# Beyond AUTOSAR – Optimized AUTOSAR Compliant Basic Software Modules

Thomas M. Galla
Elektrobit Austria GmbH
Stumpergasse 48 / 28, A-1060 Vienna, Austria
Email: thomas.galla@elektrobit.com

**Abstract**

The number of electronic systems in cars is continuously growing. Electronic systems, consisting of so-called electronic control units (ECUs) interconnected by a communication network, account for up to 30% of a modern car's worth. Consequently, software plays an ever more important role, both for the implementation of functions and the infrastructure.

In order to benefit from the reuse of software modules, the major automotive companies have standardized a large number of these modules in the context of the AUTOSAR consortium in form of a layered architecture of software modules.

We investigate and discuss the optimization potentials in this layered software architecture, namely making use of dedicated hardware units (e.g., DMA controller, co-processors) in the module implementation to reduce execution time and code size, providing means to tailor the modules of the layered software architecture to the specific application to reduce the memory footprint, and combining adjacent modules to reduce memory footprint and execution time while still remaining conforming to the AUTOSAR API at the "outer" interfaces.

We demonstrate the feasibility of this approach by focusing on an optimized version of FlexRay specific modules implemented on two different hardware platforms. Finally we show the impact of these optimizations in terms of execution time as well as the memory consumption of the optimized modules by comparing these measures to the measures obtained from the corresponding non-optimized AUTOSAR modules.

## 1 Introduction

In the last decade the percentage of electronic components in today's cars has been ever increasing. According to [6] the S-Class Mercedes for example utilizes seven communication buses and 72 micro controllers.

Since 1993 major automotive companies have been striving for the deployment of standard software modules in their applications since the potential benefits of using standard software modules are huge [5]. This trend has been a key motivation for the formation of the AUTOSAR [3] consortium in 2003. Important issues in this context are safety (increased test depth of standard software modules), software reuse, for the possibility to combine software modules supplied by different vendors due to standardized interfaces, and last but not least cost reasons in order to cope with ever reducing development cycles.

The software stack proposed by AUTOSAR is made up of a layered architecture of basic software modules comprising communication modules, operating system, and modules providing access to the micro controller's integrated peripheral devices (e.g., A/D converters, digital I/O).

In this paper we will reason that strictly adhering to the layered architecture proposed by AUTOSAR is inefficient as far a resource usage is concerned (especially memory consumption) due to limited or no

scalability to the needs of the application software. In order to overcome this drawback we will propose possible refactorings and optimizations of the AUTOSAR stack of basic software modules. Hereby we will focus on a well-defined part of the AUTOSAR software stack namely the software modules related to the FlexRay communication system [4, 12]. Note that although this paper focuses on the FlexRay specific communication modules, the concepts presented here are applicable to CAN and LIN as well.

Finally we will show by means of a sample application that the refactored version of these AUTOSAR software modules provides better scalability and reduces the overall resource consumption of the software stack for a specific application.

The paper is structured as follows: Section 2 illustrates the AUTOSAR hard- and software architecture. Section 3 describes the optimization concepts that will be applied to the AUTOSAR FlexRay communication modules. In Section 4 the refactored optimized AUTOSAR FlexRay communication modules are deployed in a sample application. The resulting memory footprint and the execution time of the refactored stack is analyzed and compared to an ordinary AUTOSAR stack applied to the very same application. Section 5 gives a short summary of the results and concludes the paper.

# 2 System Architecture

## 2.1 Hardware Architecture

The hardware architecture of automotive systems can be viewed at different levels of abstraction. On the highest level of abstraction, the *system level*, an automotive system consists of a number of networks interconnected via gateways. In general these networks correspond to the different functional domains that can be found in today's cars (i.e., chassis domain, power train domain, body domain).

The networks themselves comprise a number of electronic control units (ECUs) which are interconnected via a communication media. The physical topology used for the interconnection is basically arbitrary; however bus, star, and ring topologies are the most common topologies in today's cars. – This *network level* represents the medium level of abstraction.

On the lowest level of abstraction, the *ECU level*, the major parts of an ECU are of interest. An ECU comprises one or more micro controller units (MCUs) as well as one or more communication controllers (CCs). In most cases, exactly one MCU and one CC are used to build up an ECU. In order to be able to control physical processes in the car (e.g., control the injection pump of an engine) the ECU's MCU is connected to actuators via the MCU analogue or digital output ports. To provide means to obtain environmental information, sensors are connected to the MCUs analogue or digital input ports. We call this interface the ECU's environmental interface. The CC(s) facilitate(s) the physical connectivity of the ECU to the respective network(s). We call this interface of an ECU the ECU's network interface.

