16th IMEKO TC10 Conference
"Testing, Diagnostics & Inspection as a comprehensive value chain for Quality & Safety
Berlin, Germany, on September 3-4, 2019

RTOS aware non-intrusive testing of cyber-physical systems in HIL (Hardware In the Loop) environment

Balázs Scherer

Department of Measurement and Information Systems
Budapest University of Technology and Economics
Budapest, Hungary
scherer@mit.bme.hu

***Abstract* – Statistics show that more and more cyber-physical systems are using RTOS (Real-Time Operating System). RTOS based system software can introduce multitasking and real-time behaviour based errors. Therefore, the testing processes of such systems should address the detection of these possible errors. Unfortunately, the multitasking and real-time behaviour based errors are among the hardest to detect. The best way for the detection is to perform extensive testing in a very realistic environment like in a HIL (Hardware In the Loop) simulation. These environments provide the possibility to perform overloaded event simulation, that increase the chance of multitasking and real-time behaviour based error occurrence. Traditionally the RTOS aware measurement methods (providing information for the error detections) use software instrumentation, and are not integrated into HIL test environments. This paper introduces a novel integration of RTOS aware measurements into a HIL test development environment. Our integration also focuses on the non-intrusive measurements of multitasking behaviour of RTOS based software systems. These non-intrusive measurements are not widespread and their integration into a HIL based environment is also a novel solution.**

Keywords – Testing, HIL (Hardware In the Loop) tests, RTOS (Real-Time Operating System) aware testing, Non-intrusive software measurements.

I. INTRODUCTION

Statistics show that over two thirds of embedded systems use RTOS (Real-Time Operating System) [1]. The type and complexity of these RTOS versions are very divers, starting with Embedded Linux versions running on high performance application processors down to simple preemptive multitasking kernels like (FreeRTOS and uC/OS) running on small microcontrollers of typical cyber-physical systems. Our work will focus on the microcontroller based systems using simple preemptive multitasking kernels.

A. Typical testing environment of microcontroller based cyber-physical systems

One of the typical test method of microcontroller based cyber-physical systems is the HIL (Hardware in the Loop) test, where the behaviour of the integrated software and hardware of the DUT (Device Under Test) can be investigated in a simulated and stable environment. These tests are originated and widespread in the automotive and transportation industry, but using them for general purpose cyber-physical systems is getting more and more widespread [2], [3]. A typical HIL test setup is shown on Fig. 1.

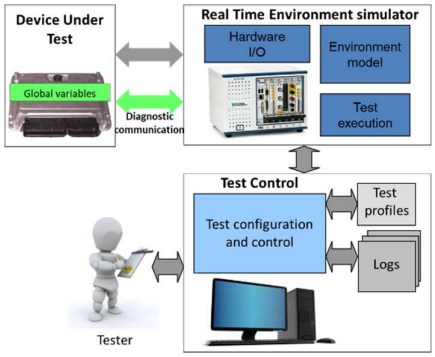


Fig. 1. Typical HIL test setup

HIL tests can reduce the time spent on real environment testing, because most of the failures can be detected earlier in the simulated environment. This can reduce the overall testing cost significantly, because real environment setups like prepared track with staff for autonomous robots, or real plant setups for industrial

controllers are very costly. HIL tests also have the advantage of repeatability, controllability and stability, which is usually not given in the real environment.

B. Challenges of detecting multitasking and real-time behaviour based errors

Cyber-physical systems using RTOS based software introduce new challenges for the testers. The failure model of such a real-time embedded system [4] can be divided into sequential and multitasking real-time behaviour based failures shown on Fig. 2.

Industrial experiences have shown that the test methods for detecting sequential failures are already elaborated. Traditional white box and HIL environment based tests catch these types of failures. The detection of multitasking and real-time failures is a much harder problem. Most of the traditional tests do not have sophisticated methods to do that. Therefore, the multitasking and real-time failure detection capabilities of test systems should be improved.

