SIMULATION-BASED APPLICATION SOFTWARE DEVELOPMENT IN TIME-TRIGGERED COMMUNICATION SYSTEMS

Alexander Hanzlik

Austrian Institute of Technology, Vienna, Austria
alexander.hanzlik.fl@ait.ac.at, ahanzlik@gmx.at

ABSTRACT

This paper introduces a simulation-based approach for design and test of application software for time-triggered communication systems. The approach is based on the SIDERA simulation system that supports the time-triggered real-time protocols TTP and FlexRay. We present a software development platform for FlexRay based communication systems that provides an implementation of the AUTOSAR standard interface for communication between host application and FlexRay communication controllers. For validation, we present an application example in the course of which SIDERA has been deployed for development and test of software modules for an automotive project in the field of driving dynamics control.

KEYWORDS

Simulation Based Software Development, Time-Triggered Communication Systems, FlexRay, Distributed Systems, Software Development Process Acceleration

1. INTRODUCTION

Distributed fault-tolerant real-time systems are more and more deployed for dependable control systems in the automotive industry. Modern vehicle control systems are built from spatially separated electronic control units (ECUs) interconnected via a shared communication resource. ECUs are embedded systems that control one or more of the electrical systems or subsystems in a vehicle. Different ECUs are assigned different control tasks, like engine control, transmission control, convenience electronics control and more.

For communication between the different ECUs in a car, time-triggered communication systems like Flexray [5] are particularly suitable due to their deterministic behavior. Among other benefits, time-triggered communication systems guarantee a-priori known maximum message transmission times using a collision-free access to the shared communication resource. The FlexRay industry consortium drove forward the standardization of a time-triggered fault-tolerant communication system for advanced automotive applications. With the completion and the delivery of the final version of the FlexRay specification the consortium disbanded in 2010. Currently, activities are in progress to integrate the FlexRay standard into the ISO (International Standardization Organization) catalogue of norms.

AUTOSAR (AUTomotive Open System ARchitecture) is an open and standardized automotive software architecture, jointly developed by automobile manufacturers, suppliers and tool developers. The AUTOSAR consortium provides, among others, standard specifications for the FlexRay communication stack that define the interaction between host application and FlexRay communication controllers via standardized interfaces. These interfaces hide hardware related dependencies from the host application and allow to incorporate FlexRay communication controllers from different vendors without impacts on the application software.

SIDERA [9] (SImulation model for DEpendable Real-time Architectures) is a simulation environment for time-triggered communication systems. The environment is based on the Time-Triggered Architecture TTA [13] and allows the execution of the FlexRay protocol on simulated communication controller networks; SIDERA has been used for the investigation of synchronization mechanisms in the Time-Triggered Architecture [8][12] and in FlexRay based communication systems [10]. It is a pure software solution - simulation is performed on a single computer system without the need of special (communication controller) hardware. A host application interface allows to incorporate and run user provided code on the simulated communication controllers. This mechanism allows analysis and debugging of existing application software modules in a simulation environment as well as development and test of new application software modules prior to integration into the real system.

The technical challenge is to switch between a development and test environment and the real system without modifications of the original application code. This challenge is addressed in this paper.

2. MOTIVATION AND OBJECTIVES

Why should a simulation based approach for ECU software development make sense? The following reasons and considerations are, among others, inspired from daily practice.

Building and loading. The application software for a typical ECU consists of many different modules like the real-time operating system, the communication protocol, drivers for peripheral devices, diagnostic management services and others. A common method is to compile all modules and to link them into one binary file that is loaded into a non-volatile memory area (usually a flash memory) on the ECU from where it is started after each power-on of the ECU.

Each modification in any single ECU software module makes the following steps necessary:

1. Implement the changes in the software module.
2. Build the ECU application binary.
3. Load the binary into the ECU non-volatile memory.
4. Test the software.

Steps 2) and 3) typically take 5-10 minutes (this duration is based on personal experience made in typical software deveopment projects for embedded systems). This is a considerable delay between implementing a possibly simple modification in the software and being able to test this modification. Especially in early development phases where many modifications and tests are necessary much time gets lost for such ”idle” phases in the development process.

Debugging. Debugging in a distributed real-time environment is an ardous task due to the following adversaries:

- **Probe effect**. Setting breakpoints in one application task may change the timing behavior of other ECU components. Effects that are observed during debugging may not occur in the running system and vice versa.

- **Breakpoint limitations.** Some processors only allow a limited number of on-chip breakpoints to be set. It’s quite hard to realize reasonable debug sessions having only a handful (or less) breakpoints available.

- **External components.** Debugging communication with an external component (e.g. a sensor) that is not in the scope of control of the debugger is quite hard. Setting a breakpoint in the host application will not stop operation of the external component.

- **Watchdogs.** Safety measures (e.g. hardware watchdog timers) may cause the ECU to reset when a task is stopped by a breakpoint.

Hardware availability. A quite pragmatic aggravation that should not occur, but that is however observed practical experience, especially in very busy development phases: there is always some piece of hardware missing, be it the ECU itself, the debugger, the power supply, some special cable or anything else.

The objectives of a simulation-based approach for software development and testing are straightforward: it shall be possible to

- **Execute host applications on simulated FlexRay communication controllers**
- **without the need of FlexRay communication controller hardware,**
- **without the need of ECU hardware and**
- **without modifications of the original code**

with the following aims:

- **Speed up the software development process** by eliminating ”idle” phases due to time consuming load generation in early development phases.
- **Ease the analysis of distributed system behavior** by removing obstacles to debugging in distributed realtime systems, like the probe effect.
- **Overcome hardware non-availability** by testing software components on simulated ECUs.
- **Speed up the integration process** by testing the correctness of the control flow of the software part in the simulation environment prior to integration into the hardware system.

3. PREREQUISITES AND ASSUMPTIONS

3.1. System Structure

For the considerations in this paper we assume a system structure as shown in Figure 1.

