Low Effort Evaluation of Real-Time and Reliability Requirements for Embedded Systems

Ralf Mitsching, Carsten Weise, Thomas Gatterdam, Stefan Kowalewski
Embedded Software Laboratory
RWTH Aachen University
Aachen, Germany
Email: lastname@embedded.rwth-aachen.de

Abstract—Measuring reliability of embedded systems is an important but non-trivial problem. In a system design process, it is desirable to have early indicators for the reliability of an embedded system. Such reliability measurement will typically be carried out on system prototypes. The reliability measurements should thus be easy in set up, flexible to system changes, and hopefully low in cost.

In the paper we present a simple approach to do reliability tests for small-scale embedded systems. We explain our approach around a case study, but the method can easily be generalized. We discuss the advantages and disadvantages of our approach.

Keywords—embedded systems; reliability; measurements; rapid prototyping; real-time; microcontroller; linux;

I. INTRODUCTION

Embedded systems are omnipresent, they can be found anywhere. Their application areas are numerous and very often they are part of a safety-critical or at least mission-critical system. While failures of embedded systems might only lead to inconveniences in the best case, they can cause severe damage in worse cases. Reliability is thus a major focus in embedded systems design.

In the area of soft- and hardware architectures, methods and metrics are an established way to check and ensure the reliability of systems. In the area of hardware design, reliability of the complete systems can be inferred from the reliability of the components. Mean time between failures is another approach to reliability that can be found in hardware and has also been adapted to software. However, the holistic view on systems is often missing: methods that work with integrated systems composed of soft- and hardware components are rare [1]. This is especially true in the presence of real-time constraints.

In the design of embedded systems, it is important to test the reliability of the system and to probe for its compliance with the real-time requirements in case of a combined soft-/hardware-solutions with real-time constrains. Early evidence of evidence for the system reliability is an advantage. Thus, reliability checks should be carried out as early as possible in the software life cycle, when often the target hardware is not yet available. Instead, one needs use prototypes of systems to measure system reliability in early phases. As in the phases, changes in system architecture are still common, the measurement set up should come at low cost.

In our paper we will discuss a simple approach for early evidence on system reliability. We will present a straightforward set up that we have been using to test for reliability of a prototype we built within an industrial cooperation. We will use this case study as our illustrative example, but our architecture and methodology is of general nature and can be applied easily in similar situations. To help in deciding when our method could best be applied, we analyze our method and give lessons learned. We conjecture that this information is useful for people in research building prototypes within their projects, but also for people in industry where prototypes are built in early phases of projects.

Our paper is organized as follows: we first look into the set up of our case study and discuss how the prototype was built. Specifically, we discuss the choices made in selecting the hardware and software platform of the prototype. For this we first recapture basic real time design philosophies, and then discuss and describe the prototype of our case study. After these prerequisites, we explain how we set up the measurement system for the reliability measurements, and how the measurements were conducted. We close our presentation with an analysis of the features of our approach, including remarks on how to generalize the approach as well as lessons learned.
II. RELIABILITY MEASUREMENT SYSTEM ARCHITECTURE

This section describes the principle set up of our measurement system. As usual, we assume an embedded system consisting of a controller and a controlled system, connected via a well-defined interface (cf. Figure 1). The goal of our work is to test the reliability of the controller, called the ‘measurement object’ or ‘candidate’ in our set up. For the measurements, the controller must be connected to a measurement environment, cf. Fig. 2.

The controller consists of a real-time application and a soft- and hardware platform for the execution of the application. We assume that the real-time application is available, at least in parts. During early development phases a final version will not be available, but one can use the latest release of the application from previous development phases. Also the soft- and hardware platform might not be available: in practice, in an industrial cooperation, this platform is often a proprietary product of the industrial partner, and is often not accessible from outside the company labs. Even inside the company labs, access may be restricted. Therefore it is desirable to have a simple prototype platform available that resembles the real target platform in performance. We will sketch how such a prototype can easily be constructed. It may also be the case the real target platform is available. For the rest of the paper, we assume that the controller is available for use in our test set up.

