AUTOSAR Software Architecture

Robert Warschofsky

Hasso-Plattner-Institute für Softwaresystemtechnik

Abstract. AUTOSAR supports the re-use of software and hardware components of automotive electronic systems. Therefore, amongst other things, AUTOSAR defines a software architecture that is used to decouple software components from hardware devices. This paper gives an overview about the different layers of that architecture. In addition, the upper most layer that concerns the application specific part of automotive electronic systems is presented.

1 Introduction

Many goals have to be reached to create automotive systems on a competitive base, in today's automotive electronics. For example, the increasing number of network components leads to a level of complexity that is difficult to handle using traditional development processes. Additionally more and more resources have to be spend on adapting already existing solutions to different environments. But the main drivers in in automotive electronics are the reduction of hardware costs and the implementation of new features [1].

One approach to cope with these problems is the standardization of the AUTomotive Open Systems Architecture (AUTOSAR). AUTOSAR is a framework to create electronic automotive control systems. With that framework comes a layered software architecture that supports the software developer to deal with different environments and complex network structures.

1.1 Focus and overview

This paper introduces some aspects of the AUTOSAR software architecture in more detail than others. In Section 2 an overview over the architecture is given. The layers which are not in the focus of this paper are shortly described in that section, too.

Other papers describe these layers in more detail. For example, the *Runtime Environment* layer is handled in [2]. The functionalities of the communication stack are described in [3]. This communication stack uses different layers of the AUTOSAR software architecture to provide communication as described later.

The *Application layer* which is in the focus of this paper, is described in detail in Section 3. Additionally to the concepts of the *Application layer*, Section 3 also shortly describes how the software part of an AUTOSAR application is implemented.

Apart from the software architecture, Section 4 introduces the transformation from a designed AUTOSAR application to a running system. For such a transformation, the AUTOSAR framework provides a methodology that specifies the order of the activities for the development process of an AUTOSAR system. That methodology is described in detail in [4]. Finally, Section 5 summarizes the information given in this paper.

In addition to the software architecture, the AUTOSAR framework provides a methodology that specifies the order of the activities for the development process of an AUTOSAR system. That methodology is described in detail in [4].

The AUTOSAR framework further provides a UML Profile [5]. That profile restricts UML so that it can be used to model the AUTOSAR application during design time.

1.2 Example

To explain some concepts of the *AUTOSAR software architecture* in a less abstract way, a running example is used in this paper. That example comes from the *dSpace SystemDesk tutorial* [6] and covers a simple car direction indicator system. A schematic view of this system with the involved hardware components is shown in Figure 1. The car direction indicator system consists of two front

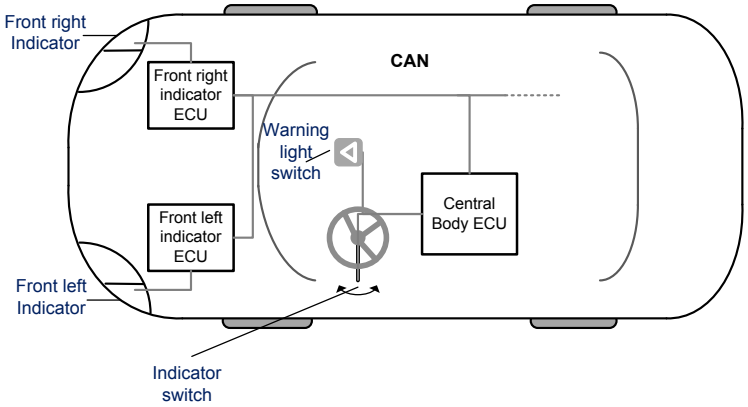

Fig. 1. Hardware components of the simple car direction indicator system example.

light indicators with corresponding ECUs and a central body ECU. The central body ECU has sensors for two switches, an indicator switch to activate either the left or the right indicator and a warning light switch to activate both, left and right, indicators at once. There also exists a CAN bus as one part of the vehicle network system.

The next Section describes the *AUTOSAR software architecture* and uses this car direction indicator system to explain the purpose of selected parts of the software architecture.