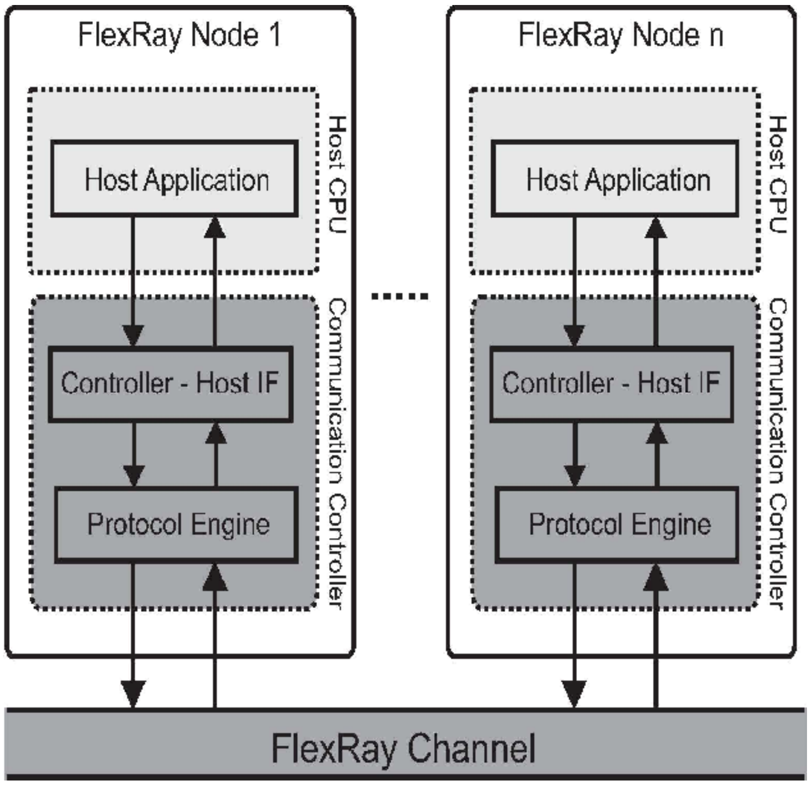


Figure 1. FlexRay Cluster

A system consists of a set of *nodes* that concurrently execute a distributed real-time application. The nodes communicate by sending and receiving messages over a dedicated communication *channel*. The nodes and the communication channel form a *cluster*.

Each *node* contains a *Host CPU* and a *Communication Controller (CC)*. The host CPU executes the real-time application whereas the CC executes the communication protocol and provides access to the communication medium for the host CPU.

The CC contains a *controller host interface (CHI)* and a *protocol engine (PE)*. Via the CHI, the host computer can issue commands to the PE (e.g. send a message) and receive indications from the PE (e.g. a message has been received). The PE is responsible for execution of the communication protocol and provides protocol services for the host via the CHI.

3.2. SIDERA Software Architecture

Figure 2 shows the software architecture of SIDERA.

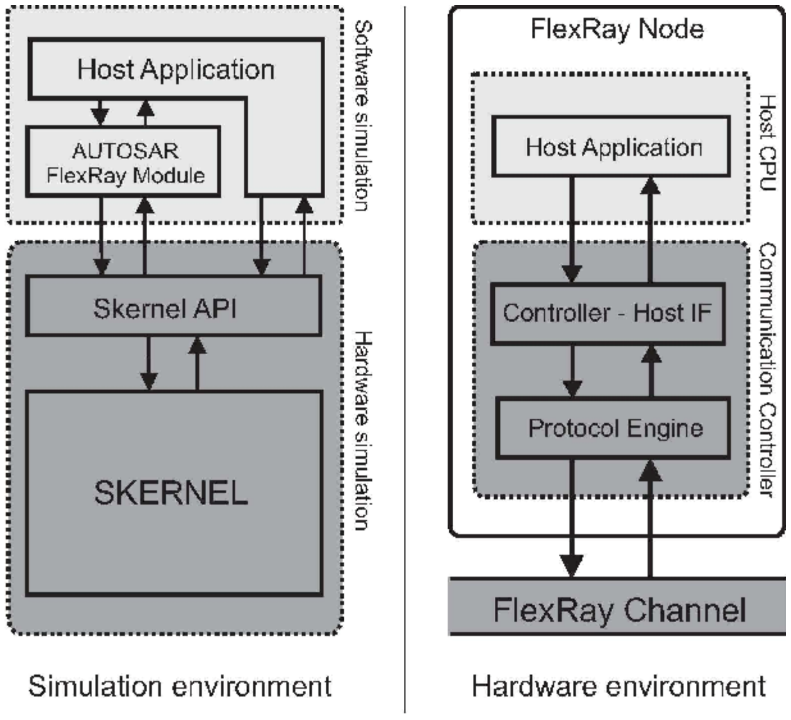


Figure 2. SIDERA Software Architecture

It consists of a set of layers, namely the *host application* layer, the *kernel interface* layer SKERNELAPI and the runtime *kernel* layer SKERNEL. The *AUTOSAR FlexRay* module is the protocol-specific part of the architecture and contains the implementation of the FlexRay Interface for the host application (an implementation of the Time-Triggered Protocol TTP [13] as well as a basic implementation of the Time-Triggered Ethernet (TTE) [11] protocol functions are also available). The mapping of the different software layers to the system model components is also shown in Figure 2.

The *host application* layer contains the host application under test. The host application communicates with one or more communication controllers using the *AUTOSAR FlexRay* module.

As shown in Figure 3, the *AUTOSAR FlexRay* module consists of the *FlexRay Interface* (FrIf) [6] that controls one or more *FlexRay Device Drivers* (Fr) [7]. The Fr part hides the vendor specific hardware and implementation details of each communication controller in a set of standardized functions accessible by the FrIf. As also shown in Figure 3, the host application of a single ECU may control a set of communication controllers of different types (e.g. CC1 Type A, CC1 Type B and CC1 Type C). Each of the different controller types is managed by a dedicated device driver (e.g. CC1 Type A is managed by driver FR_1_A, CC1 Type B by driver FR_1_B and CC1 Type C by driver FR_1_C, respectively). The different device drivers are managed by the FrIf. A unique addressing scheme determines how the possibly different communication controllers are referenced by the host application.