Most of our measurements are carried out with the controller seen as a black box, only accessed via the system interface. We will also comment on several grey box measurements, where we have access to internals of the soft- and hardware platform.

As we are measuring compliance to timing constraints, we will need to monitor the interface with exact timing. This can normally not be achieved by off-the-shelf PCs. Instead, we use a special set up for our measurement environment, split up in a real-time capable measurement device and a measurement front end that needs not comply to any real-time constraints. Figure 2 illustrates this approach.

A. Basic principles

For the setup of our measurement system, we have components that need to real-time capable, and components that need not be. Here we recap some basic definitions of real-time.

As usual, we define a real-time system as a system that “must produce results and react to events in a timely, predictable manner, guaranteeing temporal restraints that are imposed by the applications” [2]. We further follow [3] in that we see (the execution of) a real-time system as faulty if results are not ready within the time specification. A correct execution requires all data to be available within the defined period. In order to fulfill timing requirements, we look at worst case execution times (WCET)[4] in our prototype.

To ensure real-time capabilities, there are two basic approaches: either use an existing real-time operating system, or design your own proprietary real-time kernel for the PC.

In the case of real-time operating systems, it is mandatory for our approach to use an open-source real-time operating system as adaptations of the operating system will be needed to achieve an efficient and functional system. In the next subsection we will discuss some alternatives of open-source operating systems that are suitable for building embedded system prototypes.

B. Open source real-time operating systems

An obvious choice for an open-source real-time operating system are Linux derivates. In contrast to a common Linux OS, which is designed to treat as much processes as possible in parallel, a real-time extension provides guaranteed execution time of special processes. Therefore the existing Linux kernel needs to be patched. Changes in the kernel are made in the interrupt handling – to avoid that interrupts can stop the execution of the real-time tasks – and an adjustment of the scheduler that ensures that real-time tasks (which have higher priority than usual Linux tasks) will not be preempted by other tasks. This priority based scheduling is a common method used in real-time systems.

C. Real-time on microcontrollers

An alternative approach to design a real-time system is using a microcontroller. Basically there are two ideas of how to implement software for a microcontroller. On the one hand there is the native implementation, on the other hand using a real-time OS for embedded systems.

1) Native approach: The native approach is an obvious possibility to reach low and predictable execution times. All parts of the program are sequentially executed. In this approach there is neither a scheduler given nor constructs for communication, synchronization, etc. Everything needs to be done by handwritten code. There is no need of task switches at run-time, which decreases latency. The upper bound of the execution time is the WCET. Thus hard real-time can be reached. Furthermore this approach allows full control over all timers and interrupts.
2) **Real-time OS for embedded systems:** The development of extensive software may require splitting the program into several smaller tasks. Running these tasks in parallel requires a scheduler. Real-time operating systems for embedded systems typically provide a scheduler plus functions and structures for communication and synchronization of the tasks.

There is a difference between statically and dynamically scheduled operating systems. Whereas a dynamic scheduler calculates at run-time, which task has to be executed next, in a static real-time operating system the task-set with its priorities is fixed. The scheduling is calculated at compile-time and cannot be changed after this. The advantage is a very low scheduling time overhead at run-time, which is independent of the complexity of the scheduling algorithm [5]. In contrast, a dynamic scheduling allows for more flexibility when reacting to environmental events at run-time, with the side-effect of more scheduling overhead. Examples for real-time operating systems for embedded systems are:

- OSEK-OS  
  A real-time OS with static scheduler.
- uCLinux  
  As any Linux, uCLinux is a dynamically scheduled operating system.

### III. Setting up the Measurement System

There is a growing number of methods available to evaluate real-time capability and reliability of software and hardware. For software systems several methods are available to evaluate real-time compliance. However, input values of these methods are based on estimations, as their calculation is a laborious task. In the case of embedded systems, such methods need additionally be capable of handling complex combinations of hardware and software components. The definition of such methods is a quite challenging task.

In this section, we present a workable set-up. A main objective is the simplicity of our methodology. This methodology can be used to compare different systems with regard to the real-time and reliability behaviour.

