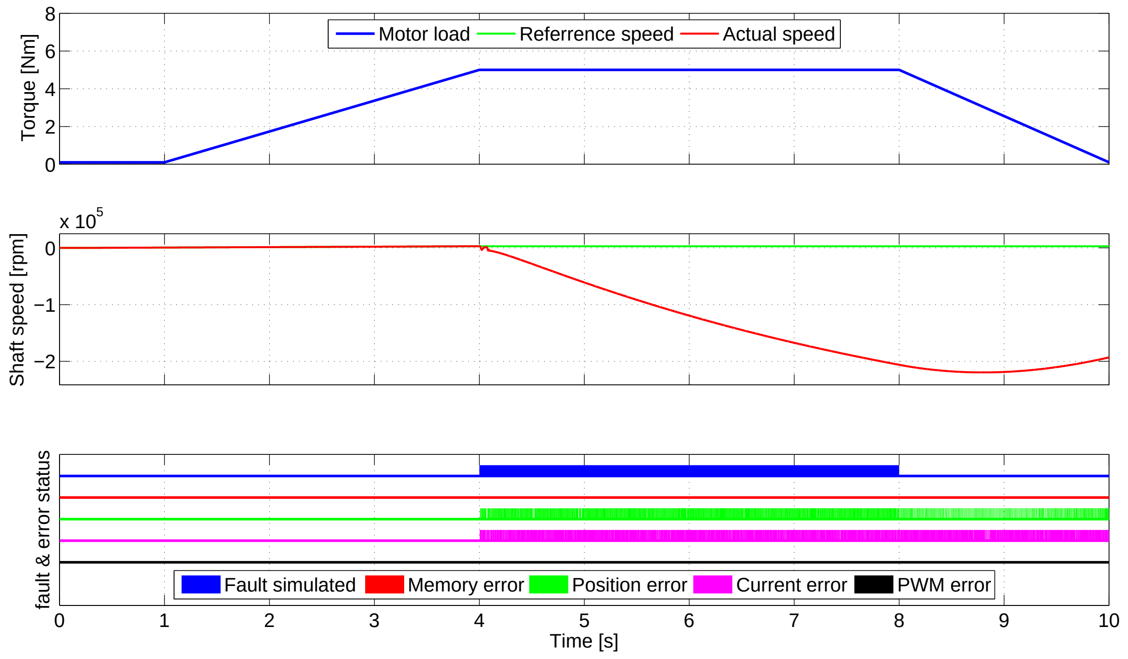


Figure 5.2.: Position fault

5.2.2. Position sensor fault

In this case the position sensor becomes stuck at time 4 s, outputting the last value read before that event. Then it reverts back to the normal operation at 8 s.

Results in Figure 5.2 show, that the position sensor fault was successfully detected. The detection delay was 6.5 ms. It is worth mentioning that a current error was also reported (with delay of 3.5 ms) this is caused by the motor currents exceeding the range of simulated ADC, thus invalidating the measurements.

5.2.3. Phase A current measurement fault

In this test case we simulated short circuit fault for current measurement of phase A between 4 and 8 s.

Figure 5.3 shows that the current measurement fault was successfully detected and in this case immediately as it was introduced (in the same simulation step). It also shows that a position error is reported simultaneously with the current error. The reason is described in the next section (5.2.4).

If the figure is zoomed enough, one can see that the reported current error is periodically interrupted. This is expected, because when the actual current crosses zero the sum of currents calculated from the measurement is valid. Since the error detection uses a threshold greater than zero, the current is valid in a small surrounding of the zero crossing.

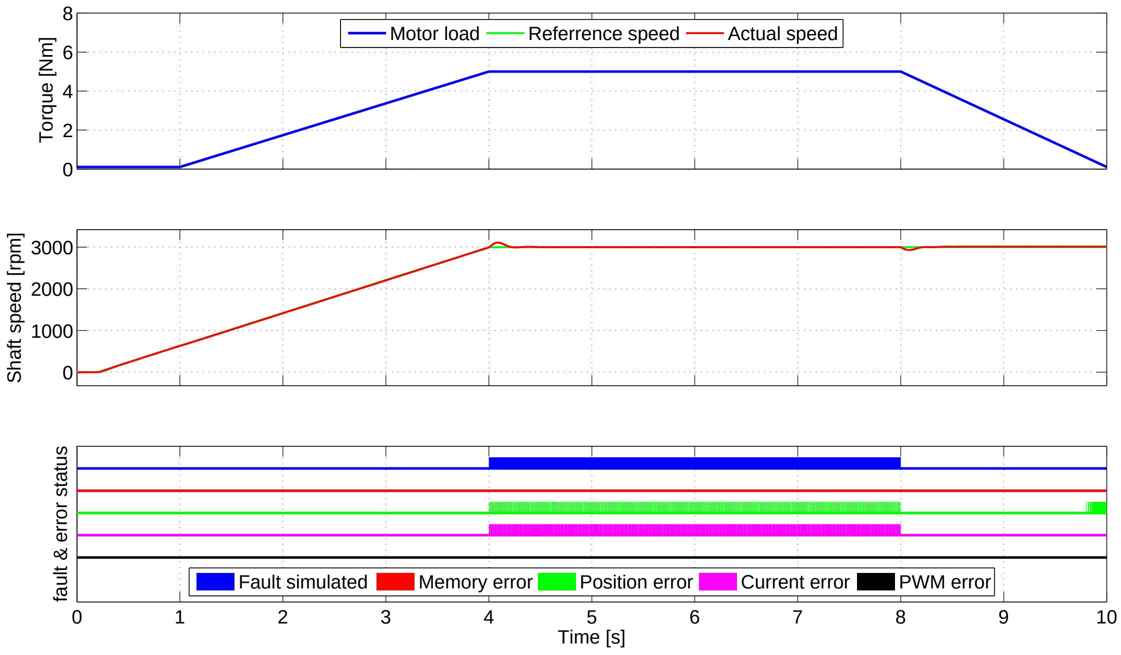


Figure 5.3.: Current phase A short-circuit fault

5.2.4. Phase B current measurement fault

In this test case we simulated short circuit fault for current measurement of phase B between 4 and 8 s.

Figure 5.4 shows that the current measurement fault was successfully detected and in this case immediately as it was introduced (in the same simulation step). Similarly to the previous test case the reporting of the current error is periodically interrupted.

Now you can see that, unlike in the previous case, the position error was not reported in this case permanently, but only for a short time. This is caused by the implementation of Clarke's transformation used for position estimation. Clarke's transformation transforms phase currents i_a, i_b and i_c to α–β currents i_α, i_β as follows:

$$i_\alpha = \frac{3}{2}(i_a - 1/2 i_b - 1/2 i_c), \tag{5.1}$$

$$i_\beta = \frac{1}{\sqrt{3}}(i_b - i_c), \tag{5.2}$$

but when one assumes that $i_a + i_b + i_c = 0$ the calculation of i_α can be simplified to

$$i_\alpha = i_a \tag{5.3}$$

and this is exactly what is implemented in the eMotor. The consequence is that when $i_a = 0$, $i_\alpha = 0$ as well. When $i_\alpha = 0$ the internal motor model used for position validation stops, resulting in discrepancy between measured and estimated position.

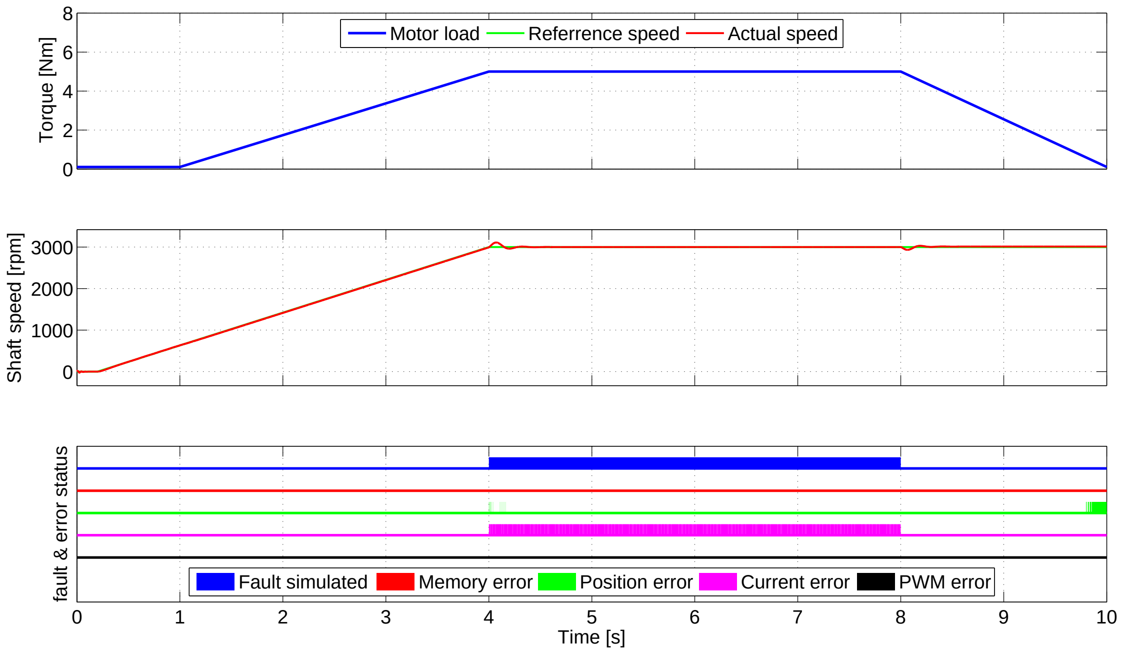


Figure 5.4.: Current phase B short-circuit fault

However when one of the $i_b, i_c = 0$ the i_β only changes from one sinusoid to another with slightly different amplitude and phase but same frequency. This results in some drift in estimated position but this drift falls under the set threshold so no position error is reported.

As a consequence of this the correct functionality of position validation depends on correct functionality of current measurements. So in the final implementation this must be taken into account. Suppressing the position errors when current error is reported seems like one simple solution, but more complex one may be required.