2 Software architecture of the AUTOSAR framework

The *AUTOSAR software architecture* is a layered architecture that has the goal to abstract from hardware components. Furthermore, these layers encapsulate functionality that can be used by all *AUTOSAR applications*. The architecture supports the realization of functional requirements, but it yet not fully supports non-functional requirements. Therefore, also the moddeling of real-time aspects of an application is not realized, yet. But there are projects which try to add the realization of such non-functional requirements to AUTOSAR. One project which deals with the timing issue is the *TIMMO Project* [7].

The software architecture shown in Figure 2 consists of four main layers. The lower most layer is the *Microcontroller layer* which contains the ECU hardware. The next two layers above are the *Basic Software layer* and the *Runtime Environment layer*. The upper most layer is the *Application layer* which contains all application specific software components (SWC).

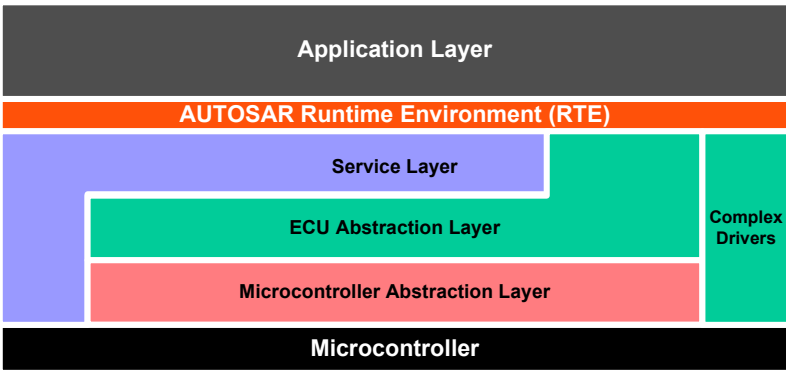

Fig. 2. Layers of the AUTOSAR software architecture

The layers have different responsibilities. The *Basic Software layer* and the *Runtime Environment layer* are responsible for the abstraction between the hardware and the application software. Therefore the *Basic Software layer* contains ECU specific modules as well as general AUTOSAR modules. The *Runtime Environment layer* enables the inter component communication as well as communication from software components to basic software modules.

The *Basic Software layer* can further be divided into three different layers, namely the *Service layer*, the *ECU Abstraction layer* and the *Microcontroller*

Abstraction layer. It may also contain *Complex Drivers*, which are described later.

The *Microcontroller Abstraction layer* uses drivers to abstract from specific controllers on the ECU. There are several kinds of microcontrollers, for example CAN-Bus microcontrollers. The drivers of the *Microcontroller Abstraction layer* provide interfaces to the *ECU Abstraction layer* to enable generalized usage of different microcontrollers of the same kind.

The *ECU Abstraction layer* abstracts from the location of the controller. For layers above the *ECU Abstraction layer* it is not necessary to know whether the controller is an *on chip* or *on device* controller.

The *Service layer* provide basic services for each AUTOSAR application. An AUTOSAR application can access these services through *standardized AUTOSAR interfaces*.

The *Basic Software layer* can also be divided into different stacks corresponding to the general functionality the basic software provides. These stacks that are orthogonal to the *Basic Software layer*s are shown in Figure 3.

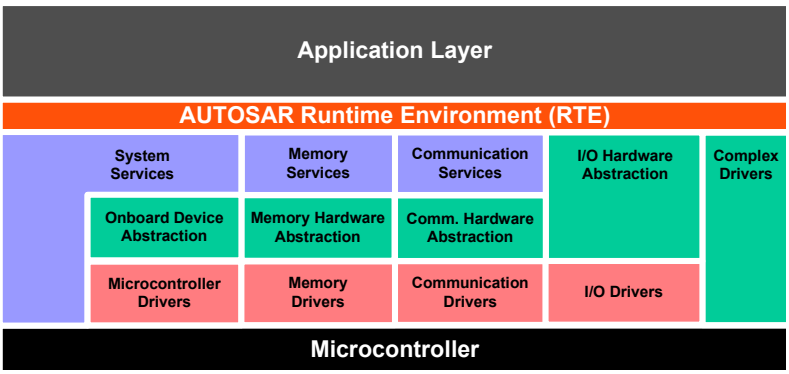

Fig. 3. Stacks of functionality in the *Basic Software layer* of the AUTOSAR software architecture

The *System stack*, consisting of *Microcontroller Drivers*, *Onboard Device Abstraction* and *System Services*, provides standardized services and library function for example for timer operations or operating system functionality. It also provides ECU specific services like ECU state management and watchdog management for hardware components.