First, we give an overview of typical requirements for real-time systems before deriving reasonable criteria from them. Then we present the chosen measurement set-up.

#### A. Requirements

The following are the requirements towards our measurement system:

- the candidate needs to respond to stimuli within a predefined period
- violation of the time limit is treated as failure
- high message throughput
- a required respond must always be present (within the given time limits)
- low effort for porting the software

#### B. Measurement Criteria

The measurement criteria are derived from the requirements as given above. ISO 9126 [6] defines six attributes for software quality, that can be used to assess the interesting aspects of the system behaviour [7]. To evaluate our platforms it is necessary to refine them into sub-attributes, that depend on our list of requirements. Below, we present a list of attributes that are relevant in the context of our work. These attributes can be used in order to check compliance to our requirements.

1) **Latency:** Latency means the time between the end-of-frame of a transmitted message and the end-of-frame of the received message. Therefore messages will be send periodically to the real-time system. The minimal, maximal and average latencies are measured over a given time period. The maximal latency is an essential characteristic of a real-time system.

2) **Throughput:** This attribute describes how many messages can be handled by the system per time unit. The system will be flooded with messages and the number of correctly processed return messages is counted.

3) **Integrity:** Message integrity is an important criterion, as it is the prerequisite for all further measurements. Absence of an expected message results in an incorrect measurement.

4) **Robustness:** The attribute robustness is derived from the requirement, that a violation of the time limit is treated as failure. In our context robustness is defined as follows: Robustness is the ability of the system to send correct answers without increased latency in the case of interfering influences. These could be parallel running processes, hard disk accesses or similar. To show the robustness of the RTAI system, all measurements are done twice. First with fully loaded CPU and then without load. There is no need of special measurements for the native approach on the microcontroller as the native approach does not support any parallel running processes. In other words, robustness is given to the native microcontroller approach by definition.

5) **Portability:** Portability is measured as the effort needed to port the software form the prototype to the real target hardware platform. The effort is rated by the code changes needed to port the original software. This criterion is a subjective benchmark, because the effort depends on the skills of the developer.

6) **Re-usability:** Re-usability describes how much effort is needed to use this software with a new hardware while using the same code base. The amount of re-usable code is an indicator for this attribute.

#### C. Measurement environment

Fig. 3 illustrates the complete setup for our measurements. The measurement environment (ME) includes the measurement unit and the diagnostic unit: The measurement unit is realized by an Atmel AT90CAN microcontroller. The architecture of the software executed on the measurement
unit is also shown in Fig. 3. The real-time communication unit is responsible for the controlling of microcontroller CAN interface. Transmitted and received CAN messages are marked with timestamps that are created by an internal CAN timer. Functions to measure the message throughput and latency are implemented for the microcontroller in this unit. The component CAN communication protocol defines the communication behavior used to command the system under test. The component message generation encapsulates the control and evaluation of the complete measurement. After evaluating the messages, the results are stored in a permanent storage. The diagnostic unit uses a serial port to access the measurement unit. The diagnostic unit does not support real time capabilities. It provides the measurement front end to command the measurement and to display the results. The program consists of three components: The communication unit provides the low level communication. The measurement control unit contains all functionalities to control the measurements. The measurement data display is used to display the results. Furthermore, Fig 3 shows the connection between the Measurement Environment and the measurements candidates.

Figure 3. Measurement Setup

Figure 4. Measurement environment and ARM hardware platform

IV. CASE STUDY

A. Use Cases

1) Selection of Hardware platforms: Within the industrial case-study used to evaluate our approach we selected two different hardware platforms. On the one hand we used a system with a real-time extension for Linux, on the other hand we used a native implementation on a microcontroller. To select the matching real-time extension, the following goals had to be achieved:

- **Hard real-time in user-space**
  Simplifies development, because user-space supports safety mechanisms like memory protection.
- **Device drivers with real-time support**
  When hardware devices are used, there is a need of real-time drivers which do not block while running a real-time task.
- **Mature and well documented**
  There should be information available that makes it easy for the developer to learn the use of the real-time extension. This implies a good system description as well as a well documented API.