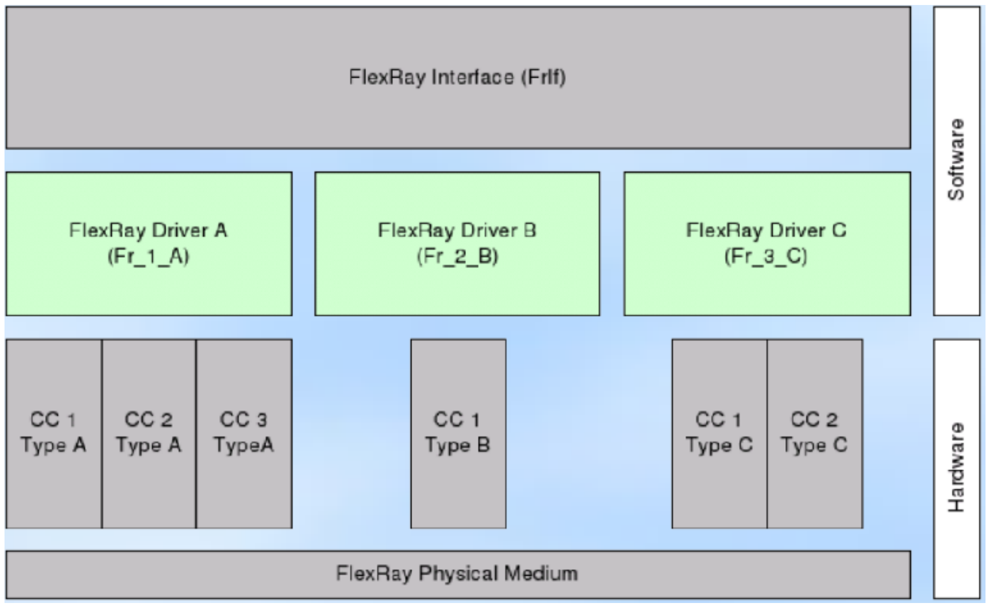


Figure 3. AUTOSAR FlexRay Module

The SKERNELAPI layer provides access functions for control and of the SKERNEL. These functions can be grouped into

- *Configuration functions*
  for configuration file handling. *Example:* Read/Write configuration file.
- *Control functions*
  for interaction with the Graphical User Interface (GUI). *Example:* Start/Stop/Pause/Resume simulation.
- *User code integration functions*
  for interaction with the host application. This function group allows to associate user defined functions to specific events during simulation. *Example:* call a given function whenever a message is received.
- *Kernel access functions*
  for manipulation of specific system components. *Example:* take a specified node out of service.
- *Protocol functions*
  implement protocol specific functionalities. *Example:* Determine the FlexRay frame identifier of a received message.

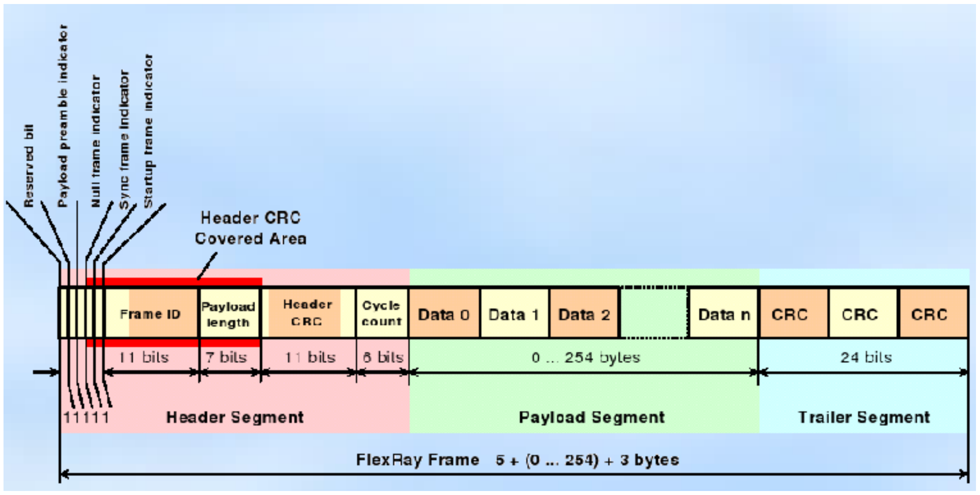


Figure 4. FlexRay Frame Format

The runtime kernel layer SKERNEL contains the hardware related part of the system architecture. It implements the complete FlexRay cluster, i.e. the set of ECUs communicating via the FlexRay communication channel as shown in Figure 1. For the communication between the different ECUs, the FlexRay Standard Specification [5] defines the frame format shown in Figure 4.

3.3. Host Application Integration

The integration of the host application proceeds in two steps. Figure 5 shows the principle of operation.

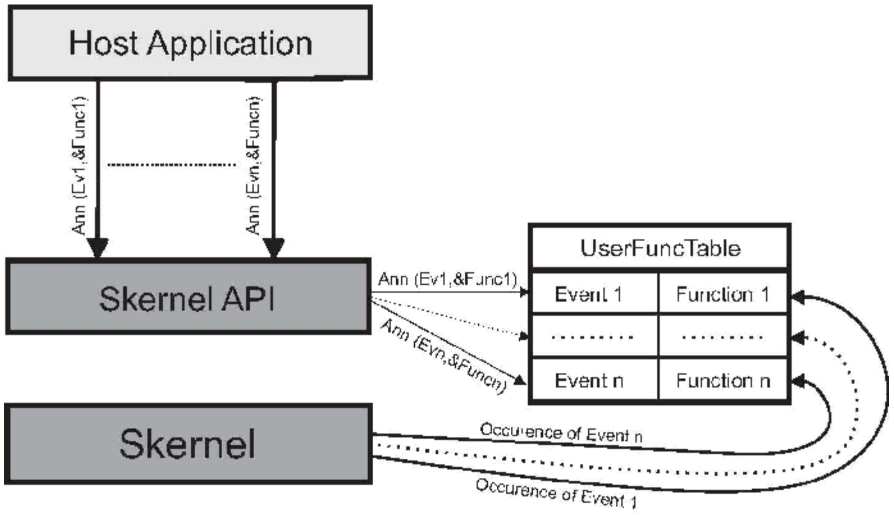


Figure 5. Host Application Integration – Principle of Operation

Initialization of the host application. The simulation environment provides the global variable *g_pUserInitFunc* that contains a pointer to the host application init function. The *g_pUserInitFunc* pointer has to be initialized by the host application. The host application init function is called by the SKERNEL layer when the simulation starts.