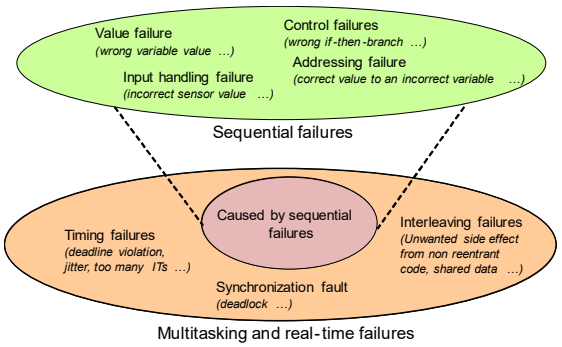


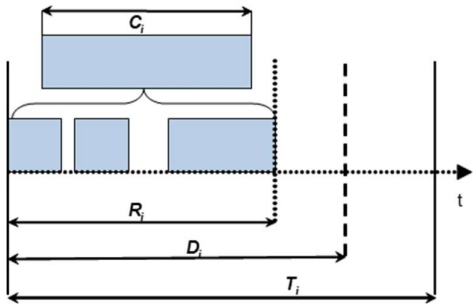
Fig. 2. Software failure models

There are three categories of multitasking and real-time failures: Timing failures (deadline violations of given tasks, unwanted jitter of periodically executed control functionality, too many and too long interrupts etc.), Synchronization failures (deadlock of tasks waiting for the same resource) and Interleaving failures (Unwanted side effect from non-reentrant code or shared data).

In our work we will focus on the Timing and Synchronization problems, because traditional tests leave these rather uncovered. Another reason to focus on these types of failures is that these can be measured well in HIL testing environments.

C. Methods of Timing and Synchronization failure detection

The most foundational thing of Timing and Synchronization failure detection is the static prediction or runtime measurement of software task WCETs (Worst Case Execution Time). These execution time predictions or measurements can be used to calculate the Worst Case Response time for each task, and therefore analyse whether the system will be able to behave in real-time in every situation by keeping the schedule of the tasks. Real-time systems out of their schedule can show symptoms like unwanted resets and strange behaviour for a short time. Fig. 3. introduce the notation and definitions of algorithms like Deadline Monotonic Analysis [5] used for Worst Case Response time calculation.



Ti is the period of task ***i***
Di is the deadline of task ***i***
Ci is the worst-case execution time of task ***i***
Ri is the worst-case response time to task ***i***

Fig. 3. Notations and definitions of task timing parameters

The static calculation of execution times and WCET is based on the pure source code of the tasks. The predictor calculates the task executional time for a given architecture by analysing the compiled assembly code without executing it, or by executing it in an emulator. Either method is used, the prediction of task execution times is a very complex problem, and many articles discuss its challenges [6].

Summary, there are tools for creating these static calculations, but their precision cannot reach the precisions of measurements made in HIL simulations or in real environments. Therefore, our paper focus on measurements made in HIL tests or real environments.

II. THEORETICAL BACKGROUND OF MEASURING TASK EXECUTION TIME

The value of ***Ti*** and ***Di*** used for Worst Case Response time calculation can be derived from requirements. While the value of ***Ci*** is need to be measured.

A. RTOS instrumentation based measurements

In some RTOS the built in instrumentation can provide the value of ***Ci***, but this requires a built in low level kernel measurement code. For example, FreeRTOS, which is currently the most widespread low weight preemptive multitasking embedded RTOS [1], [7] has the so called Trace hooks. These Trace hooks enables the user to perform measurements about the internal behaviour of the RTOS.

Trace hooks like traceTASK_SWITCHED_OUT (called when the RTOS switch out an old task), and

traceTASK_SWITCHED_IN (called when the RTOS switch in a new task) can be used to measure the ***Ci*** value of tasks.

There are professional solutions using these instrumentation capabilities: FreeRTOS+Trace is a run-time diagnostic and optimization tool provided by FreeRTOS's partner company Percepio. FreeRTOS+Trace captures valuable dynamic behaviour information, then presents the captured information in interconnected graphical views.

FreeRTOS+Trace is a very useful tool, but it is not integrated into a HIL testing environment. Therefore, its timing displays, are not synchronised with the HIL simulators stimulus signals and measurements, which could cause problems during testing.

RTOS instrumentation used by FreeRTOS+Trace also have the disadvantage, that for measuring the ***Ci*** values many data communication is required between the RTOS and the measuring software, which can cause significant overhead to the cyber-physical system.

RTOS instrumentation also have to deal somehow with the problems caused by interrupts. The interrupts can modify the measurements of ***Ci*** values unless the interrupt execution is not instrumented too, but instrumenting the interrupt execution can increase the overhead of the measurement significantly.

B. Non-intrusive measurements of task execution times

Most of the modern microcontrollers provide ways for high data rate non-intrusive data and instruction tracing. These tracing interfaces can be used to measure ***Ci*** values of tasks and interrupt timing too.