The *Memory Management stack*, consisting of *Memory Drivers*, *Memory Hardware Abstraction* and *Memory Services*, provides standardized access to non volatile memory. Through this stack each software application component can allocate memory to maintain its internal state. The memory can be ECU internal or external, but because of the abstraction layers of the *Memory Management stack* the application component does not need to know whether it is

internal or external memory. The component only requests the memory through the standardized interface.

In the example of Section 1.2 the *AUTOSAR Software Component* on the central body ECU will use the *Memory Management stack* to save their operational state. A management component on that ECU may have to know which indicators are activated by which switches. Otherwise an activation of an indicator could be "forgotten". If for example both switches are active and the direction indicator switch is turned to none active, the corresponding direction indicator must not be deactivated, since the warning light switch remains still active.

The *Communication stack*, consisting of *Communication Drivers*, *Communication Hardware Abstraction* and *Communication Services*, provides standardized access to the vehicle network system. Through this stack software components can communicate with each other even if they are located on different ECUs. The components do not use the communication stack directly, but use the *Runtime Environment*. The *Runtime Environment* then manages the communication between the components corresponding to their location. Components on the same ECU communicate directly while for the communication of components on different ECUs the *Communication stack* is used. A more detailed description of the *Communication stack* is given in [3].

In the direction indicator system from Section 1.2 the *Communication stack* has to be used, if for example a software component on the central body ECU communicates with a software component on one of the direction indicator ECUs. The *Communication stack* encapsulates the messages between these two components and uses the CAN-Bus to send the messages between the two corresponding ECUs.

The *I/O stack*, consisting of *I/O Drivers* and *I/O Hardware Abstraction*, provides standardized access to sensors, actuators and other ECU on board peripherals. This stack does not have a service layer since there is no general interface for all possible sensors and actuators. Therefore special software components are required to access sensors and actuator, named *Sensor/Actuator Software Component*. These components are described in more detail in Section 3.

In the direction indicator system from Section 1.2 there exist sensors for the two switches on the central body ECU which have to be read by some *Sensor Software Components* to get their state. On the indicator ECUs there are actuators for the direction indicators to activate and deactivate them. These actuators need also *Actuator Software Component* to control them.

The *Complex Driver stack* is not a real stack, but enables it to bypass the *Basic Software layer* abstraction. This can be useful for application components which need direct access to the hardware devices on the ECU for performance reasons. Such applications are for example injection control or electronic valve control applications. These complex drivers are not provided by AUTOSAR but each manufacturer which needs those drivers has to implement them itself.

As already mentioned, this paper can not describe all BSW modules in detail, since it is focused on the application layer of the AUTOSAR software architecture. More information about the single modules in the *Communication stack* are given in [3], the RTE is described in detail in [2] and [8] describes many BSW modules in detail.

3 Application layer

The AUTOSAR software architecture enables the software developer to design an AUTOSAR application almost independent from the involved hardware. There is no knowledge about the network required, since the AUTOSAR software architecture and especially the RTE hide the network from the application. There is also nearly no knowledge about the used ECUs required, since the software architecture abstracts from the specific ECU and the controller on it. But there is of cause knowledge required about the sensors and actuators of an ECU which are used in the specific AUTOSAR application, so the software development can not be completely independent from the existent hardware components.

This section describes the important parts of the application layer of the AUTOSAR software architecture. These parts are *AUTOSAR software components*, the *AUTOSAR ports* of these components and the implementation of such components. The components and ports are modeled using the AUTOSAR UML Profil [5].

3.1 AUTOSAR Software Components

An application in AUTOSAR consists of interconnected *AUTOSAR Software Component*s. An *AUTOSAR Software Component* is an atomic piece of software that has to be deployed on one ECU. Such a component implements a part of an *AUTOSAR application*. To connect software components, each component has well defined ports through which the component can communicate with other components or with *Basic Software* modules.

There exists special kinds of *AUTOSAR Software Component*s. One kind is the *Sensor/Actuator Software Component*. A *Sensor Software Component* is responsible for reading a sensor and provide its data to other components. The reading of the sensor data is done through the *I/O stack* of the software architecture. An *Actuator Software Component* is responsible for setting the state of an actuator on an ECU. Therefore it can also provide an interface to other components, so that these components can initiate the state setting of the actuator.