5.2.5. Phase C current measurement fault

In this test case we simulated short circuit fault for current measurement of phase C between 4 and 8 s.

Figure 5.5 shows that the results are almost the same as for phase B described in the previous section.

5.2.6. PWM wire break

In this test case the PWM signal wires for all phases break at 4 s, outputting duty cycle of 0, which results in maximum voltage applied to all phases, and then they revert back to normal operation at 8 s.

Results can be seen in Figure 5.6. In this figure we replaced the load profile, which was the same as in previous cases, with phase A PWM signal sent out by the eMotor

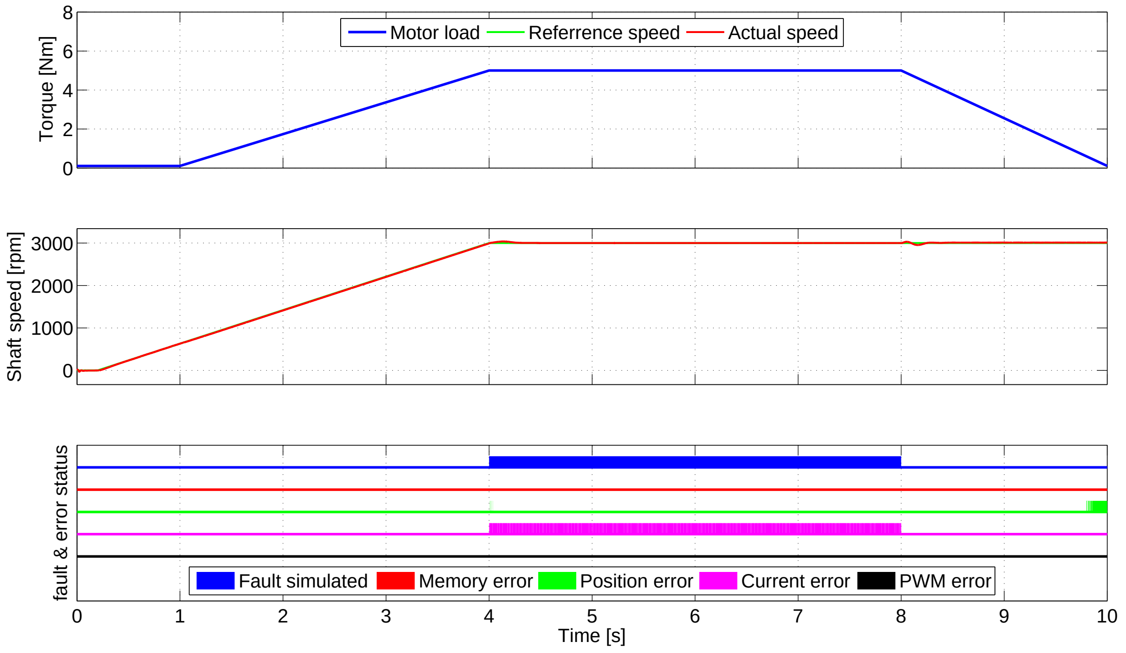


Figure 5.5.: Current phase C short-circuit fault

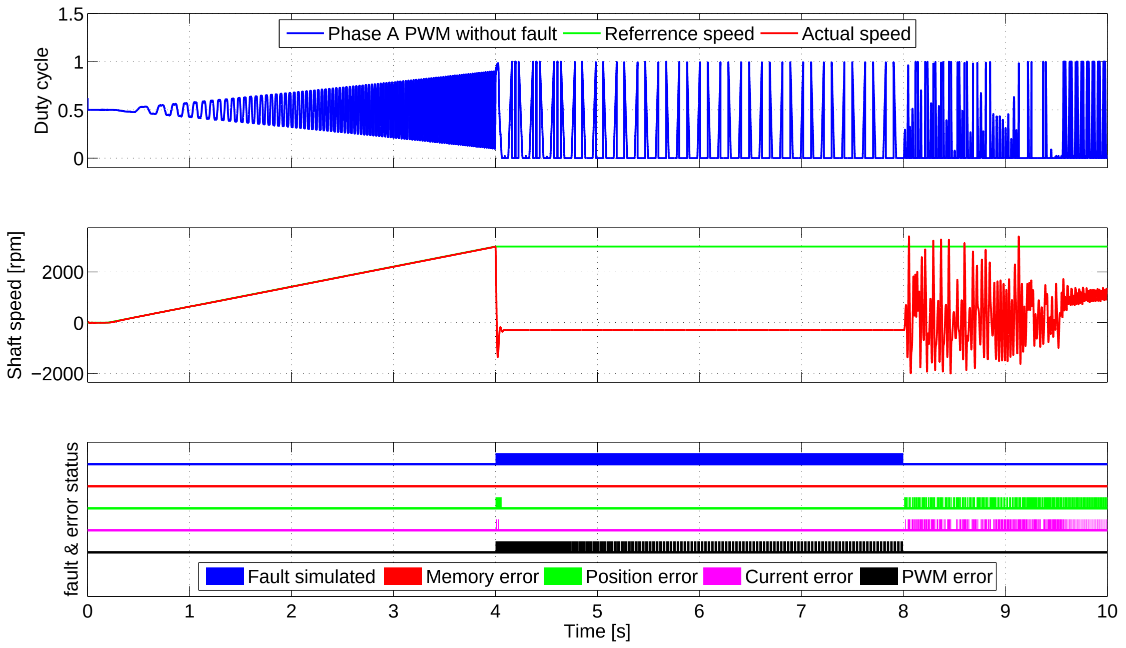


Figure 5.6.: PWM fault

– so this signal is fed to the motor only when the fault is not simulated. The other two phase signals had similar shape so we left them out of the picture for the sake of readability. The PWM fault was successfully detected and the detection delay was one eMotor algorithm cycle (50 µs). This is the minimal possible delay, because the PWM diagnostic is implemented as a two step process alternating reading and validation in each algorithm cycle. At the beginning of the fault there are current and position errors reported, these are caused by the reaction of the motor to the loss of control and maximal voltage applied to all phases. After a short delay the motor stabilizes and those errors disappear. Because there eMotor driver does not react to safety errors yet, the implemented the controller is not stopped and as soon as the fault is over the signals affected by controller wind-up are fed to the motor and this results in erratic behaviour and reported current and position errors that can be seen in the figure.

5.3. Hardware-in-the-loop experiments

The main goal of these experiments is to measure the execution time of the eMotor controller and see how it is influenced by the implemented safety and security functions as well as by possible faults and attacks.

The execution time of the main controller loop (the one called by periodic ADC interrupt) is measured using on-board system timer (STM) and with each measurement the information whether each error was reported during that cycle is stored. Also the fault initiation time and the time when an error was first reported are measured, allowing exact measurement of the fault detection delay.

The STM runs at 100 MHz thus having 10 ns resolution. The timer itself is 56 bit wide but only the lowest 32 bits are read and stored, because the overflow (which in the lowest 32 bits occurs every 42.9 s) can be easily handled during data post-processing. The start and end times of each loop and 4 error flags are stored in on-board RAM.

By using 1 MiB of available SRAM (unused by the original eMotor software) we can store 87 381 measurements, which at the 20 kHz control frequency results in slightly under 4.5 s of run time. Each measurement needs 12 bytes of memory – two 32 bit unsigned integers for timestamps and four 8 bit unsigned integers for error flags. Although it should be possible to use bit variables for error flags, the size of the measurement structure would not be multiple of 4 bytes, which would cause aligned memory access exceptions.

After the end of the experiment, the measured data are transferred to the PC via virtual serial port provided by the TriBoard.

5.3.1. Test cases

The tests consisted of starting the idle motor with constant 1 Nm load to 1000 rpm and holding it there for the entire measurement. In all experiments, the reference speed was set to 1000 rpm. Unless said otherwise all measurements were initiated after the motor speed stabilized.

5.3.2. No safety measure enabled

In this section, we describe the experiments when no safety measure was configured in the eMotor driver, i.e. the safety measures are not compiled into the eMotor binary. Therefore, we measure the properties of the control algorithm itself. In these experiments, error information was not saved, because it is impossible for eMotor to report any errors in this configuration, resulting in increased number of measurements.

No fault

In this case no fault was simulated. With this test, we want to investigate the properties of normal operating conditions.

The results can be seen in Figure 5.7. Figure 5.7a shows distribution of execution time over experiment time and figure 5.7b shows a histogram of those execution times.

Most eMotor invocations have execution time between 10.2 and 10.7 µs. Although it may not be clear from the figure it is exactly every 20th invocation that is cca 1 µs longer than the rest. This is most likely caused by the speed PI controller that runs at 1 kHz and causes cache trashing.

We have also measured the time interval between two control algorithm invocations. Result of this measurement can be seen in Figure 5.7c. The bins in this histogram are across all distinct values and one interesting thing to note is, that when the invocation is off, it is more likely to be off by an even number of ticks, than by an odd number of ticks. We don't know what causes this behaviour and it is most pronounced in this case – when there are no safety measures enabled and no faults present.

Current phase A fault

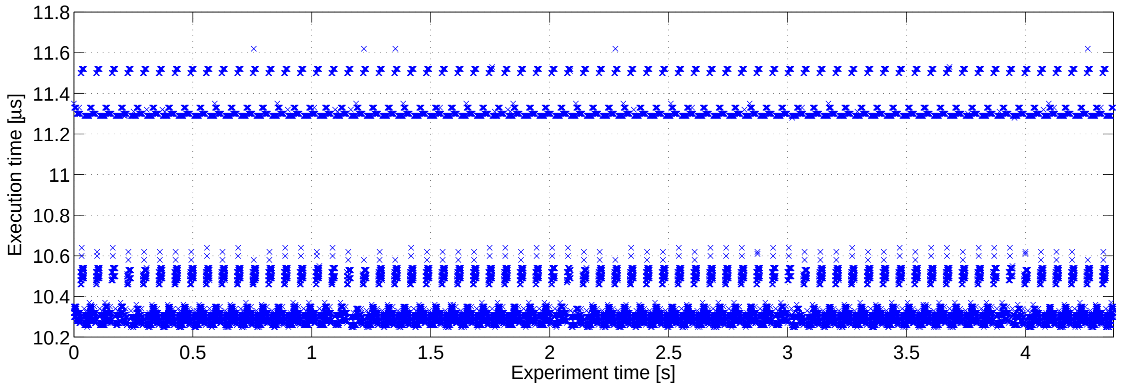
In this case phase A fault was simulated to check the effects of a fault on the timing properties of the eMotor driver.