## 2.2 Software Architecture

The AUTOSAR software architecture makes a rather strict distinction between application software and basic or system software. While the *basic (or system) software* provides functionality like communication protocol stacks for automotive communication protocols (e.g., FlexRay [4, 12]), an operating system, and diagnostic modules, the *application software* comprises all application specific software items (i.e., control loops, interaction with sensor and actuators etc.). This way, the basic or system software provides the fundament, the application software is built upon.

The so-called *Runtime Environment (RTE)* provides the interface between application software components and the basic software modules as well as the infrastructure services that enables communication to occur between application software components.

### 2.2.1 Application Software Architecture

Application software in AUTOSAR consists of application software components, which are ECU and location independent and sensor-actuator components that are dependent on ECU hardware and therefore location dependent. Whereas instances of application software components can easily be deployed to and

relocated among different ECUs, instances of sensor-actuator components must be deployed to a specific ECU for performance/efficiency reasons.

Application software components as well as sensor-actuator components are interconnected via so-called connectors. These connectors represent the exchange of signals or the remote method invocations among the connected components.

### 2.2.2 System Software Architecture

In addition to the application software components, AUTOSAR also defines a layered architecture of system software modules [1], which provide the basic platform for the execution of the application software components. Figure 1 gives a coarse grained overview of the major categories of system software modules.

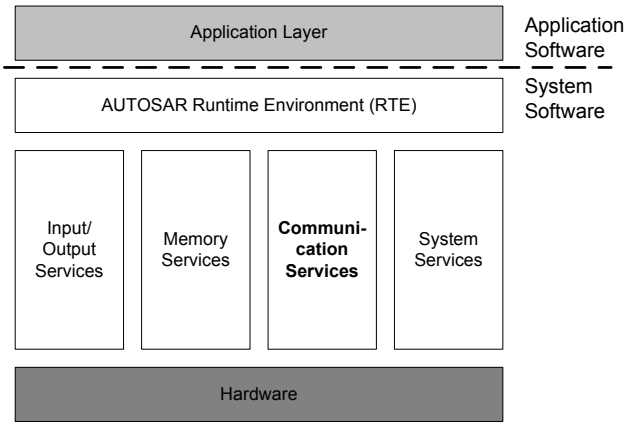



Figure 1: AUTOSAR – System Software Stack Overview

The *Input/Output Services* are software modules that provide standardized access to sensors, actuators and ECU on-board peripherals (e.g., D/A or A/D converters etc.). The *Memory Services* comprise software modules that facilitate the standardized access to internal and external non-volatile memory for means of persistent storage. The *Communication Services* category, which is of primary interest for the remainder of this paper, contains software modules that provide standardized access to vehicle networks (i.e., the Local Interconnect Network (LIN) [2], the Controller Area Network (CAN) [8,9], and FlexRay). Last but not least, the *System Services* encompass all software modules that provide standardized (e.g., operating system, timer support, error loggers) and ECU specific (ECU state management, watchdog management) system services and library functions.

The structure of the Communication Services for the FlexRay communication system are depicted in Figure 2.

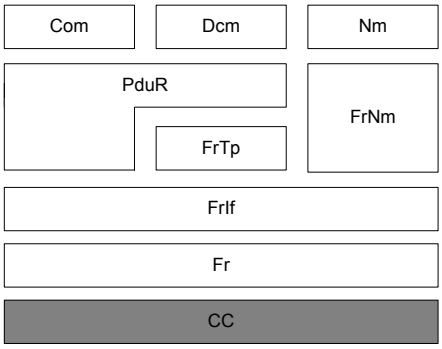



Figure 2: AUTOSAR – FlexRay Communication Services

The *COM* module provides signal-based communication to the higher layers (RTE). The signal-based communication service of COM can be used for intra-ECU communication as well as for inter-ECU communication. In the former case, COM mainly uses shared memory for this intra-ECU communication, whereas

for the latter case at the sender side COM packs multiple signals into a PDU and forwards this PDU to the PDU router in order to issue the PDU's transmission via the respective interface. – On the receiver side, COM obtains a PDU from the PDU router, extracts the signals contained in the PDU and forwards the extracted signals to the higher software layers.

The *Diagnostic Communication Manager* module (Dcm) provides services which allow a tester device to control diagnostic functions in an ECU via the communication network (i.e., CAN, LIN, FlexRay). Hereby the Dcm supports the Keyword Protocol 2000 (KWP2000) [7] standardized in ISO/DIS 14230-3 and the Unified Diagnostic Services (UDS) protocol [11] standardized in ISO/DIS 14229-1.