Installation of event functions. In a second step, so-called *event functions* are associated to specific events during simulation. These event functions are called by the SKERNEL layer every time a specified event occurs (e.g. if a message has been received). The event functions are passed the cluster number *iCluster* and the node number *iNode* as parameters to know at which node the specified event has occured. Figure 6 shows a code sample of a host application integration process. The host application init function *host_app_init()* installs two event functions: *host_app_callback_macrotick* is called when a new macrotick is generated, *host_app_receive_msg* is called upon reception of a message, respectively.

```
#include <skernelapi.h>

int host_app_callback_macrotick (int iCluster, int iNode)
{
      int MacroTick = skernel_get_macrotick_counter(iCluster, iNode);
      ADD_LOG_ENTRY (
            "Node %d’s clock in Cluster %d shows global time %d.\n",
            iNode, iCluster, MacroTick);
      return (0);
}

int host_app_callback_receive_msg (int iCluster, int iNode)
{
      ADD_LOG_ENTRY(
            "Node %d in Cluster %d has received a message.\n",
            iNode, iCluster);
      return (0);
}

void host_app_init()
{
/* init user data here */
      skernel_user_callback_announce(USERFUNC_MACROTICK,
            host_app_callback_macrotick);
      skernel_user_callback_announce(USERFUNC_RECEIVE_MSG,
            host_app_callback_receive_msg);
}

userfunc_t g_pUserInitFunc = (userfunc_t)&host_app_init;
```

Figure 6. Host Application Integration – Code Sample

3.4. Scope of Services

SIDERA provides the following scope of services for host application software development and testing:

Graphical User Interface (GUI). A GUI is provided that allows control of the simulation process as well as subsequent analysis of simulation experiments. The visualization of simulation results can be customized. The user may select what is logged and how the logged data is presented (see right hand side of Figure 7).

FIBEX file handling. For simulation of FlexRay clusters, a FIBEX [4] file is needed. This file contains all characteristics of the FlexRay cluster, including the hardware structure (among others, the number of nodes and the number of communication channels) and the communication schedule (the assignment of communication bandwidth to nodes).

FlexRay Cluster Hardware Simulation. The hardware simulation part includes simulation of the host CPU, the communication controller and the FlexRay communication channel according to the contents of the FIBEX file.

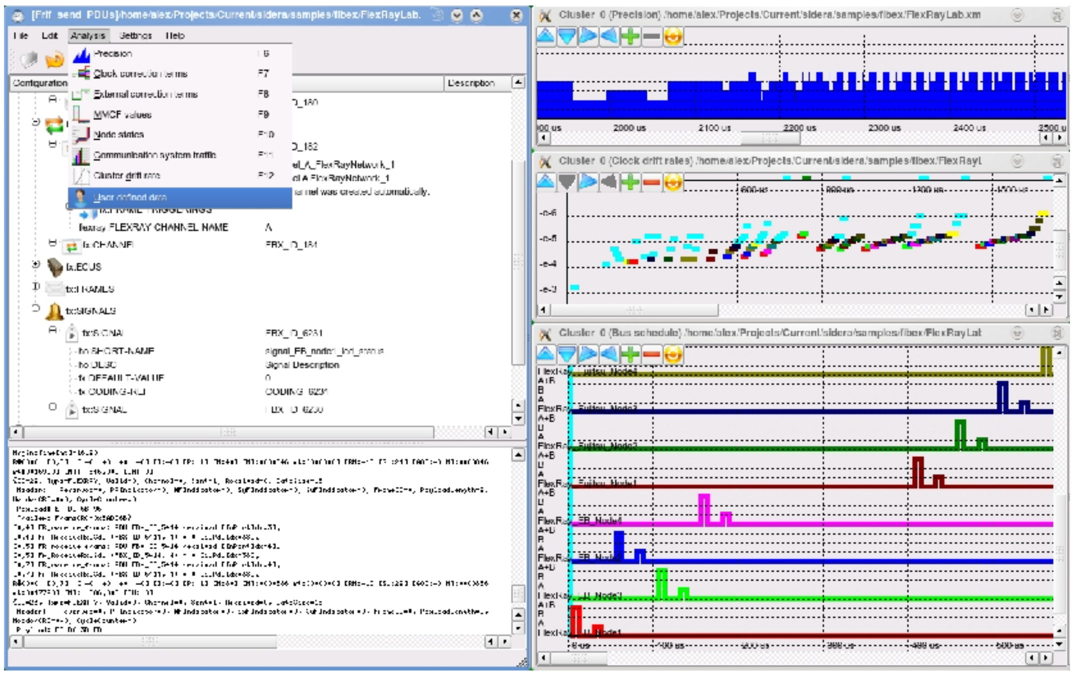


Figure 7. SIDERA Graphical User Interface

FlexRay Protocol Simulation. The FlexRay protocol simulation part provides simulation of the FlexRay protocol services (according to the FlexRay Communications System Protocol Specification [5]) including clock synchronization, communication (static and dynamic segment, symbol window, network idle time) and frame construction and distribution.

AUTOSAR Software Interface. For communication between a host application and the simulated communication controller(s), SIDERA provides implementations of the AUTOSAR FlexRay Interface and of the AUTOSAR FlexRay Device Driver Interface, respectively (according to the AUTOSAR Standards [6] and [7]).

4. APPLICATION EXAMPLE: DEVELOPMENT OF AN ALGORITHM FOR SENSOR COMMUNICATION

This section presents an extensive application example to illustrate the concepts presented. SIDERA has been used for development and test of software modules for an automotive application, a driving dynamics control unit.

4.1 System Architecture

The aim of the project was the development of ECU firmware software modules for a driving dynamics controller application. Figure 8 shows the project relevant hardware components of the system architecture.