The results can be seen in Figure 5.8. Algorithm execution time is the same as in the previous case. Therefore, it can be claimed that with no safety measures the current measurements faults do not interfere with the algorithm execution.

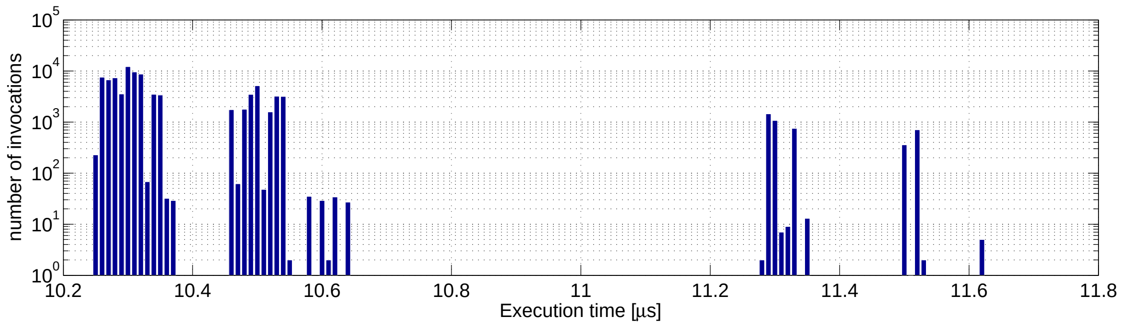
Position fault

In this case the "stuck" position fault (see Section 4.1.2) was simulated from 0.9 s onward to check the effects of a fault on the timing properties of the eMotor driver.

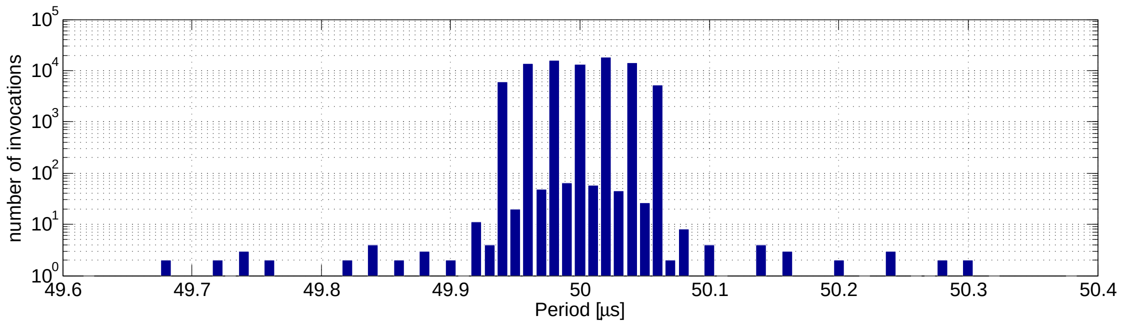
Results can be seen in Figure 5.9. These results differ from all others, because there is significant change in execution time from the moment the fault was initiated, although this change seems to be positive – there are no slightly longer executions after the fault – it is abnormal and only occurs with position faults and we were unable to find out what causes it, so we think that someone with better understanding of TriBoard's and eMotor's architecture should look into it. Moreover after short time (cca 0.7 s) all PWM channels were set to maximal duty cycle resulting in stopped motor. We are not sure what caused this behaviour, but one possibility is controller wind-up.



(a) Execution time



(b) Histogram of execution time



(c) Histogram of invocation period

Figure 5.7.: No safety measure, no fault

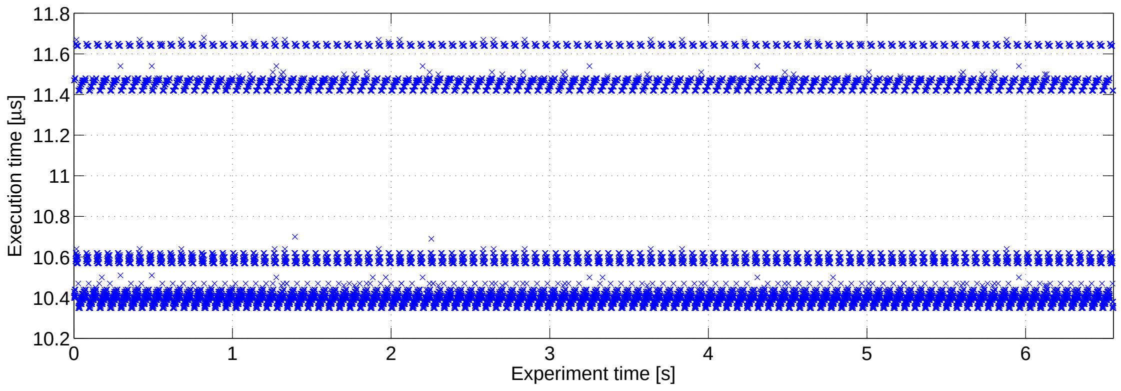


Figure 5.8.: No safety measure phase A fault

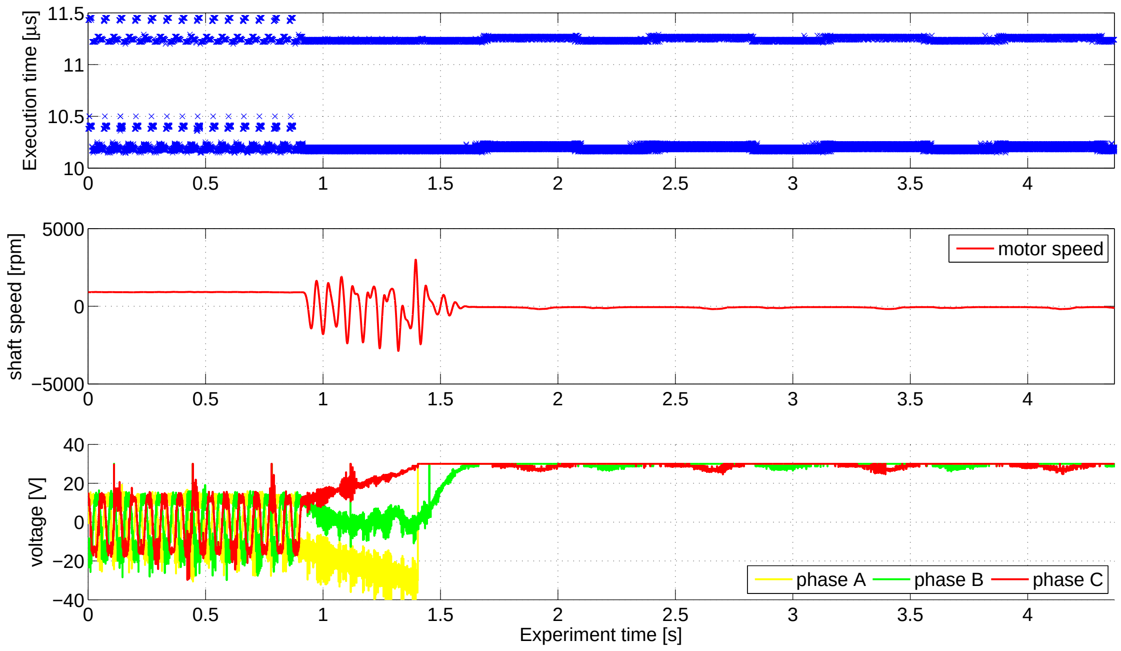


Figure 5.9.: No safety measure position fault

5.3.3. Single safety measure enabled

In this section, we perform experiments with only one safety measure enabled (compiled in) and without faults. This allows us to see the overhead caused by the respective safety measure.

Current validation, no fault

In this case only the current validation was enabled and no fault was simulated.

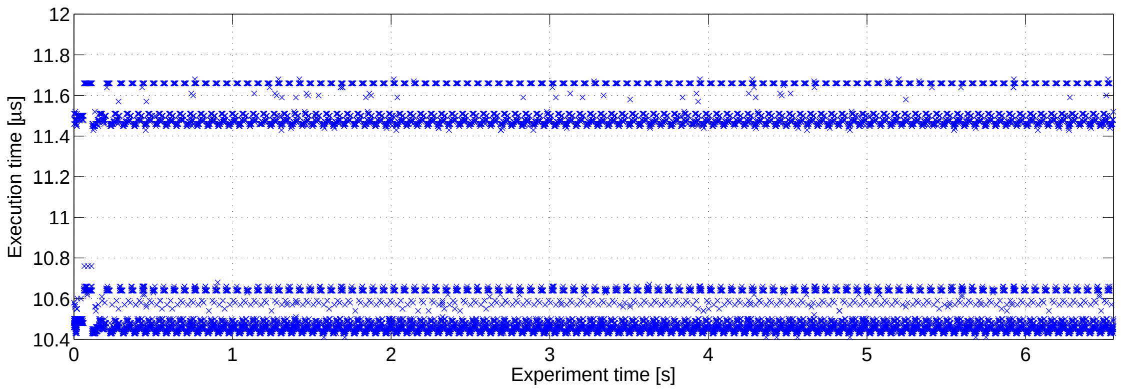


Figure 5.10.: Current validation, no fault

Results in Figure 5.10 show that the current validation has negligible effects on the timing properties of the control algorithm. Compare the figure with Figure 5.7.

Position validation, no fault

In this case only the position validation was enabled and no fault was simulated.