Since both, the *Sensor Software Component* and the *Actuator Software Component*, have to use special sensors/actuators on an ECU, these components naturally have to be deployed on an ECU with a corresponding sensor/actuator. Therefore the software designer at least need to know which hardware sensors and actuators are used in the software application to design the *AUTOSAR application*.

Another kind of special *AUTOSAR Software Component* is the *Composite AUTOSAR Software Component*. A composite component is a logical interconnection of other component, either atomic or again composite. These components are called *prototypes*. The prototypes of a composite component do not need to be deployed on the same ECU but can be distributed over several ECUs.

An composite component can also have ports. These ports are mapped to ports of the prototypes of the composite component, since the component itself does not provide any additional application logic. The application itself is also represented as a special composite component that contains all software components of that application. There are no ports on the application component, since its prototypes are only interconnected, but not connected to components outside the application.

3.2 AUTOSAR Ports

A Port belongs to exactly one *AUTOSAR Software Component* and represents a point of interaction of that component. The kind of interaction that a specific port provides, is defined by the *AUTOSAR Interface* that is provided or required by that port. A Port can either provide the service of an *AUTOSAR Interface*, making it a *PPort*, or require the service of an *AUTOSAR Interface*, making it an *RPort*.

If the interface that is encapsulated by a port is provided by a module of the *AUTOSAR Service layer* (see Figure 2), it has to be a *Standardized AUTOSAR Interface*. Such an interface can be used by each *AUTOSAR Software Component* of any AUTOSAR application. If the interface is provided by an *AUTOSAR Software Component* or a module of the *I/O Hardware Abstraction* (see Figure 3), it is an *AUTOSAR Interface*. *AUTOSAR Software Component*s can only request *AUTOSAR Interfaces* or *Standardized AUTOSAR Interfaces*, since only these can be addressed through the *Runtime Environment* and that is the only way, components can communicate, as it is explained later.

An *AUTOSAR Interfaces* as well as an *Standardized AUTOSAR Interfaces* can be one of three kinds. First, it can be a *Client-Server interface*. A call to a method of such an interface can be either blocking, which denotes the communication of client and server is synchronous, or non-blocking, which means asynchronous communication. In either case the client awaits a response from the server.

The second kind of interface is a *Sender-Receiver interface*. Using such an interface, all calls are non-blocking calls and the client will never get a response.

The third kind of interface is the *Configuration interface*. This interface enables a *AUTOSAR Software Component* to receive values for configuration parameters.

During the application and component design, the later distribution of the components over the ECUs is not relevant. Important is only that *Sensor/Actuator Software Components* require one or several ECUs with the corresponding sensors/actuators. Since the communication between the components already

has to be defined during the design phase, in that phase the components communicate through the *Virtual Function Bus* (VFB). More details about the VFB are given in [2] and [9].

3.3 Design of a Direction Indicator System

Using several AUTOSAR components with AUTOSAR ports to interconnect them, the car direction indicator system from Section 1.2 can be created like the one shown in Figure 4.

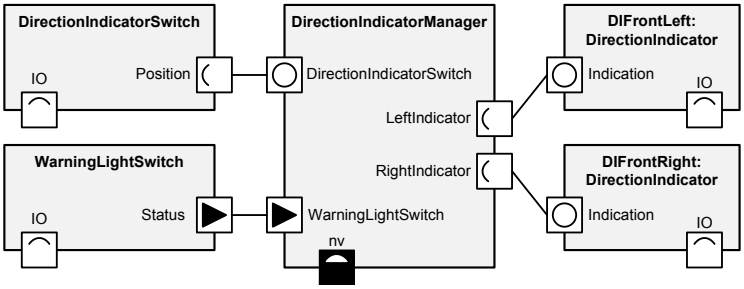

Fig. 4. Design of a car direction indicator system consisting a main managing component and four *Sensor/Actuator Software Components* interconnected through their ports.

This system contains two *Actuator Software Components* for the two direction indicators. These two components use their IO labeled port to set the state of their direction indicator and offer a service to set this state through their Indication labeled port.

The system also contains two *Sensor Software Components* to read the state of the two switches. The reading of the switch state is done through the IO labeled ports. The *Direction Indicator Switch* component send the state of the switch to a service, while the *Warning Light Switch* component uses a *Sender-Receiver* communication to send the state of the switch to another component.