RTLinux only provides hard real-time in kernel-space, thus does not fit our requirements. Since we use a Peak System PCAN-PCI card, it is important to use real-time drivers for it. Such CAN-drivers are supported by Peak System for Xenomai and RTAI. With respect to lower latencies, we selected RTAI as candidate for the real-time extension. The hardware setup consists of an Intel Pentium 4 with 3.2 GHz and the previous mentioned Peak System PCAN-PCI card.

The requirements for the microcontroller are as follows:

- **Integrated CAN-controller**
- **ARM or similar architecture**

These are achieved by an Olimex development board with AT91SAM7X256 ARM-microcontroller by Atmel. It also provides the required CAN-interface. According to our experience in developing embedded systems on this board the clock speed should be low. The microcontroller runs with a frequency of 55 MHz. We decided to use a native approach as this is a popular method in industrial projects. This approach should exhibit the best possible performance of the platform.

2) Selection of Software: The software meant for the selected hardware platforms is a software to simulate elements of signaling equipment for rail traffic (taken from an industrial partner), like track switches or electrical light signals. It is implemented in C++ and QT for the use on a windows system. The simulator communicates with a real railway control center by four CAN-interfaces.

B. Conducting the Case Study

1) Port to RTAI:

- **Installation of Ubuntu and kernel patching** - The installation of the RTAI real-time extension was mostly
done as described in [8]. When doing this, the actual Ubuntu Linux version was 9.04. Furthermore RTAI 3.7 and a vanilla kernel 2.6.28.7 were used. This kernel was patched with hal-linux-2.6.28.7-x86-2.2.06.patch, provided by RTAI.

After this, the following changes were done in the kernel configuration (`menuconfig`):

- General setup - Local version - append to kernel release = - rai-3.7
- Enable loadable module support - Module versioning support = no
- Processor type and features - Subarchitecture Type = PC-compatible
- Processor type and features - Processor family = Pentium-4
- Processor type and features - Interrupt pipeline = yes
- Power management options - Power Management support = no
- Power management options - CPU Frequency scaling - CPU Frequency scaling = no

Now the kernel could be compiled, which took more than one hour, and installed on the system.

- **Installation of RTAI** After installing a patched Linux kernel, RTAI needs to be configured with `menuconfig` as well:
  - General - Linux source tree = /usr/src/linux-headers-2.6.28.7-rtai-3.7
  - Machine (x86) - Number of CPUs (SMP-only) = 1
  - Add-ons - Real-Time Driver Model over RTAI = yes

It is now ready to compile and install. To test the correct functionality of RTAI, one of the included testsuites was used.

- **Real-time CAN-driver** Peak System provides real-time drivers for their PCAN-PCI card. The up to date driver version at the time we installed the system was 6.9b. [9] describes how to configure and install the driver with real-time support on RTAI.

Before loading the driver, the RTAI modules `rtai_hal`, `rtai_sched`, `rtai_sem` and `rtai_rtdm` need to be loaded.

- **Changes** At first all funtion calls of the CAN API need to be adapted to the Linux CAN driver. The only thing to consider is that calling the CAN API needs to be done within a real-time task. This is achieved by adapting the task for real-time. The existent QThreads are internally implemented by pthreads. Using the RTAI/LXRT API as described in [10], these pthreads can achieve hard real-time in userspace.

Furthermore it is essential to get rid of so called syscalls [11]. Syscalls can cause unpredictable execution times and thus violate real-time requirements. A common example for a syscall is the access to a hard disk. But even QT signals could cause syscalls, so it is preferred to use semaphores provided by RTAI instead.

2) **Port to ARM microcontroller:**

- **CAN-driver implemenation** Before porting the software to the ARM architecture it is necessary to implement a CAN-driver. Therefor we developed a driver with nearly the same interface as provided by the RTAI API. The driver supports the use of the hardware CAN timestamp. Incoming CAN messages are automatically added with this timestamp by the microcontroller.

- **Changes** Instead of using a thread to check for new CAN messages, the microcontroller uses an interrupt service routine to handle incoming CAN messages. Furthermore, as we are using a native approach, threads are not available. Thus, instead of checking semaphores within threads, a central checking of semaphores ensures that the correct functions are executed.