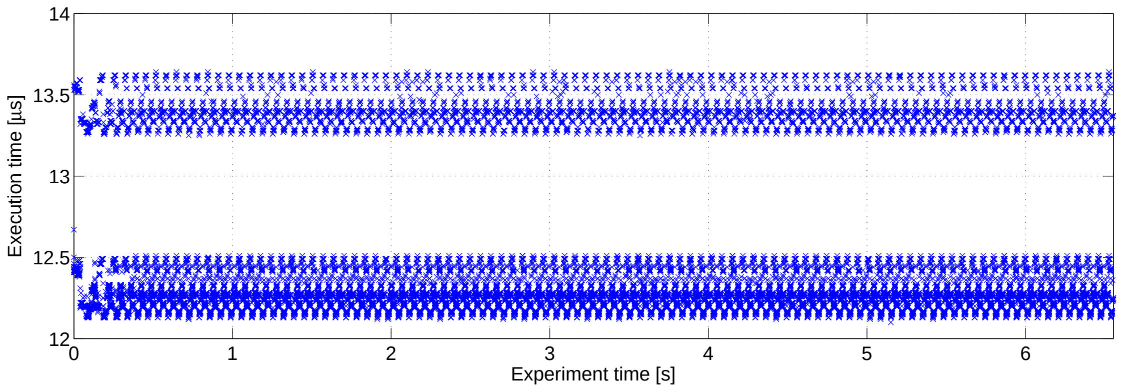


Figure 5.11.: Position validation, no fault

From results in Figure 5.11 can be seen that the position validation prolongs the execution of the control algorithm roughly by 1.5–2 µs and adds more jitter.

PWM diagnostic, no fault

In this case only the PWM diagnostic was enabled and no fault was simulated.

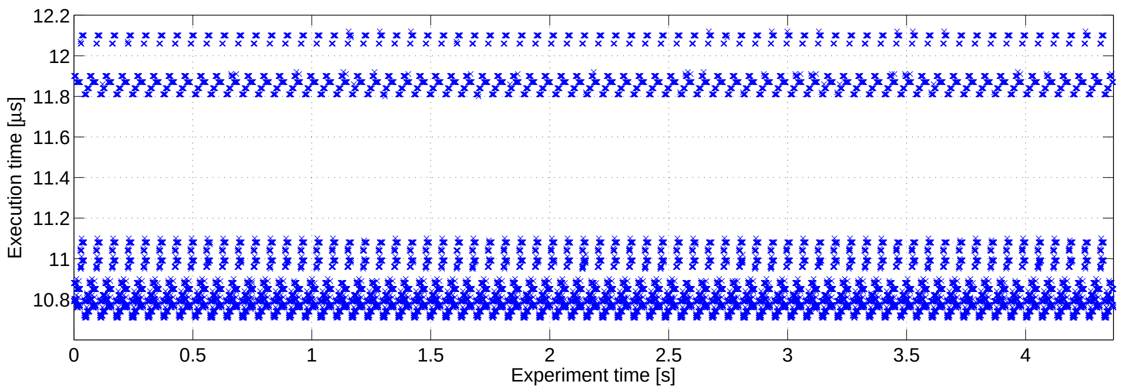


Figure 5.12.: PWM diagnostic, no fault

As can be seen from Figure 5.12 the PWM validation prolongs the execution of the control algorithm roughly by 0.5 µs and it makes every other execution take slightly more time, because it is implemented as a two step process – read back and validate – and only one step is performed per control algorithm execution alternating the executed code branch every time.

Memory validation, no fault

In this case only the memory validation was enabled and no fault was simulated.

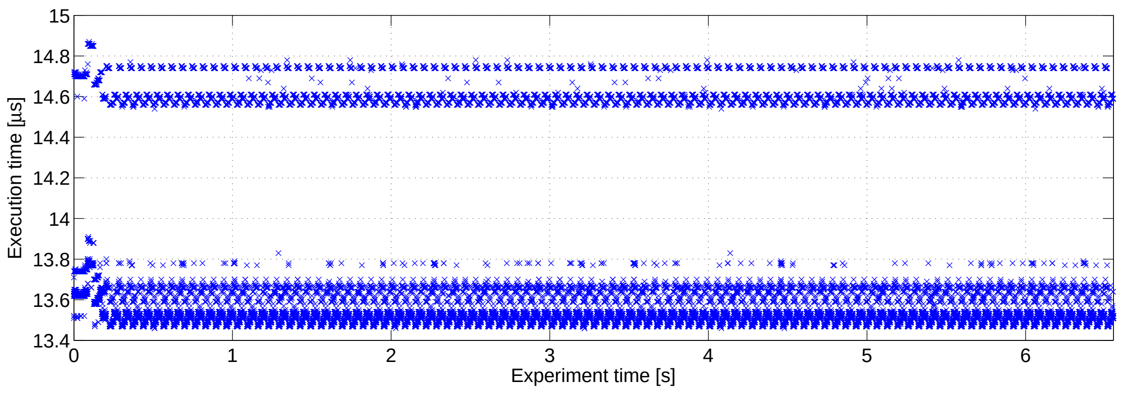
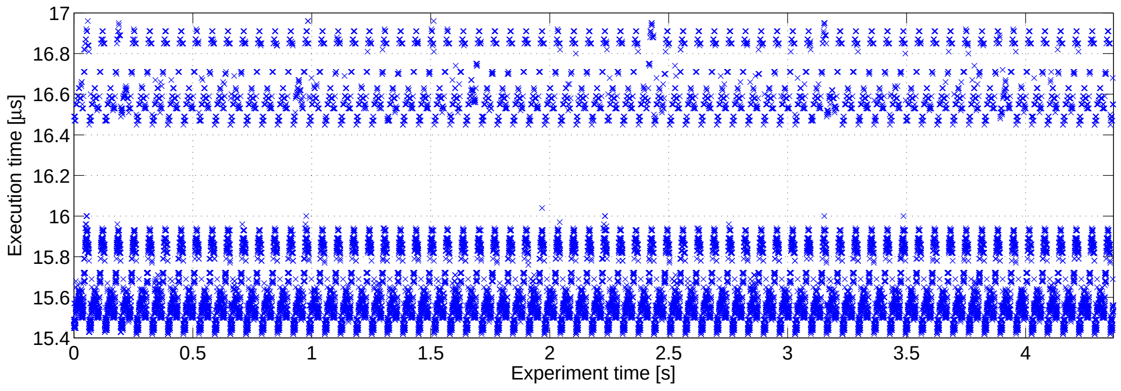
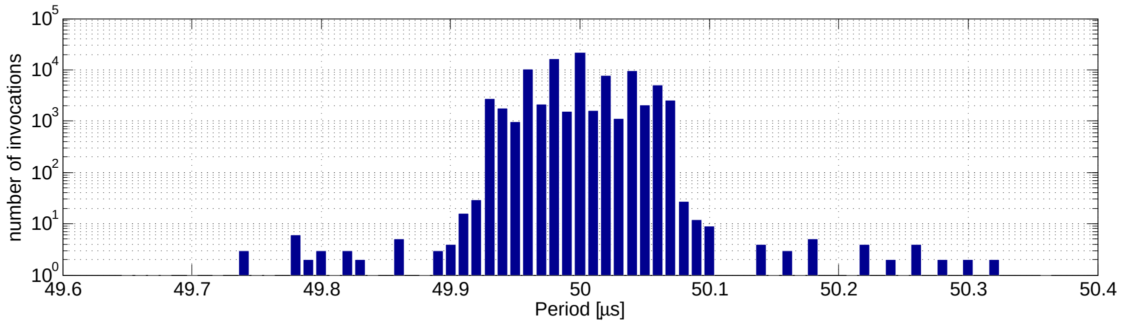


Figure 5.13.: Memory validation, no fault

Results in Figure 5.13 show that the memory validation prolongs the execution of the control algorithm roughly by 3 µs making it the most demanding safety measure.



(a) Time series



(b) Histogram of invocation period

Figure 5.14.: All safety measures, no fault

5.3.4. All safety measures enabled

Experiments in this section run with all safety measures enabled, as in a production system. The purpose is to see how they operate together and how do various faults influence the execution time.

No fault

In this case all four safety measures were enabled and no fault was simulated to measure the combined effects of all safety measure on the eMotor driver timing properties.

Results can be seen in Figure 5.14a. All safety measures combined prolong the execution of the control algorithm roughly by 5 µs and slightly prolong every other cycle due to the nature of PWM diagnostic (see Section 5.3.3).

Figure 5.15 shows a more detailed view of the same data and reveals a repeating pattern in execution time. Period of this pattern is roughly 65.7 ms. We have no explanation for why is the pattern there.

For this case we also made the period histogram – by period we mean the time interval between two consecutive invocations. The histogram can be seen in Figure 5.14b. It shows, that addition of all safety measures has some effect on how the invocation period

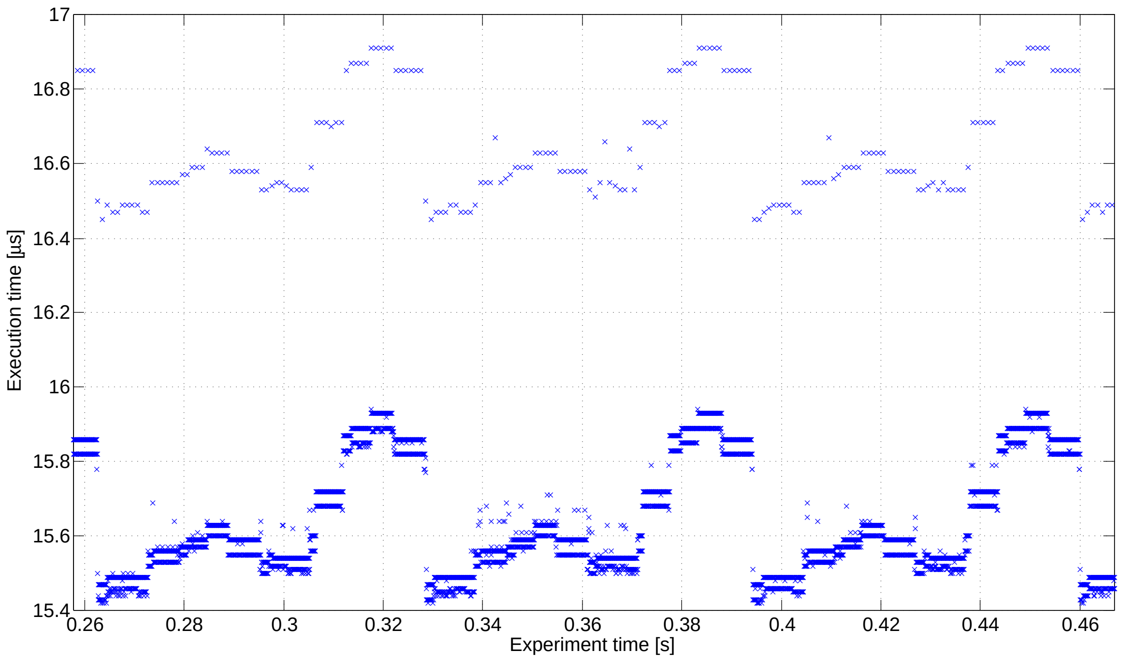


Figure 5.15.: All safety measures, no fault (detail)

is spread across the interval where most values are located but it does not significantly broaden it (compare with Figure 5.7c), so it has little to no adverse effect on the control algorithm invocation period.