The last component of Figure 4 is the managing component. That component provides a service to get the information about the switch of the *Direction Indicator Switch* component. It also has a receiver for the information from the *Warning Light Switch* component. The information from both sources have to be evaluated and the corresponding direction indicators have to be activated or deactivated. For the latter purpose, the component uses the services provided by the two *Actuator Software Components*.

All the so far mentioned ports encapsulate *AUTOSAR Interfaces*, since these interfaces are provided by *AUTOSAR Software Component*s. The managing component additionally has another port, labeled nv. This port encapsulates a *Standardized AUTOSAR Interface*. That the service part of that interface is

provided an AUTOSAR module of the communication service (see Figure 3) which is used to allocate memory for the component to maintain its internal state.

The design of that system can be done without detailed knowledge about the later used ECUs. The only requirement is, that there have to be sensors on an ECU for the two switches and actuator for the two direction indicators. Again, the communication between the components is designed only virtual by using the VFB.

To create a real application from this design two steps are necessary. In the first place, the components need a behavior, an executable part. This step is described in the next section. After that the components have to be deployed on ECUs where the communication will no longer go through the VFB. This transformation is described in Section 4.

3.4 Internal Behavior of AUTOSAR Software Components

The executable part of an *AUTOSAR Software Component* is provided by the implementation of *Runnables*. *Runnables* are for example functions in a programming language like C, or compiled *MATLAB/Simulink* models.

Since the unit of execution in AUTOSAR is an *Operating System task* (OS task), all *Runnables* must be mapped to such an OS task to be executed. This mapping is done during the configuration of an ECU which is described in more detail in the AUTOSAR methodology [10] and in [4].

An OS task may contain only one *Runnable* or a sequence of *Runnables*. The execution of a *Runnable* inside a task can furthermore depend on a special event or some timing constrains. The concept of *Runnables* and *Operating System tasks* is described in more detail [2].

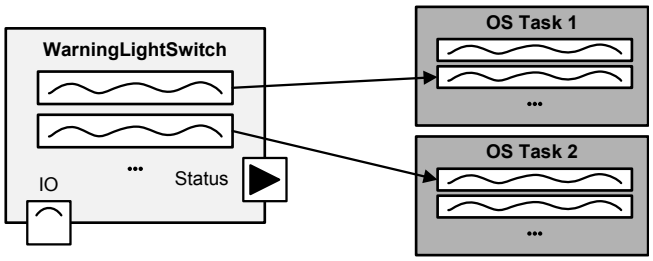

Fig. 5. Exemplary mapping of *Runnables* to *Operating System tasks*.

There is no need to map all *Runnables* of one component to the same OS task. It may on the contrary be useful to distribute the *Runnables* of one component to different OS tasks, since the *Runnables* have to be executed concurrently. An exemplary mapping of *Runnables* to OS tasks is shown in Figure 5.

4 Transformation to a running system

As already mentioned, during design time of an AUTOSAR application the VFB is used to model the communication between software components. For the single software components it is not important, where the other components are located, since all components are connected through this VFB.

During runtime this concept of abstraction for the locality should be retained. Therefore a runtime representation of the VFB is needed. Additionally to communication management between software components, this runtime representation must handle the use of several ECUs. That means, it must know, which software components are on which ECU and which components communicate with each other.

To satisfy all these requirements the runtime representation, the *Runtime Environment*, is generated uniquely for each ECU. After the generation step, the *Runtime Environment* of an ECU knows exactly which components are on this ECU and which are not. If there has to be a communication between components on different ECUs, the *Runtime Environment* takes the corresponding messages and uses the *Communication stack* (see Figure 3) to send these messages to the corresponding ECU. It also takes messages from the *Communication stack* and sends them to the receiver component on its ECU.