In the last step, all QT objects need to replaced by their equivalent C++ objects. As an example, the QT queue is replaced by the C++ queue.

### C. Results

The presented measurement environment (cf. Sect. III-C) was used for several measurements on both hardware platforms. This section presents the results of our measurements.

1) **Latency:** Table I lists the latency results. The times in the table represent the time between end-of-frame of the transmitted message and end-of-frame of the received answer. There is an additional line “RTAI with HDD access” to demonstrate how hard disk access within a real-time task will cause violation of real-time. The table shows that the RTAI approach has much lower latency times than the microcontroller. It is also clearly seen that there is no difference in the maximal latency of the RTAI with full load compared to no load.

<table>
<thead>
<tr>
<th>Platform</th>
<th>Minimum</th>
<th>Average</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM</td>
<td>440 µs</td>
<td>456 µs</td>
<td>550 µs</td>
</tr>
<tr>
<td>RTAI no load</td>
<td>230 µs</td>
<td>250 µs</td>
<td>330 µs</td>
</tr>
<tr>
<td>RTAI full load</td>
<td>240 µs</td>
<td>255 µs</td>
<td>330 µs</td>
</tr>
<tr>
<td>RTAI with HDD access</td>
<td>20,3 ms</td>
<td>22,3 ms</td>
<td>81,3 ms</td>
</tr>
</tbody>
</table>

Table I

**RESULTS LATENCY**

2) **Throughput:** Table II presents the results of the throughput measurements. The numbers are the processed messages per second. On each platform a total of more than 500,000 messages were processed. One can see that the microcontroller is able to process 10% more messages per second. Furthermore there is only a very small difference in the throughput achieved with and without load on the RTAI system.

<table>
<thead>
<tr>
<th>Platform</th>
<th>Minimum</th>
<th>Average</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM</td>
<td>2397</td>
<td>2397</td>
<td>2398</td>
</tr>
<tr>
<td>RTAI no load</td>
<td>2178</td>
<td>2181</td>
<td>2184</td>
</tr>
<tr>
<td>RTAI full load</td>
<td>2176</td>
<td>2177</td>
<td>2183</td>
</tr>
</tbody>
</table>

Table II

**RESULTS THROUGHPUT**
3) Integrity: During each measurement, the number of transmitted and received messages was compared to guarantee message integrity. Additionally each message was added a unique number, which was checked on receive. Both platforms fulfilled this criterion.

4) Robustness: For measurements with full load on the RTAI system, the tool stress [12] was used. This tool creates a 100% CPU load. As seen in Table I the maximal latency did not change compared to no load. According to our definition of robustness, this criterion is fulfilled. As mentioned before, the criterion is implicitly fulfilled by the microcontroller approach.

Table III
PORTING EFFORT

<table>
<thead>
<tr>
<th>Platform</th>
<th>new/changed LOC</th>
<th>.diff lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM</td>
<td>204</td>
<td>1946</td>
</tr>
<tr>
<td>RTAI</td>
<td>198</td>
<td>1698</td>
</tr>
</tbody>
</table>

5) Portability: The porting effort displays a subjective value. To represent it in numbers, the sourcecode of the ported software was compared to the originally software. Table III shows two methods of this measurement. On the one hand there is the number of new and changed lines of code. The lines of code needed to implement the CAN-driver is not included in this table. Therefore another 75 lines of code are implemented.

On the other hand there is the number of changed lines generated by an unix diff command. The numbers in that column show the sum of lines over all .diff files.

As the numbers do not constitute an absolute measurement, an additional qualitative evaluation, shown in Table IV, shows the porting effort from the view of the developer.

Table IV
QUALITATIVE PORTING EFFORT

<table>
<thead>
<tr>
<th>Platform</th>
<th>Concept</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM</td>
<td>+</td>
<td>0</td>
</tr>
<tr>
<td>RTAI</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

+ low o medium – high effort

6) Re-Usability: Most of the source code in the RTAI approach can be re-used on any platform which is able to run a linux with the RTAI real-time extension. The only changes to be done is the adaption to the platform dependend CAN driver. Thus, this code is highly re-usable on such systems. The native approach on the microcontroller is a very specific implementation to the hardware: The CAN driver needs to be implemented for each hardware individually; the scheduling algorithm and interrupt handling need also be adapted per architecture. So the re-use of this approach would need a lot more adaption work than the RTAI approach.