Current phase A fault

In this case all four safety measures were enabled and phase A measurement fault was simulated – measured phase A current was set to 0 roughly 0.9 s after the start of the measurement.

As can be seen from results in Figure 5.16, the current fault was correctly reported – the interruption of the error report is caused by the actual current entering the insensitivity interval around 0 (see Section 5.2.3). The algorithm execution time is slightly prolonged when the error is reported solely due to the calls of the reporting function. Interesting fact is that unlike in SW-in-the-loop case (Section 5.2.3) the position error was not reported simultaneously with the phase A fault. One possible explanation is, that the added jitter in the measured value together with the correct information from phases B and C was enough to keep the validation motor model spinning, whereas the exact value used in SW-in-the-loop stopped it.

Current phase B fault

In this case all four safety measures were enabled and phase B measurement fault was simulated – measured phase B current was set to 0 roughly 0.9 s after the start of the

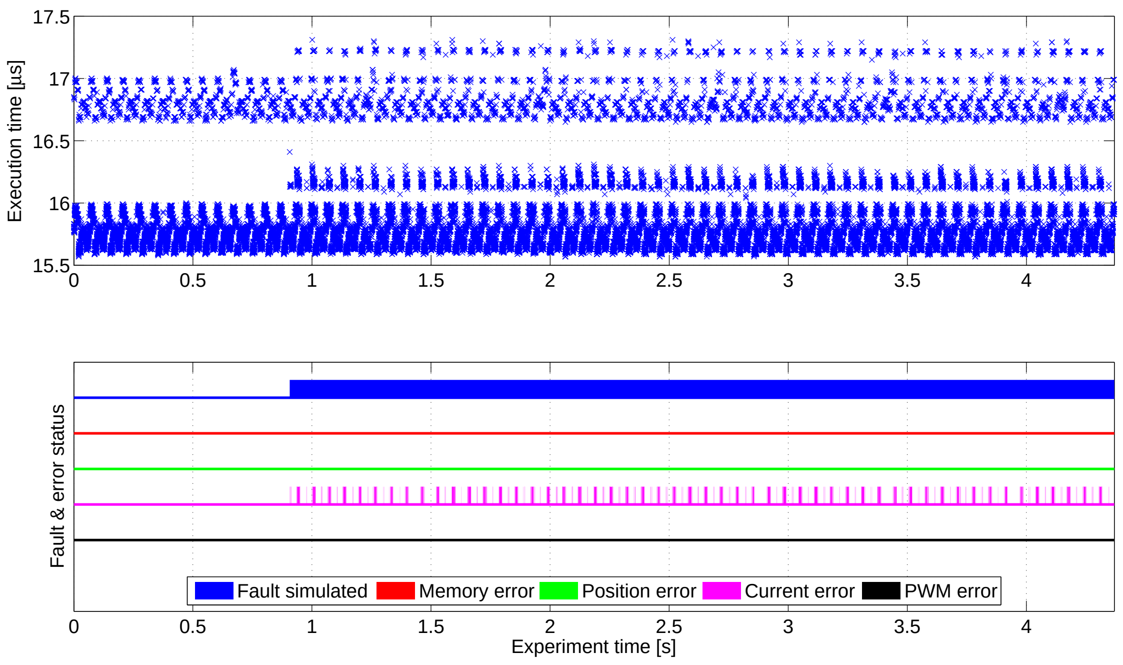


Figure 5.16.: All safety measures, phase A fault

measurement.

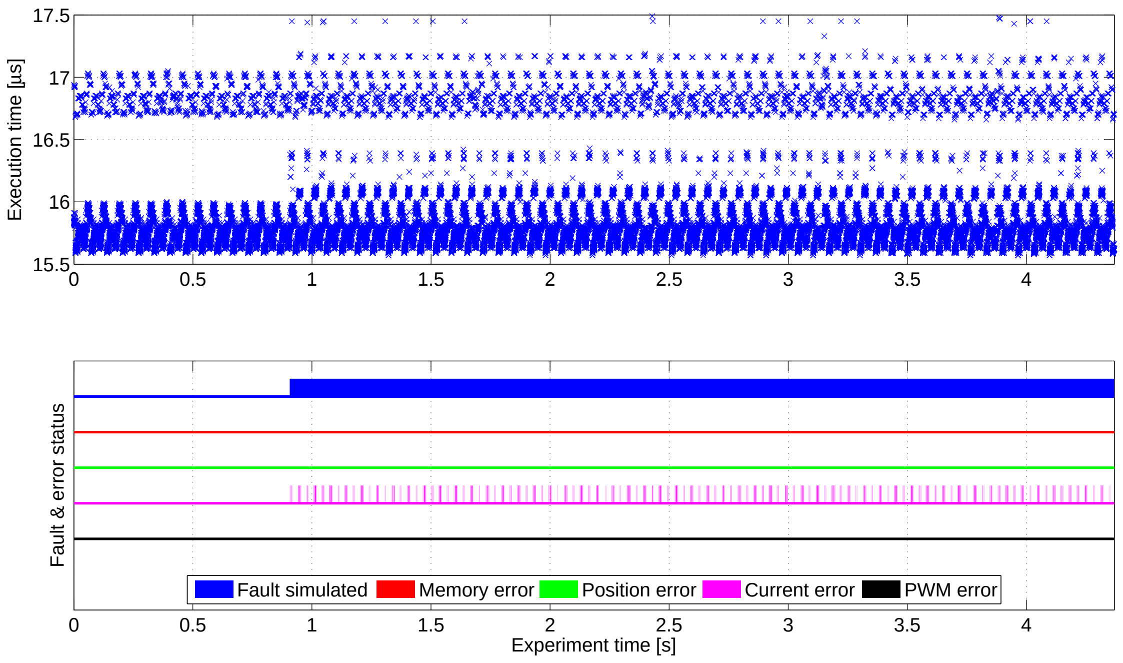
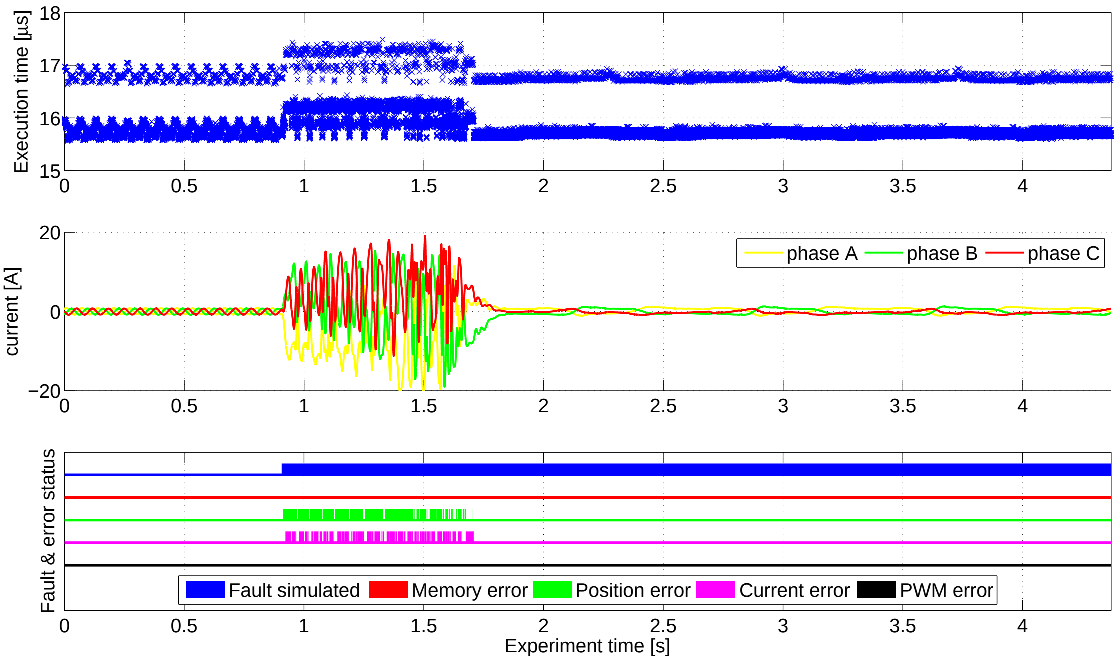


Figure 5.17.: All safety measures, phase B fault

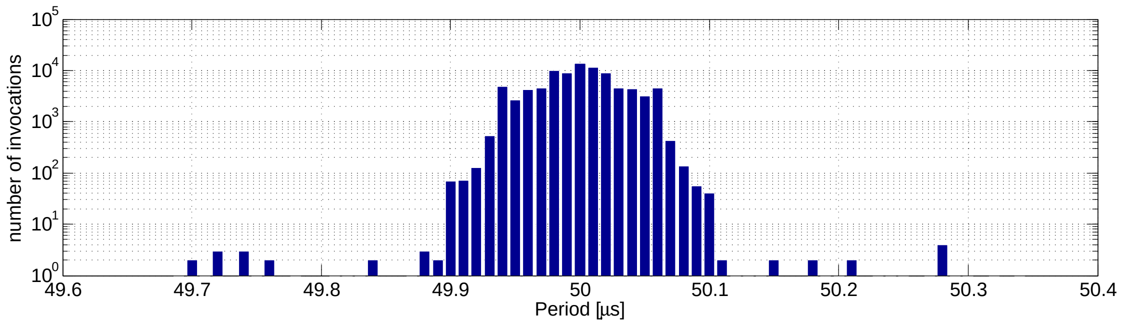
The results in Figure 5.17 are very similar to the previous experiment. This time it is no surprise that the position error was not reported because it corresponds to the results of SW-in-the-loop tests in Section 5.2.4.