Network management modules provide means for the coordinated transition of the ECUs in a network into and out of a low-power (or even power down) sleep mode. AUTOSAR NM is hereby divided into two modules: a communication protocol independent module named *Generic NM* (Nm) and a communication protocol dependent module named *FlexRay NM* (FrNm).

The *PDU Router* module (PduR) provides two major services. On the one hand it dispatches PDUs received via the underlying interfaces (e.g., FlexRay Interface) to the different higher layers (COM, Diagnostic Communication Manager). On the other hand the PDU router performs gateway functionalities between different communication networks by forwarding PDUs from one interface to another of either the same (e.g., FlexRay to FlexRay) or of different type (e.g., CAN to FlexRay).

The *FlexRay Transport Protocol* module (FrTp) is used to perform segmentation and reassembly of large protocol data units (PDUs) – also termed "messages" – transmitted and received by the Diagnostic Communication Manager. This protocol is rather similar or even compatible (in certain configuration settings) to the ISO TP for CAN [10] specified in ISO/DIS 15765-2.2.

Using the frame-based services provided by the FlexRay Driver (see below) the *FlexRay Interface* module (FrIf) facilitates the sending and the reception of PDUs. Hereby multiple PDUs can be packed into a single frame at the sending ECU and have to be extracted again at the receiving ECU. The point in time when this packing and extracting of PDUs takes place is governed by the temporal scheduling of so-called communication jobs of the FlexRay Interface. The instant when the frames containing the packed PDUs are handed over to the FlexRay Driver for transmission or retrieved from the FlexRay Driver upon reception is triggered by communication jobs of the FlexRay Interface as well. The schedule of these communication jobs is aligned with the communication schedule on FlexRay. Hereby each communication job can consist of one or more communication operations, each of these communication operations handling exactly one communication frame (including the PDUs contained in this frame). The FlexRay Interface is designed to be able to deal with multiple different FlexRay Drivers for different types of FlexRay CCs (e.g., freescale MFR4300 or FlexRay CCs based on the BOSCH E-Ray core).

The *FlexRay Driver* module (Fr) provides the basis for the FlexRay Interface module, by facilitating the transmission and the reception of frames via the respective CC. Hereby the FlexRay Driver is designed to handle multiple FlexRay CCs of the same type. Thus if an ECU contains FlexRay CCs of two different types, two different FlexRay Driver modules are required.

# 3 Optimization Concepts

In this section optimization concepts for the AUTOSAR layered architecture of system software modules are proposed. These optimization concepts proposed can be divided into four different classes, namely *tailoring to the respective ECU hardware*, *combination of adjacent modules*, *tailoring to the respective application software*, and *use of special hardware support*.

## 3.1 Tailoring to the ECU Hardware

When looking at the AUTOSAR FlexRay stack for example, the corresponding AUTOSAR basic software modules are designed to handle multiple different FlexRay CCs using different FlexRay Drivers in a single ECU. This flexibility, however, is only required if a given ECU really contains of multiple FlexRay CCs (of possibly different types).

The common case, however, is an ECU which contains exactly a single FlexRay CC and thus only requires a single FlexRay Driver module.

The goal when providing ways to tailor AUTOSAR basic software modules to a specific ECU hardware is to make the complex rare case (e.g., multiple FlexRay CCs of different types) possible and to make the simple common case fast in terms of execution time and small in terms of memory footprint. Such a tailoring can be made by providing appropriate pre-compile-time configuration switches in a module's implementation.

### 3.1.1 Single Controller/Single Driver Optimizations for the FlexRay Driver/Interface Module

When looking at the API of a typical AUTOSAR FlexRay Interface function (Figure 3 shows the declaration of the API function `FrIf_GetGlobalTime()` as an example) we see that such a functions obtains an index for the desired FlexRay CC[1] as a parameter (`FrIf_CtrlIdx`).

```
extern FUNC(Std_ReturnType,FRIF_CODE) FrIf_GetGlobalTime(
    VAR(uint8,AUTOMATIC) FrIf_CtrlIdx,
    P2VAR(uint8,AUTOMATIC,FRIF_APPL_DATA) FrIf_CyclePtr,
    P2VAR(uint16,AUTOMATIC,FRIF_APPL_DATA) FrIf_MacroTickPtr
    );
```

Figure 3: AUTOSAR FlexRay Interface Function – Without Optimization

Based on this parameter and its own configuration the FlexRay Interface has to perform the following steps:

1. Decide which FlexRay Driver module is responsible for this FlexRay CC by using the controller index in a table lookup to obtain a reference to the correct FlexRay Driver.
2. Translate the controller index of the FlexRay Interface into a controller index suitable for the respective FlexRay driver[2].
3. Call the API function of the respective driver with the translated controller index and the additional parameters (i.e., the parameters `FrIf_CyclePtr` and `FrIf_MacroTickPtr` in case of the function depicted in Figure 3).