A typical example for such embedded trace support is the ARM's CoreSight trace architecture. The trace information in the CoreSight architecture is usually generated from three trace sources: Embedded Trace Macrocell (ETM), the ITM (Instrumentation Trace Macrocell), and the Data Watchpoint and Trace (DWT) blocks, shown on Fig. 4.

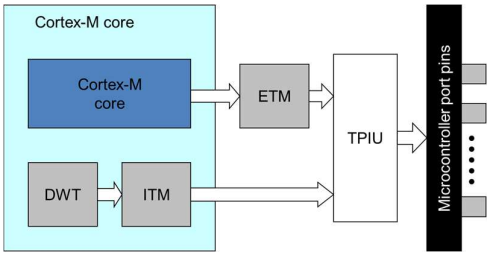


Fig. 4. ARM's CoreSight trace system in an ARM Cortex M core based microcontroller

The Trace Port Interface Unit (TPIU), formats the information form these sources into packets, and sends it to an external trace capture device.

The ITM has a capability to provide a "printf" style console message interface to the application software. The second purpose of the ITM block is to add timestamps to the DWT packets. The ETM block is used for providing instruction traces, therefore the whole software execution can be investigated. The drawback of using ETM is that it requires a huge amount of data transfer, and capturing and processing this amount of data is a challenging task [8].

From the task execution time (***Ci)*** measurement's point of view, the DWT block is the most interesting. The DWT has many of functionalities: among other things it can provide PC sampling at regular intervals and interrupt events trace. The DWT also includes several counters for measuring statistical parameters like interrupt overhead and sleep cycles. The DWT also has several comparators that can be used for data tracing on read or write accesses, or for triggering the ETM block. Summary the DWT outputs can be used for profiling and timing verifications.

The interrupt event trace function of DWT can be used to measure the interrupt execution with precise timestamps, so the interrupt overheads can be removed from the task execution times without software instrumentation. Statistics can also be created from interrupt occurrences and execution times, that can be used in schedule ability calculation.

The comparators of the DWT block also can provide useful help for execution time measurement. Every RTOS has a global variable, where the currently running task's property (usually a pointer to its task control block or TCB) is stored [7], [10]. By using the DWT's watch points, it is possible to follow every changes done to this variable, and the ITM block also can add timestamps to these events. This means that together with the interrupt tracing (every RTOS use software interrupts for task context switching) it is possible to measure the exact timing of tasks switching in and out with the information of which is the new task currently switched in. These tracing provides the same capabilities as software instrumented trace hooks, but without causing an overhead.

III. IMPLEMENTING THE NON-INTRUSIVE TASK EXECUTION TIME MEASUREMENT

To demonstrate the usability of the non-intrusive execution time measurement concept an experimental system was created. The unit under test is a Silicon Labs STK3700 development board containing an ARM Cortex-M3 based microcontroller, and running a FreeRTOS based multitasked demonstration software. The ARM Cortex-M microcontroller series is selected, because currently that is the most widespread 32-bit microcontroller core [1].

National Instruments LabVIEW development environment is used to perform the measurements. NI LabVIEW is selected because its popularity, and also because it can provide a straight forward integration to the NI-VeriStand HIL test development environment.

A. Interfacing to the trace port

The Trace Port Interface Unit (TPIU) supports two output modes, a synchronous double data rate clocked mode, using a parallel data output port with clock speed equals to the half of the system's core clock speed, and with a port width up to 4-bits (in the case of Cortex-M3 or M4 core microcontrollers) and a SWV (Serial Wire Viewer) mode, using a single-bit UART-like output.

The synchronous parallel interface has a high data rate, and it can be used for any trace measurements, including instruction trace. But, this high data rate port, which speed can easily reach the 100 Mbyte/sec requires a special FPGA based interface hardware. Such FPGA based hardware is costly and hard to integrate into a HIL environment [8].