Position fault

In this case all four safety measures were enabled and a position fault was simulated – the measured position was stuck on one value roughly at 0.9 s.



(a) Time series



(b) Histogram of invocation period

Figure 5.18.: All safety measures, position fault

As can be seen from results in Figure 5.18a, the position fault was correctly reported and a current error was reported simultaneously, which has the same cause as in the SW-in-the-loop test (see Section 5.2.2) – currents exceeding the range of used ADC, thus invalidating the measurements.

The unexplained change in the execution time is present just like in the case of position fault without any safety measures enabled (Section 5.3.2) and the subsequent controller wind-up is present as well.

The period histogram was made for this case as well. It can be seen in Figure 5.18b and it shows that the fault injection has made the period a little bit more erratic, but it still does not have significant adverse effect.

Position fault during calibration of position validation

In order for the position validation safety measure to work, an internal motor model has to be calibrated. The calibration starts simultaneously with the motor control and runs roughly 10 s. In this experiment a position fault was simulated during these 10 s.

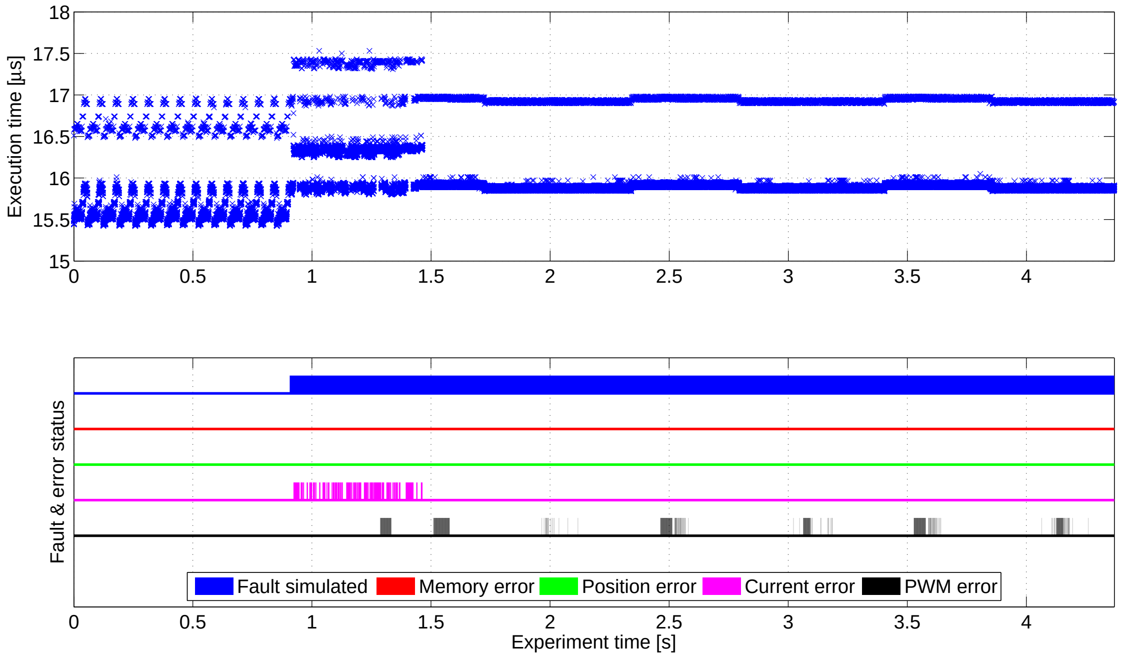


Figure 5.19.: All safety measures, undetected position fault

As can be seen from results in Figure 5.19, the position fault was not reported at all. The current error caused by the controller overreaction is present just like in previous cases with position faults and even sporadic PWM errors are detected, those are caused by noise present in the wiring, which got more pronounced with the abrupt changes in signal values caused by the fault.

5.3.5. CAN bus flooding

While all the previous experiments were merely safety-related, this section covers even security-related experiments. The main goal here is to determine how the additional security measures interfere with normal operation and whether they do or do not pose a safety problem.

In all experiments in this section except 5.3.5 the CAN bus was flooded by a continuous stream of messages with random ID, random length (0 – 8 bytes) and random data sent with the highest baud rate the TriBoard handles – 1 Mbps – to simulate an uniformed attack trying either to guess the used verification key or to simply overload the bus and disrupt normal operation. We are interested only in effects this has on the eMotor, the effectiveness of used verification algorithm is not of interest in this work.

First 3 tests measured effects of different received message handling the AUTOSAR CAN driver allows – polling or interrupt based – on the eMotor's execution. Based on these tests the polling method was selected as the only viable one and subsequent MaCAN test were carried using only this message handling method.

No safety measures, no fault, polling

In this experiment, the CAN driver was configured to receive data in the polling mode[1], i.e. reception of a CAN message generates no interrupt. The main loop of our TriBoard application does not read the messages out of the CAN controller. Therefore, this test is only used to see the effect of CAN controller activity on the CPU executing the eMotor driver.

Results in Figure 5.20a show that there is little effect on the CPU execution. Few invocations were about 0.1 µs longer that others, but this is negligible.

Invocation period was also measured, results are shown in Figure 5.20b and they show that this method of CAN message handling has no adverse effect on period of control algorithm invocations.

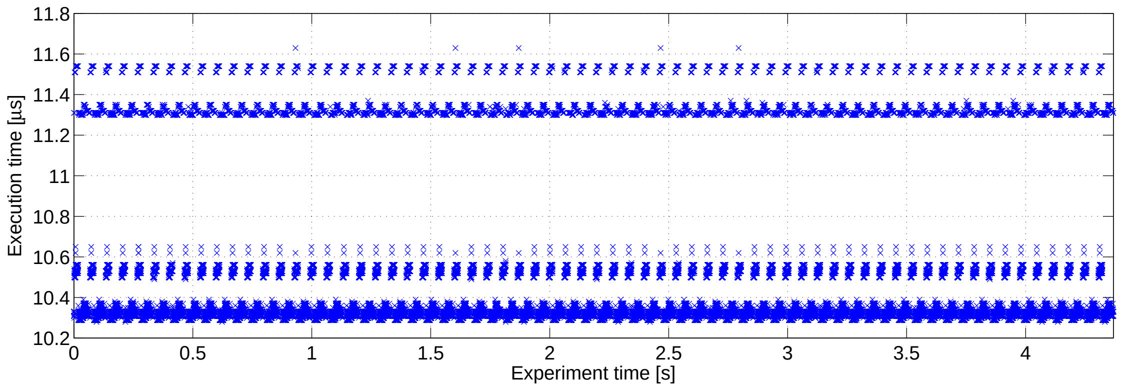
No safety measures, no fault, high priority interrupt

In this experiment, the CAN driver was configured to use interrupts to signal frame reception. However, our interrupt service routine just acknowledged that the received frame was handled and no further processing of the frame was carried out. The CAN receive interrupt priority was set higher than the priority of the ADC interrupt invoking the eMotor control algorithm.

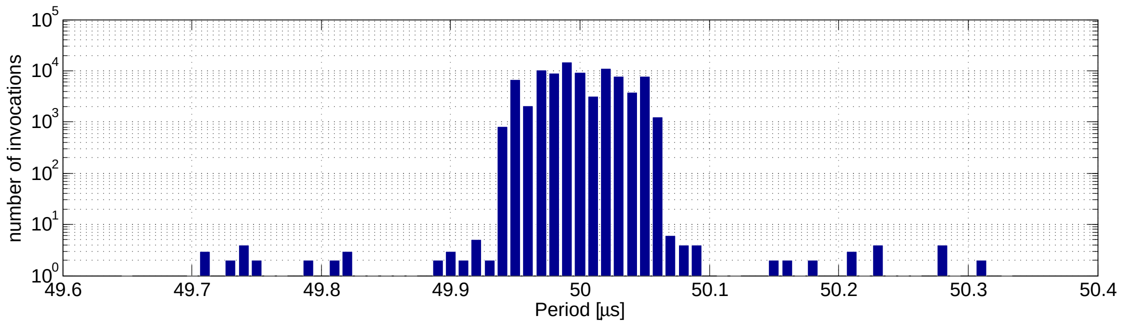
As can be seen from Figure 5.21a, this setting has significant impact on the control algorithm, some invocations are delayed up to 2 µs and jitter is significantly increased.

As for the effect on period of control algorithm invocations Figure 5.21b clearly shows that this method has significant adverse effect on it, same as on the execution time. The period is up to 2.25 µs off (compare to max 0.3 µs off in other cases) and the number of significantly delayed executions has also increased.

[1]This is common in automotive applications.