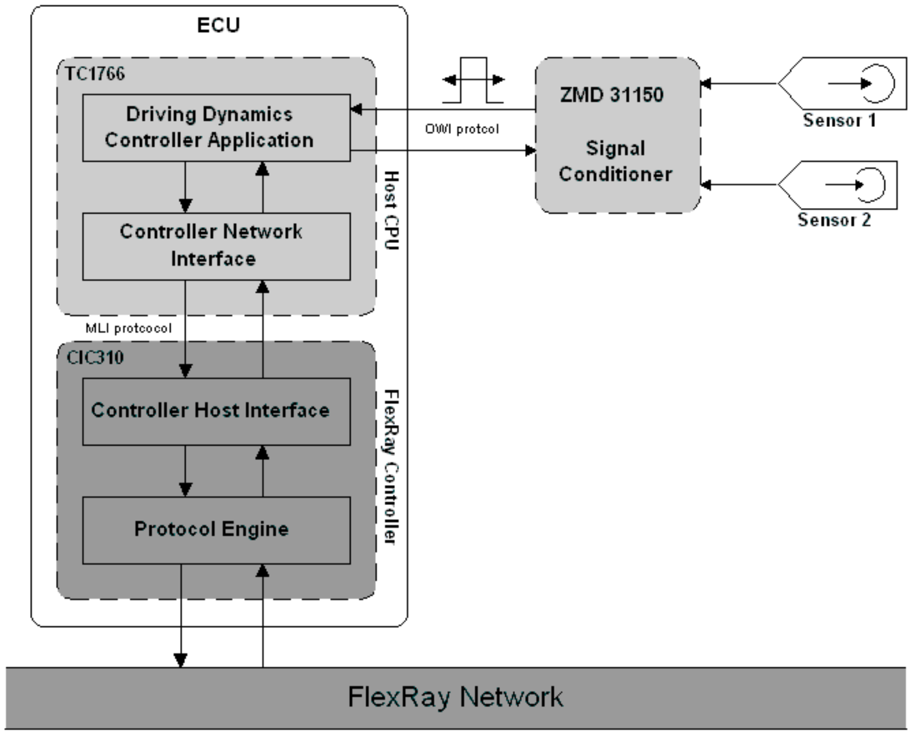


Figure 8. Target System Components

The host CPU is a TC1766 [1] 32-bit TriCore^{TM} based microcontroller for automotive applications from Infineon. The FlexRay Communication Controller is a CIC310 [2], also from Infineon, supposed for interconnection with the AUDO NG-32 bit microcontroller device family, where the TC1766 belongs to. The connection between the TC1766 and the CIC310 is done via theMLI (micro link serial bus) interface, a proprietary pin-based serial communication inferface between two Infineon microcontrollers.

Finally, a ZMD31150 sensor signal conditioner [3], delivering lateral acceleration data from a pressure sensor, is connected to two port pins of the TC1766 host CPU; one output port pin is for sending commands to the ZMD31150 and one input port pin is for receiving data from the ZMD31150. The TC1766 and the ZMD31150 communicate via a serial digital one-wire interface using pulse-width modulation (PWM) for bit encoding/decoding.

4.2 Application Description

When the ignition key is turned and the car is started, the host application performs a series of hardware checks. One of these checks verifies functionality and identity of two pressure sensors. For this purpose, the values of 3 registers, each of 16 bit length, are read from the sensors via a serial digital interface provided by the ZMD31150 signal conditioner. The information contained in these registers unambiguously identifies the type and is referred to as the *fingerprint* of the sensor. The fingerprint information is used to verify the functionality and integrity of the sensor and shall ensure that only properly working sensors of appropriate type are used (e.g. when a sensor is replaced in the course of car maintenance). Both operability and integrity of the pressure sensors are essential for proper operation of the driving dynamics controller. In case of a failed check of one or both sensors, the driving dynamics control unit shall be deactivated and the driver shall be informed by an indication on the car dashboard.

The fingerprint reading process comprises the following steps:

1. The host initializes the ZMD31150 for reading from the sensor.
2. A 16bit CRC is read from the sensor.
3. The values of three 16bit registers are read from the sensor.
4. A checksum is calculated at the host and compared to the CRC retrieved from the sensor.
5. If the calculated CRC equals the retrieved CRC, the integrity check was successful; else, the procedure is repeated starting with Step 1.
6. If fingerprint reading is not successful after 3 attempts, sensor integrity verification fails.

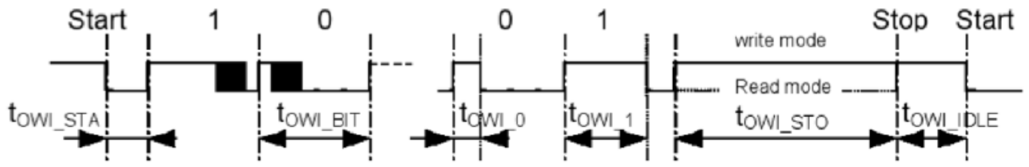
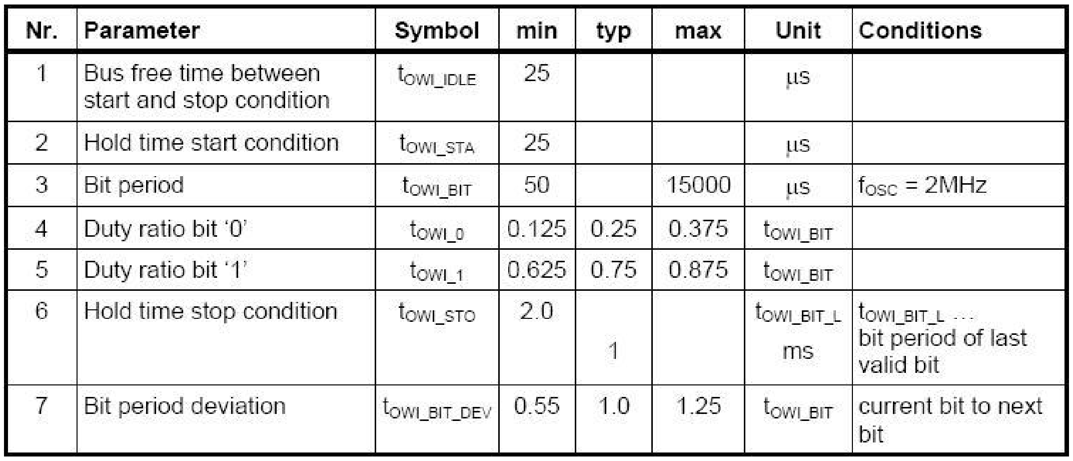


Figure 9. OWI Protocol

Communication between the Host-CPU and the pressure sensors takes place via a serial digital one wire interface (OWI) using PWM for bit encoding/decoding with a bit duration t_{OWI_BIT} of 2000µs. Figure 9 shows the bit encoding scheme. Each bit starts with a transition from low to high (rising edge) of the line signal. The *duty ratio* is the duration between a rising edge and a falling edge of the line signal related to the bit duration. According to Figure 9, the duty ratios are ¼ (500µs) for representation of a logical ”0” and ¾ (1500µs) for a logical ”1”, respectively.