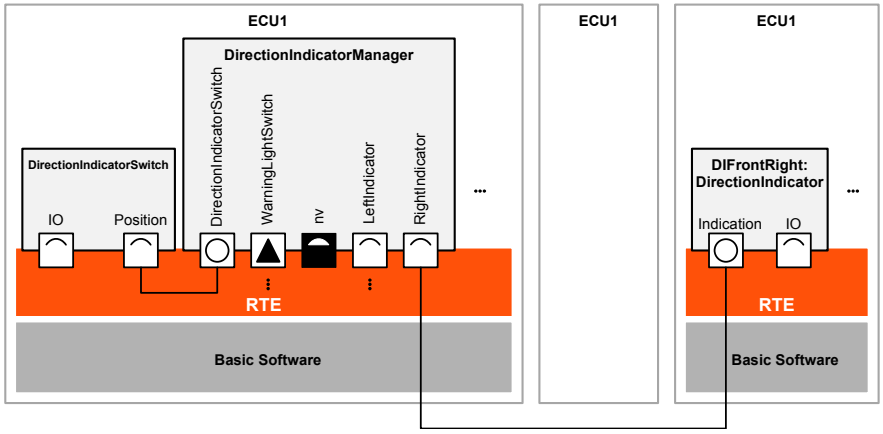

Fig. 6. Communication between *AUTOSAR Software Component* on the same and on different ECUs, supported by the corresponding *Runtime Environment* of each ECU.

For the example from Section 1.2 the generated *Runtime Environment* for the central body ECU will know, that the two *Sensor Software Components* and the managing component are deployed on this ECU. The communication between these components can be realized as direct function call. The *Runtime Environment* will further know, that the managing component has to commu-

nicate with the two *Actuator Software Components* on the two indicator ECUs. This communication will be automatically directed through the *Communication stack*. The different communication paths are shown in Figure 6.

A more detailed description of the generation of a *Runtime Environment* is also given in [2].

5 Summary

With the AUTOSAR software architecture the software part of an AUTOSAR application can be developed as independent as possible from the hardware parts. The software architecture enables this with two main abstraction layers, namely the *Runtime Environment layer* and the *Basic Software layer*.

The *Basic Software layer* additionally provides services that can be used by each AUTOSAR application. Through that, frequently used functionality is also encapsulated and provided by the AUTOSAR software architecture.

On the *Application layer* the *AUTOSAR Software Component*s are using *Ports* that encapsulate interfaces to guarantee type safety during the communication between these components. The communication itself is handled by the *Virtual Function Bus* (VFB).

The VFB as specification is represented during runtime by the *Runtime Environment*, which is generated uniquely for each ECU in the AUTOSAR system. This concept also supports the creation of nearly hardware independent software in the automotive industry and therefor reduces the complexity of the resulting systems.

References

[1] Fennel, H., Bunzel, S., Heinecke, H., Bielefeld, J., Fürst, S., Schnelle, K.P., Grote, W., Maldener, N., Weber, T., Wohlgemuth, F., Ruh, J., Lundh, L., Sandén, T., Heitkämper, P., Rimkus, R., Leflour, J., Gilberg, A., Virnich, U., Voget, S., Nishikawa, K., Kajio, K., Lange, K., Scharnhorst, T., Kunkel, B.: Achievements and exploitation of the AUTOSAR development partnership. CTEA (2006)

[2] Naumann, N.: AUTOSAR Runtime Environment and Virtual Function Bus. Technical report, Hasso-Plattner-Institut für Softwaresystemtechnik (2009)

[3] Gosda, J.: AUTOSAR Communication Stack. Technical report, Hasso-Plattner-Institut für Softwaresystemtechnik (2009)

[4] Hebig, R.: Methodology and Templates in AUTOSAR. Technical report, Hasso-Plattner-Institut für Softwaresystemtechnik (2009)

[5] AUTOSAR GbR: UML Profile for AUTOSAR V1.0.1. Online (2006), `www.autosar.org/download/AUTOSAR_UML_Profile.pdf`

[6] dSPACE: System Desk Tutorial. (2008), `http://www.dspace.com/ww/en/pub/home/products/sw/system_architecture_software/systemdesk.cfm`

[7] Timmo: Timing model. Online, `https://www.timmo.org/`

[8] AUTOSAR GbR: AUTOSAR Layered Software Architecture. Online (August, 15th 2008), `http://www.autosar.org/download/specs_aktuell/AUTOSAR_LayeredSoftwareArchitecture.pdf`

[9] AUTOSAR GbR: Specification of the Virtual Functional Bus. Online (2008), http://www.autosar.org/download/specs_aktuell/AUTOSAR_SWS_VFB.pdf
[10] AUTOSAR GbR: Specification of ECU Configuration V2.0.2 (2008), http://www.autosar.org/download/specs_aktuell/AUTOSAR_ECU_Configuration.pdf