The SWV's UART-like output is easily to interface directly through a serial port or through USB as virtual com port. The drawback of this solution is the possible message overflow because of the limited data rate. Usually the trace message outputs of the DWT blocks are short and not too frequent and therefore the DWT trace requires moderate or low data rate. An UART interface with its 1-2 Mbaud/sec usual maximum data rate can serve the average need of a normal DWT trace configuration. Unfortunately, the DWT message outputs are not to periodic, because event bursts can happen. Typical such event burst in our example is when an external interrupt happens, that cause a task activation. In this case the external interrupt sends at least 2 messages (entering and exiting), the task activation also done using a software interrupt (also 2 messages), and context switch will cause modification to the CurrentTCB global variable, which trigger another trace message. This means at least 5 trace messages with timestamps in a very short time interval (micro seconds range). However, these above trace messages are short messages, but such event burst can cause problems. The problem can be solved by integrated trace buffers specified in the CoreSight architecture, but the size of these buffers are implementation, and therefore microcontroller dependent. My measurement results show that the SWV interface can be used for RTOS aware non-intrusive testing, if the SWV data rate is higher than the 1/10 of the microcontroller's clock rate.

The implementation of the trace port interface is done using LabVIEW's OOP (Object Oriented Programing), which makes the implementation of trace interface extensible. Currently only a SWV based interface is created, but in the future the program is easily updateable with a synchronous high speed interface too.

B. Configuring the trace measurement

Configuring the trace measurement means the setting of the trace configuration registers [11]. This can be done inside the microcontroller, during the initialization phase, but that would cause software overhead and unnecessary trace output in every circumstances. More preferably this configuration also can be done through the debug port any time [3]. In this case the developers do not have to prepare the embedded software for the measurement, the test system will do this configuration.

What need to be set is the formatting and data port mode of the TPIU block (SWV wire output or parallel output). For RTOS timing measurement the DWT should be configured to have a comparator to the RTOS's Current Task Control Block global variable, and send messages if this variable is written by the microcontroller. To do so, the address of the Current Task Control Block global variable need to be known. Because this is a global variable, its address can be gathered from the .map or .elf file after building and linking the program.

The DWT block also needs to be configured to send trace messages on interrupt events.

Finally, the ITM block needs to be configured to add timestamps to the DWT messages. This timestamps can be local (relative time to the previous timestamp), or global timestamps. Global timestamps are more preferred for many reasons, but some microcontroller can only provide local timestamps.

C. Identifying the tasks of the system

The tasks running on the cyber-physical system can be directly specified by the operators before the test, or the test system can identify the software tasks during the test. For this identification the test system needs to know the type of the RTOS running on the cyber-physical system, and it also needs to know the current configuration setup of this RTOS.

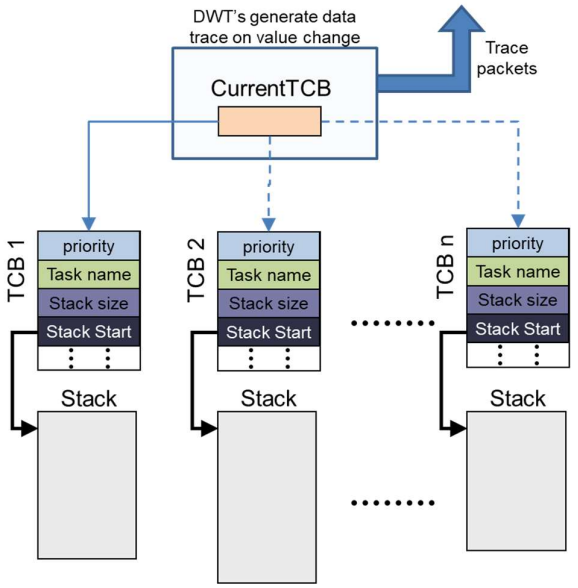


Fig. 5. Task identification through value change trace messages

Knowing the type of the RTOS and its configuration is important, because however the TCBs (Task Control Blocks) contain the same basic information regardless of RTOS type: task priority, task name, pointer to the task's

stack start, and last stacked item. But the structure of the TCB is RTOS dependent, and the TCB also can have configuration dependent values, like run time statistic counters, task tags for task identification etc.

After the test system knows the structure of the TCBs, then it can read the TCB information out though the debug port from the cyber-physical system for each task it switches on (Fig.5.).

D. Measurement results

Many measurements have been made to check the capabilities of the RTOS aware non-intrusive test system. The first measurement layer is simply an event activation graph with timestamps using LabVIEW's Digital Waveform Graphs. An example for this event activation graph is shown on Fig. 6. The figure shows an event timing capture of a sample system running two tasks: Low Priority Task (Low P. T.) and High Priority Task (High P.T.). The diagram presents the time interval, where the High Priority Task preempts the Lower Priority Task. It is also visible, that the Idle Task of FreeRTOS is running if both user task is waiting for an event. The interrupt overheads are also clearly visible: SysTick IT is the heart beat timer interrupt of the FreeRTOS system, and it runs in every 1ms, the PendSV IT (Pendable Service Call) is a software interrupt, and it is used by the FreeRTOS to perform task switching.