D. Lessons learned

The integration and application of the presented RT extension were carried out without major problems. Typical problems (e.g. incompatibility with existing system libraries or version conflicts) did not arise. In our opinion, the effort to read and understand the available document of the RT extension is low. This seems to be a matter of experience and ability of the designer, thus it is hard to give some generic rules. However, during the case study the designer spent only 16h to deal with the complete integration. Hence, the available documents seem to be appropriate to introduce developers to this field.

The native approach requires the verification of the RT requirements and additionally requires a lot of knowledge of the structure of the used microcontroller and experiences with the execution order of C and assembler code. If we consider the worst case, each code change requires a new calculation of the execution time. In our case, this needed to be done manually and has been really time consuming. This weakness could also be seen as strength of the native approach. The developer has to understand every detail of the program execution in order to determine the RT requirements. Hence, faults in the specification or implementation can be found, which is a considerable advantage of the native approach. Thus, this approach can lead to higher performance and a more efficient use of the hardware resources. Note that the skills and motivation of the developer are influencing this objective. In our opinion, there is no predictable way to force improvements this way.

Remarkable is that the RT extension facilitates the reduction of code complexity. This can be explained by the existence of explicit program statements to describe RT behaviour. These statements allow for labeling the aspects relevant to RT in the code. In this way, it is easily possible to gain a quick overview on all defined time conditions. Especially when new requirements shall be added or existing requirements shall be changed, this advantage is very important. For an external evaluation that is possibly run by people without having distinct knowledge in the area of RT programming, this aspect is also very useful. Usually, the traditional development process requires traceability between different documents of the process. Even if an automisation would be possible, the explicit representation of RT aspects in the code allows for a better traceability. In case of the native approach, this is possible using meaningful names and comments in the code. It needs to be stated that these comments need to be kept consistent with the other documents. During the case study, it became obvious that the native approach causes a significantly higher effort for porting. This can be explained with the complete rearrangement of the existing RT code in order to meet the RT requirements.

The RT OS provides a notation on an abstract level. Besides the reduction of complexity, this leads to a higher
portability and reusability of the developed code. In this way, the code developed with the RT approach should be portable between different microcontrollers without much effort. Thus, the RT approach abstracts from the used platform. Further it seems appropriate to use a company specific software library for a certain class of microcontrollers. Obviously, this implies that the new platform must adequately support the RT operating system.

V. EVALUATION OF THE PROPOSED MEASUREMENT METHODOLOGY

The main objective of our work was to see if our simple approach is suitable to evaluate two systems with regard to the real-time and reliability capacity. On the basis of the performed case study, where we ported an industrial developed simulator to both platforms, we were able to retrieve measurement results and gain experience to draw conclusions on the advantages and disadvantages of our approach.

In this section, we consider important criteria to evaluate our approach.

- **Adaptability** How does the ME handle new systems and software? The measurement environment is modularly built and thus easily adapts to changes in the system architecture. Adaptation can happen per layer. On the communication layer, our case study uses the CAN interface of a microcontroller. This part can easily be exchanged with other hardware serving different sensors and actuators. It might be necessary to exchange the microprocessor as well, but the rest of the system can be used as before. As the telegrams arriving at the microcontroller are time-stamped, differing delays in the connection between microcontroller and diagnosis PC do not pose a problem. Generation of telegrams that are sent from the microcontroller is also not a major problem, as these can often be a-priori formatted and stored in the microcontroller. If this is not feasible, then generation of telegrams in the microcontroller must be considered, and thus a more powerful microcontroller might be needed.

New software architectures of the prototype do not cause problems due to the CAN-interface. However, controlling the communication unit of the microcontroller (i.e. packet formats) should be clearly separated from the communication behaviour (i.e. state changes in the communication protocol). As the complete protocol inclusive of timing information is programmed in software, arbitrary changes of the protocol are possible. It should be ensured that the communication part is implemented according to the usual layer models.