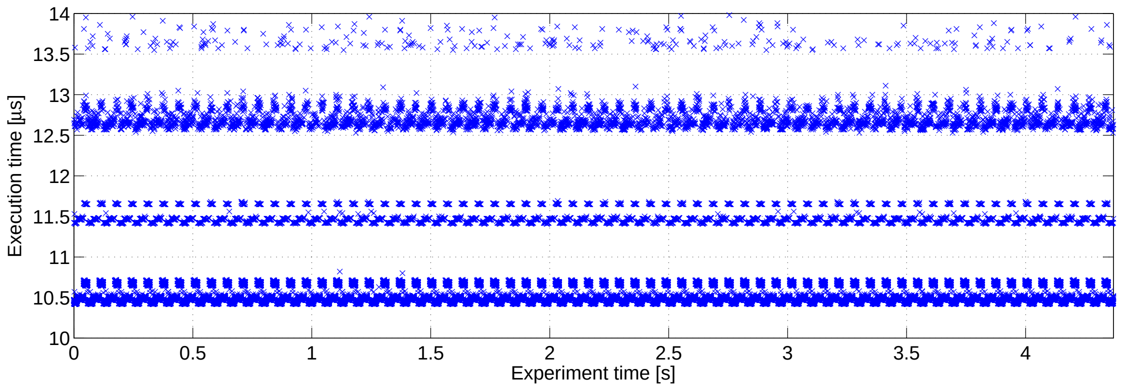


(a) Time series

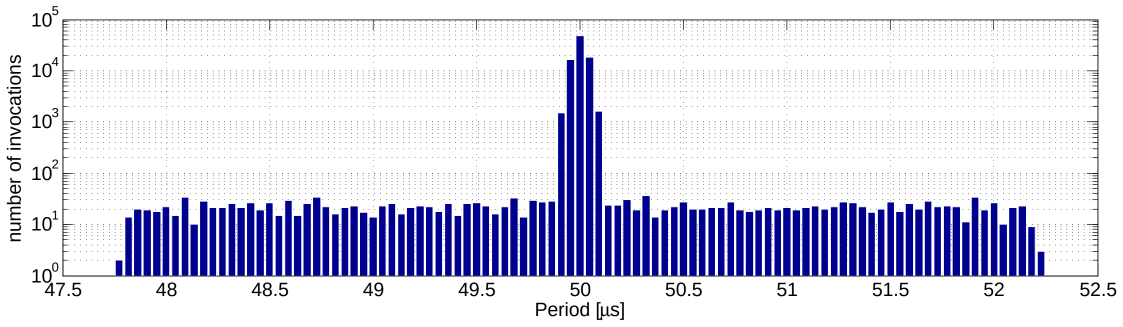


(b) Histogram of invocation period

Figure 5.20.: No safety measure, no fault, CAN flood, polling



(a) Time series

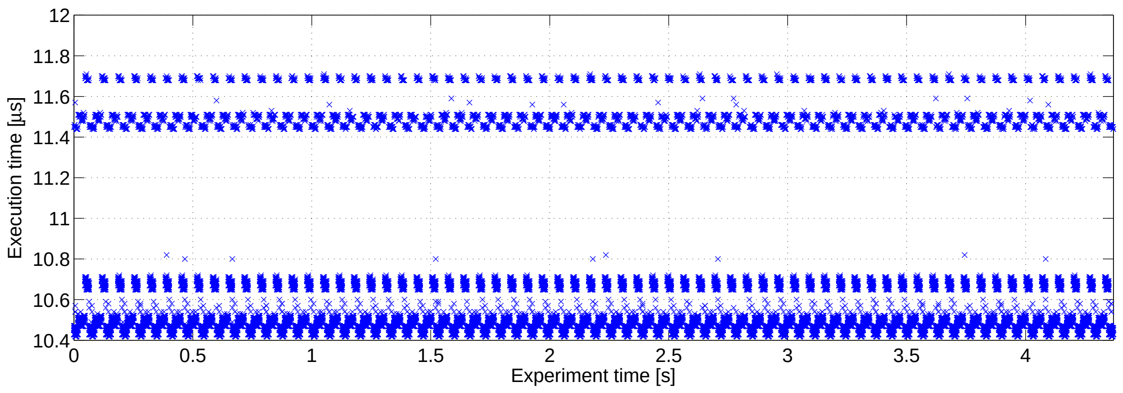


(b) Histogram of invocation period

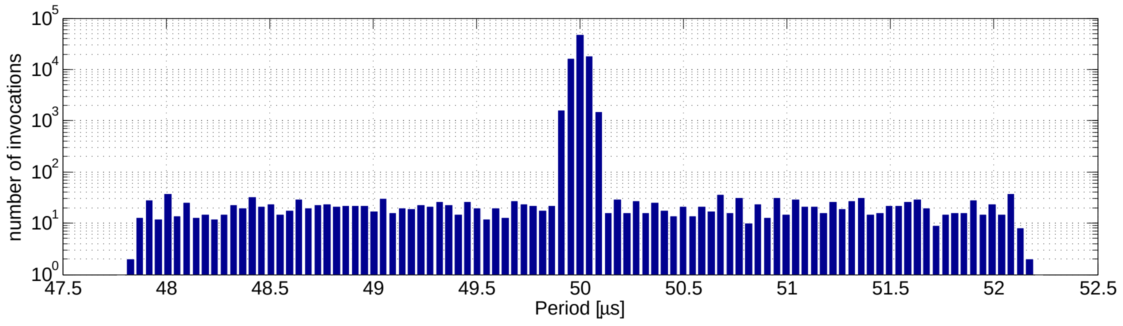
Figure 5.21.: No safety measure, no fault, CAN flood, high priority interrupt

No safety measures, no fault, low priority interrupt

In this experiment, the CAN driver was configured to use interrupts to signal frame reception. However, our interrupt service routine just acknowledged that the received frame was handled and no further processing of the frame was carried out. The CAN receive interrupts priority was set lower than the priority of the ADC interrupt invoking the control algorithm.



(a) Time series



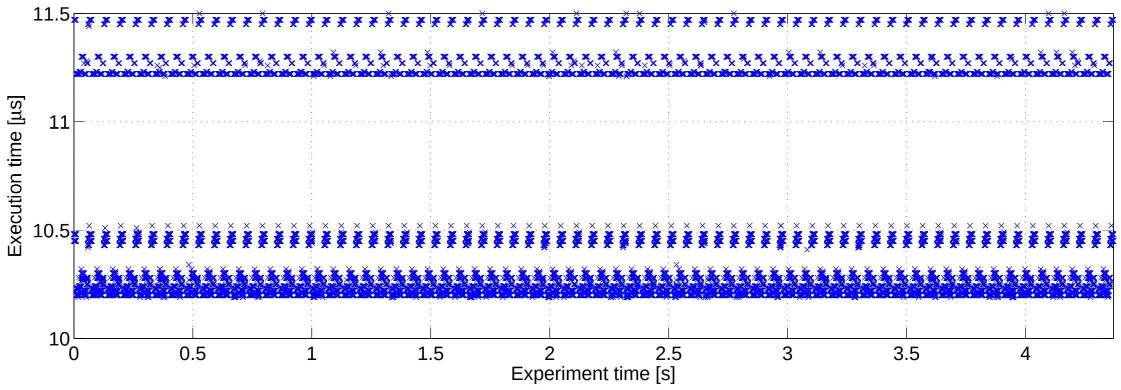
(b) Histogram of invocation period

Figure 5.22.: No safety measure, no fault, CAN flood, low priority interrupt

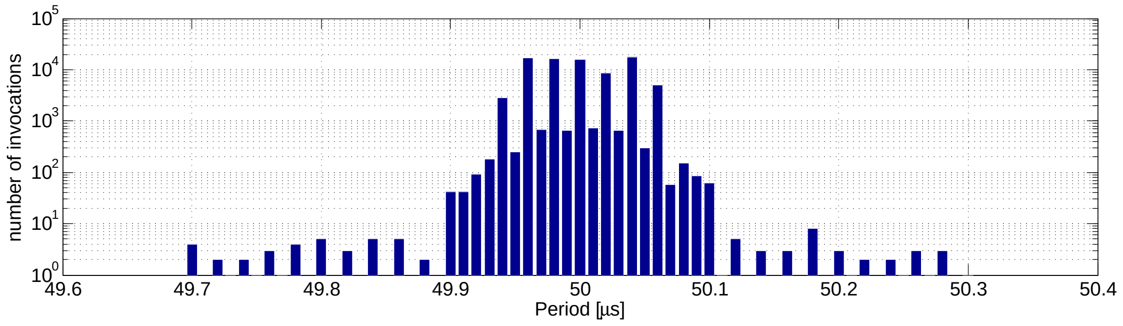
The results are shown in Figure 5.22a. It can be seen that this setting has negligible effects on the control algorithm execution time, just like the polling setting (Section 5.3.5), but it has significant adverse effects on the invocation period. This can be seen in Figure 5.22b, which clearly shows that this setting has the same effect on invocation period as the high priority interrupt setting (Case 5.3.5).

No safety measures, no fault, MaCAN control, no flood

In this case the reference speed was set remotely by sending messages to the TriBoard over message authenticated CAN (MaCAN protocol [5]). CAN message handling was set to polling, same as in Case 5.3.5 The reference speed was set to the same value as in all previous cases – 1000 rpm. All safety measures were disabled and CAN bus was not artificially flooded; only normal MaCAN traffic with time server, key serves and two nodes was present. No artificial faults were simulated.



(a) Time series



(b) Histogram of invocation period

Figure 5.23.: No safety measure, no fault, MaCAN control, no flood

As can be seen from results in Figures 5.23a and 5.23b, presence of MaCAN control has no adverse effects on the timing properties of the eMotor control algorithm.

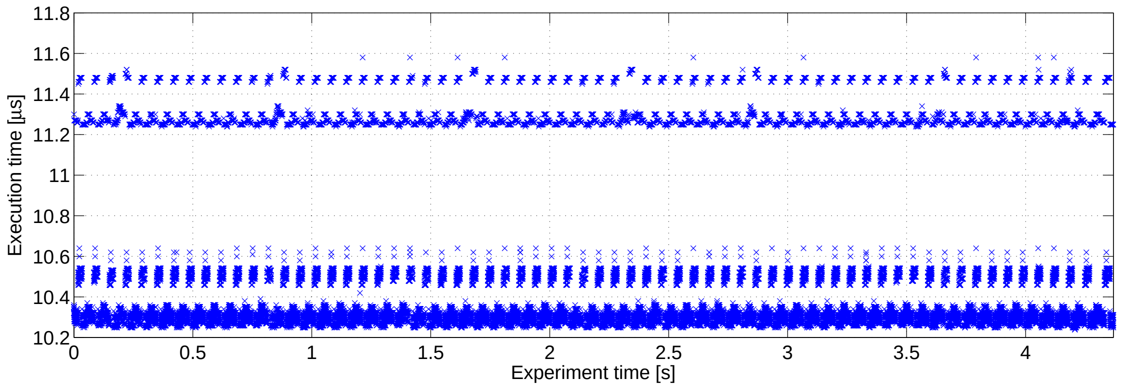
No safety measures, no fault, MaCAN control, CAN flood

In this case the reference speed was set over MaCAN and safety measures were disabled as in the previous case, but CAN bus was flooded with continuous stream of messages just like in Section 5.3.5. No artificial faults were simulated.