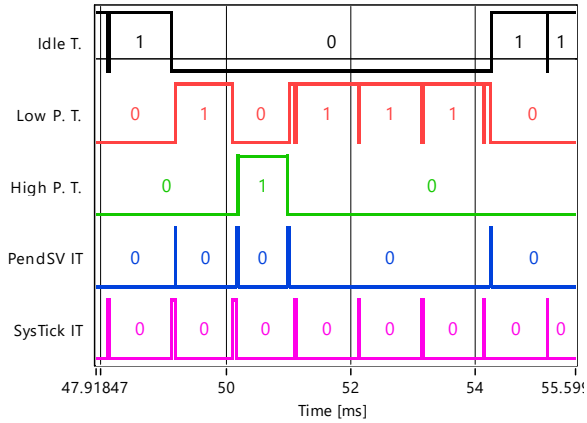


Fig. 6. Sample event activation view of a multitasked system

Based on the event activation measurements, higher abstraction level statistics and calculations can be created. From the Worst Case Response Time estimation's point of view, it is very important to have statistics for measured periods and execution times for every interrupts, and measured statistics of tasks execution times are also important (task periodicity should come from the specification). Fig. 7. shows an example of the measured parameters of the SysTick IT. SysTick IT is the heart beat timer of the FreeRTOS in this configuration, and therefore one of the most frequent interrupts. As the figure shows the periodicity of the SysTick IT is very stable as it can be expected from a high priority timer interrupt routine. The execution time results show a higher variability. The execution time depends on the functions needs to be performed during the interrupt. For example, Fig. 6. shows that the execution time increases, when the SysTick IT is executed before a task context switch. The reason for this, is that the SysTick IT notices that a task delay is elapsed, and then make that task ready to run.

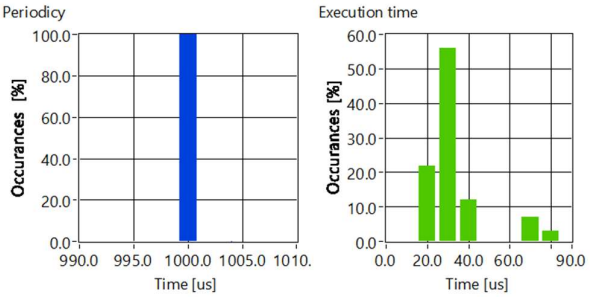


Fig. 7. SysTick IT periodicity and execution time statistics

Example for task execution time statistics is shown on Fig.8. These execution times show much higher variability, because the tasks usually perform much complex functions than interrupts, and their execution time highly depends on the actual control flow path.

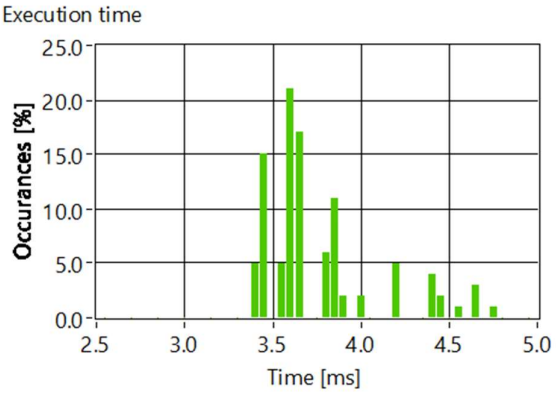


Fig. 8. Execution time statistics example of Low Priority Task

*E. Calculating the Worst Case Response Time (**Ri**) values*

Usually Deadline Monotonic Analysis (DMA) is used to calculate the worst-case response time of tasks (1). DMA is applying the following iterative formula, using the notations from Fig.3.:

$$R_i^0 = C_i$$

$$R_i^{n+1} = C_i + \sum_{\forall k \in hp(i)} \left\lceil \frac{R_i}{T_k} \right\rceil C_k \quad (1)$$

where

$$\sum_{\forall k \in hp(i)} \left\lceil \frac{R_i}{T_k} \right\rceil C_k$$

is the total interference from all higher-priority tasks, and **hp(i)** is the set of tasks with priority higher than **i**.