- **Verification of results** How does the ME avoid faults in the Measurements? Message losses may lead to faulty measurements of latency times, therefore a sequence number is added to every message sent from the measurement device. Latencies are only computed for matching sequence numbers, otherwise an error counter is increased. The CAN-bus itself is protected by checksums, so we assume correct transmissions on the bus.

- **Repeatability (results)** Does the ME always deliver the same results? For each measurement candidate several measurements were performed. More than 20,000 messages were processed for the latency measurements on each candidate. Their jitter is depending strongly on the candidate, while the microcontroller showed stable results. There was no jitter caused by the measurement device. A total number of more than 500,000 messages were processed in the throughput measurement for each candidate. The numbers achieved during this were almost the same in each run.

- **Evaluation of response data (I/O)** How is the relation between measured input and output are collected? To distinguish between time critical and non time critical tasks in the measurement device, a CAN message ring buffer was implemented. For time critical tasks, both incoming and outgoing messages trigger an interrupt, and its interrupt service routine timestamps the messages in the ring buffer. A flag identifies a message as incoming or outgoing. The calculation of the results is done in the main loop, as it is not time critical. For new messages in the ring buffer the calculation can be done on the buffered messages.

- **Evaluation of response data (Timing)** How is the timing behaviour of the measured input and output are collected? As said before, sent and received messages trigger interrupts, that simultaneously set a hardware time stamp at the end-of-frame of the CAN message, that is stored together with the CAN message in the ring buffer. When the system is idle, i.e. not serving interrupts, it processes the timestamps in the ring buffer. This guarantees exact measurements due to the use of interrupts and hardware timestamps.

To ensure the correct timing of the outgoing messages, a timer is used that regularly triggers an ISR which will send messages out at the right time. As the ISR is triggered every millisecond, we are accurate up to one millisecond.

- **Objectivity of the methodology** How does the ME limit the influence of the designers (=degree of human influence)? The influence is different when looking at the measurements and evaluation of the measurements by the experiment supervisor. If the measurement is not well-suited for the problem, then this can only be adjusted by the supervisor. For the measurements, our diagnosis approach ensures correctness, but it cannot ensure appropriateness. This can only be dealt with on the team level, e.g. by distributing responsibilities. These management problems are not in the focus of
our work.

- **Effort of measurements** How time consuming is the proposed methodology? The time needed is of cause proportional to the degree of detail in the measurement. The approach for building the prototype seems to be easy to learn. For the evaluation and interpretation of measurement results, domain knowledge is a must. This is no restriction on our method as such, but merely a prerequisite for the real-time controller as well.

- **Portability** Is the methodology applicable in other domains? Due the generic architecture of the system, the approach can be used wherever CAN-buses are used for the communication network. Other transport media should however not cause a problem, but might need some tailoring on the interface-hard- and software. As we have a black box view of the SuT, functionality of the system is of no major interest. Our system only measures the communication along the system bus, but in most situations one can conjecture that correctness of the system follows from the correctness and efficiency of the protocol. In all these cases it should be possible to adapt our approach.

- **Scalability** How does the ME handle increasing complexity of system and software architecture? Scalability ask for the relation between problem solution and size of the environment. We have only used our approach in our lab, and thus cannot comment on the consequences for our approach when the system becomes medium-sized, say a network of 80 controllers, as can e.g. be found in modern car technology. In fact, we do not expect our approach to scale very well. It is meant as a simple and cost efficient approach to small prototypes. We see the benefits from the simple architecture as more useful in our area than the benefit from a more complex, but scalable architecture.

VI. **Conclusion**

We discussed a simple approach for early evidence on system reliability. We presented a simple set up that we have been using to test for reliability of a prototype we built within an industrial cooperation and used this case study as our illustrative example.

We analyzed our method and gave lessons learned from our experience. We assume that this information is useful to people in research building prototypes within their projects, but also for people in industry where prototypes are building early phases of projects.

**References**