4.3 Algorithm Analysis

For correct bit decoding, the line signal must be sampled with a sample period d_{sample} that is lower than the minimum duration between any two changes of the line signal in case of an undisturbed line. Transient disturbances of the line signal are disregarded; they will reliably be detected by the subsequent CRC calculation.



Nr.	Parameter	Symbol	min	typ	max	Unit	Conditions
1	Bus free time between start and stop condition	t_{OWI_IDLE}	25			µs	
2	Hold time start condition	t_{OWI_STA}	25			µs	
3	Bit period	t_{OWI_BIT}	50		15000	µs	f_{OSC} = 2MHz
4	Duty ratio bit '0'	t_{OWI_0}	0.125	0.25	0.375	t_{OWI_BIT}	
5	Duty ratio bit '1'	t_{OWI_1}	0.625	0.75	0.875	t_{OWI_BIT}	
6	Hold time stop condition	t_{OWI_STO}	2.0	1		$t_{OWI\_BIT\_L}$ ms	$t_{OWI\_BIT\_L}$ ... bit period of last valid bit
7	Bit period deviation	$t_{OWI\_BIT\_DEV}$	0.55	1.0	1.25	t_{OWI_BIT}	current bit to next bit

Figure 10. OWI Timing Characteristics

For determination of the sample period d_{sample}, two factors have to be taken into account:

- Figure 10 shows the OWI timing characteristis. It can be seen that the effective bit period duration may differ between 55% and 125% from the nominal bit duration. Further, the effective duty ratios may also differ between 87,5% and 112,5% from the Further, the effective duty ratios may also differ between 87,5% and 112,5% from the nominal duty ratios. These deviations account for variations in the manufacturing process of the signal conditioner as well as for variations in environmental conditions during operation, e.g. changes in the ambient temperature.
- Line signal sampling is done in the context of a periodic task. The jitter imposed by the operating system with regard to task activation is 50µs.

Which sample period d_{sample} is necessary and sufficient to guarantee correct results (in the absence of line faults)? The following calculation answers this question:

$$\mathbf{d_{sample} < 190\mu s}$$

= 2000	[bit duration in µs]
× 0,25	[duty ratio bit 0]
× 0,55	[bit period deviation]
× 0,875	[min. duty ratio bit 0]
−50	[task activation jitter in µs]

The above calculation determines the worst case scenario where the operational tolerances of all variables are pushed to their limits: minimum bit period duration and minimum duty ratio in the OWI timing characteristis coincide with a minimum task invocation period. If all operational tolerances are observed, correct operation of the fingerprint reading mechanism can be guaranteed for a sample period **d_{sample} < 190µs**.

4.4 Algorithm Implementation

To implement and test the algorithm in the simulation environment, three steps were necessary:

1. A model of the ZMD31150 had to be implemented, needed for subsequent implementation of the fingerprint mechanism in the host application.
2. The fingerprint mechanism had to be integrated into the simulation environment.
3. The OWI protocol had to be implemented.

Implementation of a model of the ZMD31150 signal conditioner. A model of the ZMD31150 was implemented based on the datasheet and the functional description provided by the manufacturer [3]. The therein described behaviour of the ZMD31150 was mapped to a state machine that covered all operational states necessary for implementation of the fingerprint reading mechanism in the host application.

Integration of the fingerprint module into the simulation environment. The fingerprint reading mechanism is a software module that is linked to the host application. It contains an entry function *fpEntry* that is periodically called by the host application (i.e. by the operating system scheduler) every d_{sample} µs. This function triggers the processing of a state machine handling the fingerprint reading process (see Section 4.2). For integration and testing, it is sufficient to periodically execute the *fpEntry* function from within the simulation environment. The task activation jitter imposed by the operating system is taken into account by configurable variation of the execution period dsample within the tolerance interval of 50µs.

Connection of the fingerprint module to the ZMD31150 model. Finally, the OWI protocol was implemented both in the model of the ZMD31150 and the fingerprint module. The implementation is based on the OWI protocol definition contained in the functional description of the signal conditioner. The possible deviations for the effective bit period duration and the effective bit duty ratios, respectively (see Figure 10), are taken into account by configurable variation of both the bit duration and the bit duty ratios within the specified tolerance regions. The port pins used for communication between the TC1766 host CPU and the ZMD31150 are modelled by two global variables, ”PIN” and ”POUT”, having a value of ”1” if the line signal is “high” and ”0” if the line signal is ”low”. The ZMD31150 model writes to the ”PIN” variable (the input pin of the TC1766), the fingerprint mechanism writes to the ”POUT” variable (the output pin of the TC1766), according to the OWI protocol timing characteristics (see Figure 10).

The manpower provided for the implementation and test of the algorithm was **one software developer**.

4.5 Testing the Algorithm

The testing procedure consists of several test cases that shall validate the correctness of the algorithm in the presence of

- *Variations of the OWI timing characteristics*
  To test the stability of the algorithm, the model was exposed to variations in the OWI timing characteristis as well as to variations in the activation period of the *fpEntry* function.

- *Transient signal line faults*
  To validate the error detection and recovery mechanisms of the algorithm, transient line faults were injected to check the proper functionality of the CRC calculation and the triggering of the retry mechanism. The line faults were simulated by manipulation of the ”PIN” and ”POUT” variables.