In this formula the worst case task execution times should be used as **Ci** and **Ck** parameters. This basic formula does not contain the IT effect compensation, but IT-s can be considered as very high level tasks, and in systems, where nested interrupt service (interrupts can preempt interrupts) is allowed their response time can be calculated in a same way.

Currently a basic calculation method using the maximum values from the execution time measurement statistics is used. But in the future it is possible, to implement a novel version replacing the single maximum values to statistical distributions from the task and interrupt timings.

IV. INTEGRATING THIS NOVEL MEASUREMENTS INTO A HIL TEST SYSTEM

The RTOS aware non-intrusive testing tool is written in LabVIEW; therefore, it is a straightforward decision to use NI VeriStand [9] as HIL environment form National Instruments.

To integrate the toolset into NI-VeriStand a so called Custom Device driver is needed. A NI VeriStand Custom Device functions as an interface to a special hardware. A Custom Device can have any number of input and output channels, and its functionality can be executed in every VeriStand engine cycle or can be implemented as a parallel executed task. In this integration the parallel executed task version is need, because the trace packet processing is an event based job. The Custom Device can provide event activation graph with timestamps for the test results display, and it also makes it possible to investigate the software timings in case of errors. The Custom Device also can provide Worst Case Response Time calculations to every tasks, which makes the signaling of Deadline violations possible.

V. SUMMARY AND CONCLUSIONS

This paper introduced the problem of multitasking and real-time failure detection in cyber-physical systems. The paper also described the ways of measuring RTOS task parameters important to detect this type of errors. A novel way of using non-intrusive tracing to measure these RTOS task parameters is introduced.

After the introduction of the concept the paper presented a LabVIEW based toolset, that performs these non-intrusive measurements. To demonstrate the operation of the toolset, measurement results using an ARM Cortex M microcontroller based test system running a FreeRTOS based multitasking software is presented.

These measurements are used to calculate and estimate the WCET for every tasks. With the estimated task WCETs the tool is able to determine the Worst Case Response Time of every tasks. This can ensure, that the test system notice Timing failures, like deadline violations of tasks, and interrupts.

The HIL test integration of these measurements also made it possible to provide information about the current state of the system's software in case of errors. For example, if the HIL system measure a non-correct response from the tested system, then our RTOS aware tool is able to show which tasks and interrupts have been executed and with what timings at a given time interval close to the error. Therefore, this tool can provide a great help for identifying the source and location of multitasking and real-time failures.

REFERENCES

[1] **ASPENCORE**: "2017 Embedded Market Study", *EETIMES, April 2017.*

[2] **A. Biagini, R. Conti, E. Galardi, L. Pugi, E. Quartieri, A. Rindi, S. Rossin.:** "Development of RT models for Model Based Control-Diagnostic and Virtual HazOp Analysis". *12th IMEKO TC10 Workshop on Technical Diagnostics. June 6-7, 2013, Florence, Italy.*

[3] **Balazs Scherer.**: "Hardware-in-the-loop test based non-intrusive diagnostics of cyber-physical systems using microcontroller debug ports" *ACTA IMEKO Volume 7 Issue 1* Pages 27-35.

[4] **H. Thane.**: "Monitoring, testing and debugging of distributed real-time systems" *In Doctoral Thesis, Royal Institute of Technology*, KTH, S100 44 Stockholm, Sweden, May 2000. Mechatronic Laboratory, Department of Machine Design.

[5] **Ken Tindell**.: "Deadline Monotonic Analysis", *Embedded System Progrming magazine*, June 2000.

[6] **R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, P. Stenström**.: "The worst-case execution time problem - overview of methods and survey of tools" *ACM Transactions on Embedded Computing Systems*, Volume 7, Issue 3, April 2008.

[7] **Richard Barry.: "Mastering the FreeRTOS™ Real Time Kernel"** *Real Time Engineers Ltd.*

[8] **Balázs Scherer, Gábor Horváth.:** "Microcontroller tracing in Hardware In the Loop tests", *Proceedings of the 2014 15th International Carpathian Control Conference (ICCC)*, Velke Karlovice, Czech Republic, 2014.

[9] **National Instruments**, "NI VeriStand™ Fundamentals Coures Manual" *Part Number 325785A-01*, 2011.

[10] **Jean J. Labrosse,** "MicroC/OS-II: The Real Time Kernel" 2nd Edition, CRC Press, February 5, 2002.

[11] **ARM**, "ARM® v7-M Architecture Reference Manual", ARM DDI 0403E.b (ID120114), ARM Limited 2014.