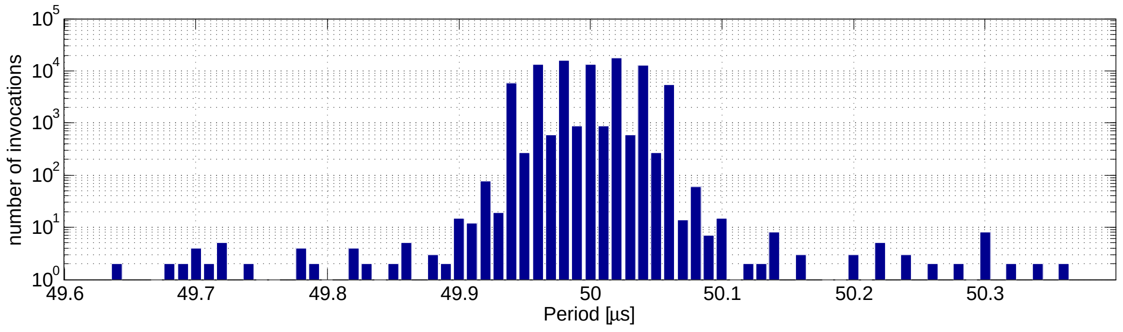
As can be seen from results in Figure 5.24a the increased load generated by validation of those random messages has some effect on the execution time, presumably caused by cache trashing, but the effects are minor. And Figure 5.24b shows that it has no effect on invocation period.

All safety measures, no fault, MaCAN control, CAN flood

In this case the reference speed was set over MaCAN to be 1000 rpm as in the previous cases. All safety measures were enabled and CAN bus was flooded with continuous



(a) Time series



(b) Histogram of invocation period

Figure 5.24.: No safety measure, no fault, MaCAN control, CAN flood

stream of messages just like in Section 5.3.5. No artificial faults were simulated.

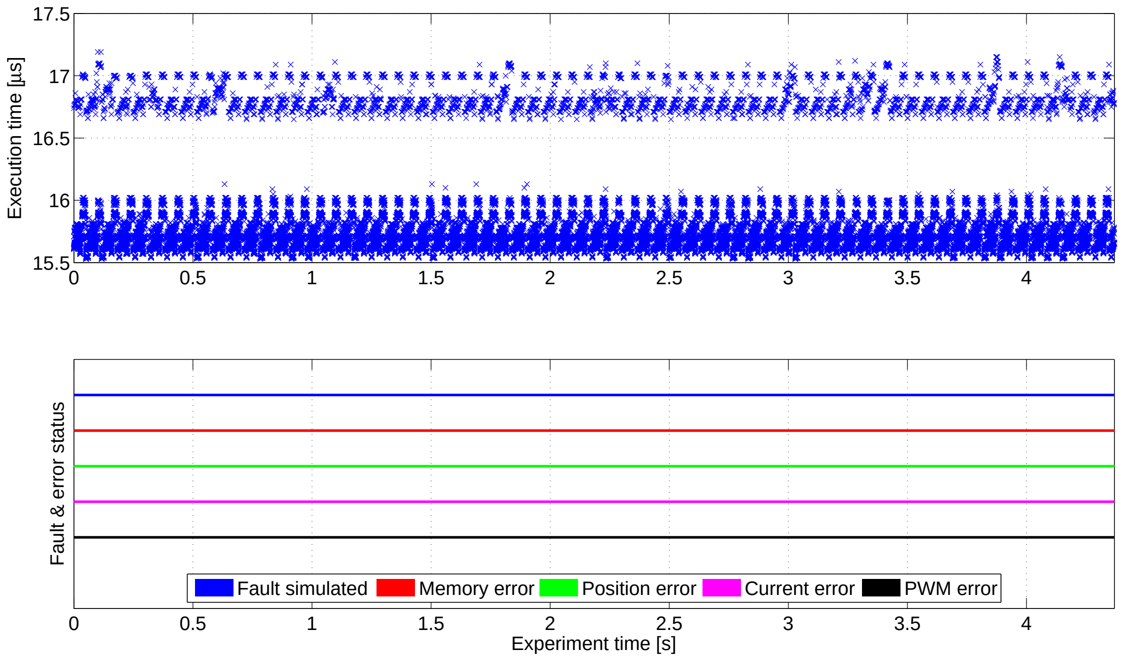


Figure 5.25.: All safety measures enabled, no fault, MaCAN control, CAN flood

As can be seen from results in Figure 5.25 the increased load generated by validation of those random messages has some effect on the execution time, just like in previous case, but the effects are minor and no safety measure reports a false error.

6. Conclusion

In this work we presented the developed testbeds for safety and security testing and demonstrated their functionality by testing the eMotor driver prototype. The safety measures in this version of the eMotor software were just a prototype implementation which limited the complexity of the tests. Nevertheless our testing revealed some deficiencies that should be addressed in the final implementation. In short, those were:

- The state of eMotor's internal motor model used for position validation sometimes diverges from the state of the controlled motor, causing false errors to be reported. See Section 5.2.1 for details.
- Correctness of position validation depends on correctness of current measurement. See Section 5.2.4. This may lead to various failures at system level if one expects these safety measures to be independent.
- Unexplained change in execution time after longer interval with active position reading fault. See Figure 5.9 and Section 5.3.2 for details. This is not a problem in itself, because the execution time stays in the expected range, but it may signal a presence of an currently unknown problem.
- Documentation does not say that even lower-priority interrupts can delay the execution of high-priority ADC interrupt used by eMotor. This can be clearly seen by comparing Figure 5.22b with Figures 5.21b and 5.20b. If this blocking is not bounded it can lead to various failures.

We also integrated the eMotor driver with an implementation of message authenticated protocol on CAN bus called MaCAN. Our implementation of MaCAN uses the Secure Hardware Extension (SHE) of the TriCore TC1798 CPU to accelerate cryptographic operations needed for its function. We conducted several experiments with this protocol to see the effect of envisioned attacks over CAN networks on the eMotor functionality. Our results show that such attacks have no significant influence on the eMotor functionality.

Based on the experience from this work, we argue that there is a big benefit in joining safety and security activities in the testing and validation phase of the development process. Development of the software- and hardware-in-the-loop testbeds consumes a lot of resources. Since these testbeds can be used for validations of both safety and security properties, there is little benefit of having two teams developing the same test bed.

Bibliography

[1] International Organization for Standardization / Technical Committee 22 (ISO/TC 22), "ISO/DIS 26262-1:2010(e) – Road vehicles – Functional safety — Part 1 Glossary", Tech. Rep., 2010.

[2] ——, "ISO/DIS 26262:2010(e) – Road vehicles – Functional safety", Tech. Rep., 2010.

[3] ISO/IEC JTC 1/SC 27 IT Security techniques, "ISO/IEC 15408:2009 – Information technology – Security techniques – Evaluation criteria for IT security", Tech. Rep., 2008/2009.

[4] R. Isermann, J. Schaffnit, and S. Sinsel, "Hardware-in-the-loop simulation for the design and testing of engine-control systems", *Control Engineering Practice*, vol. 7, no. 5, pp. 643–653, 1999, ISSN: 0967-0661. [Online]. Available: `http://www.sciencedirect.com/science/article/pii/S0967066198002056`.

[5] O. Hartkopp and R. Schilling, "MaCAN – message authenticated CAN", in *ESCAR Conference*, Berlin, Germany, Nov. 2012.

[6] AUTOSAR, *Specification of the virtual functional bus*, R3.1 rev 5, 2010. [Online]. Available: `http://www.autosar.org/download/R3.1/AUTOSAR_SWS_VFB.pdf`.

[7] Infineon Technologies AG, *MC-ISAR_AUDO_UM_EmoDriver Documentation*, release V1.3, 2012.

[8] ——, *TC1798 32-bit microcontroller – Data Sheet*, V1.0, 2012.

[9] P. Pillay and R. Krishnan, "Modelling of permanent magnet motor drives", *IEEE transactions on industrial electronics, 537-541*, 1988.

[10] K. Belda, "Study of predictive control for permanent magnet synchronous motor drives", in *Methods and Models in Automation and Robotics (MMAR), 2012 17th International Conference on*, Aug. 2012, pp. 522–527.

[11] Mathworks, inc., *MATLAB Documentation*, R2012b, 2012.

[12] DCE FEE CTU, *Linux Target for Simulink Embedded Coder project.* [Online]. Available: `http://lintarget.sourceforge.net/`.

[13] HUMUSOFT s.r.o, *MF 624 Multifunction I/O card User's manual*, 2006. [Online]. Available: `http://www2.humusoft.cz/www/datacq/manuals/mf624um.pdf`.

[14] AUTOSAR, *Specification of can driver*, R3.0 rev 2, 2008. [Online]. Available: `http://www.autosar.org/download/AUTOSAR_SWS_CAN_Driver.pdf`.

[15] O. Hartkopp and Volkswagen, AG, "The can networking subsystem of the linux kernel", *Proceedings of the 13th iCC*, 2012.

A. Contents of the enclosed CD

dp 2014 krec michal.pdf – electronic version of this document

eMotor_testbed.slx – software-in-the-loop testbed developed in this work. Unfortunately the NDA agreement with Infineon does not allow us to publish any part of eMotor software, so the binary application for the eMotor S-function can not be enclosed thus the model is not functional.

HW_in_the_loop.slx – the model used as a part of hardware-in-the-loop testbed.

mf624_SIMULINK.c,mf624_SIMULINK.h – source files, that contain code needed to operate the MF624 I/O card form a Simulink S-function.

sfAnalogInput.c,sfAnalogOutput.c,sfDigitalInput.c,sfDigitalOutput.c,sfReadPWM.c – source files, that contain code for S-functions described in Section 4.2.2

List of Figures