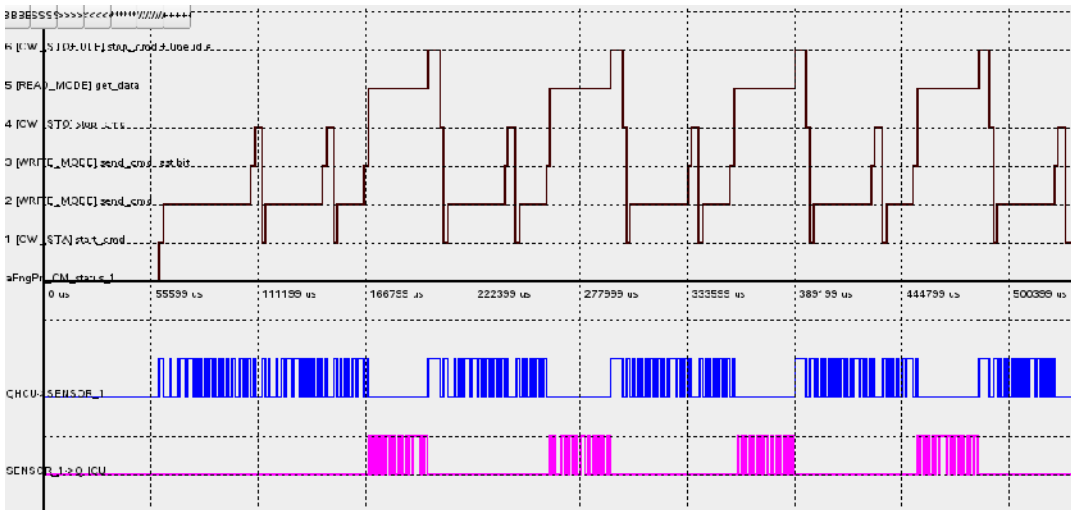


Figure 11. Successful Fingerprint Reading Operation

Figure 11 shows the visualization of a successful reading operation. The x-Axis denotes the progression of time. The upper half of Figure 11 shows the different state machine transitions of the algorithm in the course of the reading process. In the lower half, the communication between the host and the signal conditioner is shown, whereas the upper bitstream shows the commands from the TC1766 and the lower bitstream shows the responses from the ZMD31150.

4.6 Effective Benefit of Simulation Approach

The application example demonstrated the principle of operation using a practical example. However, before deploying a simulation based software development approach, the following question arises:

When does it make sense to switch to a simulation environment for software development?
The portation of (parts of) the application software to a simulation environment means costs in terms of time. These costs have to be compensated by winnings in software development time.

But how can these winnings be quantified? As already outlined in Section 2, a simulation environment provides some benefits like the possibility to develop application software without needing target hardware, but these benefits are difficult to express in terms of time savings. The only legitimate criterion to assess the usability of a simulation based approach in a given context is the time saving in building and loading.

For the subsequent analysis, we will use the following parameters:

- d^{prog}_{sim}: The development effort of the program in the simulation environment.
- d^{prog}_{HW}: The development effort of the program in the hardware environment.
- d^{prep}_{sim}: The effort to prepare the software development process for the simulation environment.
- d^{build}_{HW}: The average duration of a build and load phase in the hardware environment.
- d^{build}_{sim}: The average duration of a build and load phase in the simulation environment.
- d^{code}: The estimated (average) duration of a coding phase followed by a build and load phase. This is an empirical value dependant on the coding style and therefore pretty sure different for different programmers.

We assume that $d^{build}_{sim} << d^{build}_{HW}$ unless otherwise it would not make sense to use a simulation environment. We further assume that the development process is constituted by *development cycles DC*. A DC consists of a *coding phase* and a *build and load phase*. The duration of a development cycle d^{DC}_{sim} in the simulation environment and the duration of a development cycle d^{DC}_{HW} in the in hardware environment, respectively, equals

$$\begin{aligned} d^{DC}_{sim} &= d^{code} + d^{build}_{sim} \\ d^{DC}_{HW} &= d^{code} + d^{build}_{HW} \end{aligned} \qquad (1)$$

The overall duration of the development process is the sum of the durations of all development cycles. This calculated duration is of course shorter than the real duration of the development process, because we only consider the times of productivity here.

For the overall duration of the development process d^{prog}_{sim} in the simulation environment and d^{prog}_{HW} in the hardware environment, respectively, we get

$$\begin{aligned} d^{prog}_{sim} &= \text{n x } d^{DC}_{sim} \\ d^{prog}_{HW} &= \text{n x } d^{DC}_{HW} \end{aligned} \qquad (2)$$

where n is the overall number of development cycles required for the development of the program.
The use of a simulation environment pays if the effort induced by the use of the simulation environment for development of a given software component is smaller than the development effort for this software component if only the hardware environment is used:

$$d^{prog}_{sim} + d^{prep}_{sim} < d^{prog}_{HW} \qquad (3)$$

The *effective benefit EB* in development time saving for a given software component equals the development effort for this software component if only the hardware environment is used minus the effort induced by the use of the simulation environment for the development of this software component:

$$EB = d^{prog}_{HW} - d^{prog}_{sim} - d^{prep}_{sim} \qquad (4)$$

The use of a simulation environment pays if $EB > 0$.

4.7 Application Example Evaluation

We now evaluate the effective benefit of the application of the simulation environment in the course of the development of the fingerprint reading algorithm application example presented in the last section.

To determine the number of development cycles n (see Section 4.6, Equation 2), we used a batch script that invokes the build process and that increases a counter with each invocation of the script. Using this method, we were able to determine the total number of development cycles n in the simulation environment:

$n = 771$
To determine the overall duration of the development process d^{prog}_{sim} in the simulation environment, we need the following parameters which have been observed and measured during the case study:

$d^{code} \approx 15min$
$d^{build}_{sim} \approx 0{,}5min$
For d^{prog}_{sim}, it follows from Equations (1) and (2):
$d^{prog}_{sim} = \text{n x } d^{DC}_{sim} = n \text{ x } (d^{code} + d^{build}_{sim}) \approx 771 \text{ x } (15+0{,}5) \approx 11950min \approx 199h$

To estimate the overall duration of the development process d^{prog}_{HW} in the hardware environment, we need the average duration of a build and load phase in the hardware environment d^{build}_{HW}. This duration has been determined by measuring the build and load time of another software module of similar size as a reference:

$d^{build}_{HW} \approx 10min$
For d^{prog}_{HW}, it follows from Equations (1) and (2):
$d^{prog}_{HW} = \text{n x } d^{DC}_{HW} = n \text{ x } (d^{code} + d^{build}_{HW}) \approx 771 \text{ x } (15+10) \approx 19275min \approx 321h$

Finally, we have to consider the effort to prepare the software development process for the simulation environment:

$d^{prep}_{sim} \approx 80h$

Now we have all information needed for determination of the effective benefit of the simulation based development procedure according to Equation (4):

$$EB = d^{prog}_{HW} - d^{prog}_{sim} - d^{prep}_{sim} \approx 321 - 199 - 80 \approx 42h$$

For the simulation based development of the fingerprint algorithm, the time saving is about 13% compared to the development time in the hardware environment. The actual effective benefit is presumably higher because build and load process acceleration is not the only positive implication of using the simulation environment for the software development process (see Section 2).

5. DISCUSSION AND CONCLUSION

It may not make sense to portate the host application software to a simulation environment as a whole. However, if the host application is composed of a set of single modules that fulfill specified functions, the development of single modules in the simulation environment may make sense.

In the presented case study, a time saving of 13% of the overall software development time could be achieved. However, this case study involved only one software developer who did the whole work consisting of two steps:

1. Preparation of the simulation environment for implementation of the algorithm
2. Implementation and test of the algorithm

This example shows that even for a one-person-show a considerable time saving can be achieved using a simulation based development approach. If more developers are involved in Step 2 (Implementation and test of the algorithm), the overall time saving increases, because the effort for Step 1 has only to be spent once and is of benefit for all developers.

Howver, the portation of host application software modules to a simulation environment as well as the creation of simulation models of existing hardware components increases the development effort. Obviously, this additional effort has to be justified by some benefits. Possible benefits are

Acceleration of the software development process. Single modules can be developed and tested in isolation using the simulation environment. If a software module is developed from scratch, it makes sense to validate the correctness of the control flow of the software in the simulation environment and to switch to the hardware environment for development of the hardware related parts.

Development without hardware components. Hardware components are not yet available when the project starts but are delivered by the manufacturerat a later point in time. This situation typically occurs in prototype development projects, where innovative hardware components are integrated. Given that the interface to the hardware component is specified, it may be useful to create a model of the component and to develop the host application using the model until the hardware component becomes available.

Improved capabilites for analysis. Based on experience, debugging in a distributed environment can be a wicked problem. For instance, analyzing the communication with some remote component may be hard because in the hardware environment this component usually is not in the sphere of control of the debugger. If the host application stops at a breakpoint, the remote component continues operation making it hard to pinpoint possible sources of error in the host application. Modelling the behaviour of the remote component on the base of its interface description and integrating the model into the simulation enviornment may help to locate and correct bugs in the host application.

Reduced costs for test equipment. Dedicated hardware equipment for testing distributed systems is quite expensive. In automotive applications, for instance, testing the communication behaviour of a single ECU in isolation requires special bus simulators. These simulators are programmable hardware components that are connected to the ECU and that simulate the bus communication according to the cluster communication schedule contained in the FIBEX file. The message contents of the simulated communication partners (the other nodes) have to be defined by the tester. Having a simulation model of the communication system at hand reduces the need for hardware bus simulators. Modelling the communication partners in the simulation environment is no remarkable additional effort (the messages sent have to be defined anyway).

Acceleration of the integration process. The integration phase of an ECU into a communication network usually is a process of revealing and correcting errors. A thorough analysis of host application software components in the simulation environment prior to integration may help to diminish the number of errors. Further, the correction of errors may be supported by the improved capabilites for analysis provided by the simulation environment, reducing the overall integration effort.

REFERENCES

[1] Infineon Technologies AG. TC1766 Highly Integrated 32-bit TriCore(TM)-based Next Generation Microcontroller for Automotive Applications. Product Brief, Infineon Technolo- gies AG, 2005.

[2] Infineon Technologies AG. SAK-CIC 310-OSMX2HT FlexRay Communication Controller. Data Sheet, Infineon Technologies AG, 2007.

[3] ZMD AG. ZMD31150 Fast Automotive Sensor Signal Conditioner Data Sheet Rev. 0.7. Data Sheet, ZMD AG, 2006.

[4] ASAM e.V. FIBEX - Field Bus Data Exchange Format. Standard, Association for Standardisation of Automation and Measuring Systems (ASAM e.V.), 2006.

[5] Flexray. FlexRay Communications System Protocol Specification Version 2.1. Specification, FlexRay Consortium, 2005.

[6] AUTOSAR GbR. Specification of FlexRay Driver V.2.2.1. Standard, AUTOSAR GbR, 2008.

[7] AUTOSARGbR. Specification of FlexRay Interface V.3.0.2. Standard, AUTOSAR GbR, 2008.

[8] A. Hanzlik. Investigation of Fault-Tolerant Multi-Cluster Clock Synchronization Strategies by Means of Simulation. PhD Thesis, Vienna University of Technology, Institute for Computer Engineering, Vienna, Austria, 2004.

[9] A. Hanzlik. SIDERA - a Simulation Model for Time- Triggered Distributed Real-Time Systems. International Review on Computers and Software (IRECOS), Vol. 1(3):181–193, November 2006.

[10] A. Hanzlik. Stability and Performance Analysis of Clock Synchronization in FlexRay. International Review on Computers and Software (IRECOS), Vol. 1(2):146–155, September 2006.

[11] H. Kopetz, A. Ademaj, P. Grillinger, and K. Steinhammer. The Time-Triggered Ethernet (TTE) design. Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing 2005 (ISORC 2005), pp 22– 23, May 2005.

[12] Hermann Kopetz, Astrit Ademaj, and Alexander Hanzlik. Integration of Internal and External Clock Synchronization by Combination of the Clock-State and Clock-Rate Correction in Fault-Tolerant Distributed Systems. 25th IEEE International Real-Time Systems Symposium (RTSS'04), pp 415–425, 2004.

[13] H. Kopetz, Günther Bauer. The Time-Triggered Architecture. Proceedings of the IEEE, Vol. 91(1):112 – 126, January 2003.

AUTHOR

Alexander Hanzlik holds an Msc degree in Computer Science and a PhD degree in Computer Engineering from the Vienna University of Technology as well as an MBA degree from the PEF Private University of Management, Vienna. His main research interests are Simulation, Fault-tolerant Computing and Distributed Real-Time Systems. He has about 20 years professional experience in software development, system design and project management. He is author and co-author of more than 20 scientific papers.

Currently, Alexander Hanzlik is under contract at the AIT Austrian Institute of Technology, where he is concerned with design and simulation of distributed electronic control systems for automotive applications.