In case the ECU contains only a single type of FlexRay CC, Step 1 can be omitted and the API function of the one and only FlexRay Driver used in this ECU can be called directly. Furthermore the translation of the controller index (Step 2) does not have to take place, since in that case FlexRay Driver and FlexRay Interface use the same controller indices.

In case the ECU contains only a single CC, the `FrIf_CtrlIdx` is actually obsolete and can thus be eliminated from the function's API. This elimination reduces the function call overhead since the parameter does not need to be handed over to the called function. In order to keep this optimization transparent to the calling function, a compatibility macro can be introduces which takes care of discarding the eliminated `FrIf_CtrlIdx` parameter prior to calling the modified function (see Figure 4).

## 3.2 Combination of Adjacent Modules

Further optimization in terms of code size and execution time reduction can be achieved, if adjacent modules are combined into a single module in the optimization process. Since this combination leads to a considerable increase in the internal complexity of the resulting module, this step however shall be well considered.

[1] Note that all FlexRay CCs of an ECU (no matter whether they have the same type or different types) are assigned a unique dense index starting with zero in the configuration data structures of the FlexRay interface.

[2] Similar to the FlexRay Interface's configuration, in the configuration of the FlexRay Driver each FlexRay CC of an ECU that is serviced by the respective FlexRay Driver is assigned a unique dense index starting with zero.

```
#if (FRIF_SINGLE_CONTROLLER_OPTIMIZATION == STD_ON)

extern FUNC(Std_ReturnType,FRIF_CODE) FrIf_GetGlobalTime_SC(
    P2VAR(uint8,AUTOMATIC,FRIF_APPL_DATA) FrIf_CyclePtr,
    P2VAR(uint16,AUTOMATIC,FRIF_APPL_DATA) FrIf_MacroTickPtr
    );

#define FrIf_GetGlobalTime(ctrl, cyc, mt) \
    FrIf_GetGlobalTime_SC(cyc, mt)

#else /* (FRIF_SINGLE_CONTROLLER_OPTIMIZATION == STD_ON) */

extern FUNC(Std_ReturnType,FRIF_CODE) FrIf_GetGlobalTime_SC(
    VAR(uint8,AUTOMATIC) FrIf_CtrlIdx,
    P2VAR(uint8,AUTOMATIC,FRIF_APPL_DATA) FrIf_CyclePtr,
    P2VAR(uint16,AUTOMATIC,FRIF_APPL_DATA) FrIf_MacroTickPtr
    );

#endif /* (FRIF_SINGLE_CONTROLLER_OPTIMIZATION == STD_ON) */
```

Figure 4: AUTOSAR FlexRay Interface Function – With Optimization

When again looking at the driver and the interface modules however, this step makes perfectly sense for the single driver/single controller scenario. – In that case the driver API function to call is known exactly upon compile time of the interface module. Eliminating the module barriers between driver and interface module creates new possibilities for optimizations: Placing corresponding functions of the driver and the interface module into the very same translation unit gives the compiler the possibility to eliminate the function call overhead for the interface module's call to the respective driver function completely by inlining the respective driver function. – This is especially beneficial, for interface functions, which simply convert their given parameters and call the respective driver functions (e.g., `FrIf_GetGlobalTime()`).

### 3.3 Tailoring to the Application

In order to make it possible to tailor a given AUTOSAR basic software module to a specific application, the module itself is vertically divided into module slices. We hereby assume that each module comprises a number of rather independent functional units. The process of slicing must be governed by the following premises:

- Related functional units shall be located in the same module slice.
- The number of interactions between the functional units of different module slices must be minimized.
- The functional units within a single module slice shall exhibit a large amount of inter functional unit interaction.

This way, we arrive at a defined set of module slices, each slice consisting of one or more functional units, where the module slices exhibit a low number of inter functional unit interaction but a high number of intra functional unit interactions.

The key idea behind this slicing is that each of the resulting module slice will then be put into a dedicated translation unit (i.e., a separate source file) which is compiled separately into a respective object file. All the object files of a given AUTOSAR module are then placed in a library for this AUTOSAR module. This way, when linking the application against the libraries of the different AUTOSAR modules, the linker can select only those object files of the library (and thus only those slices of a given AUTOSAR module) that are needed by the respective application, omitting those slices that are not required.

When applying the slicing step to the AUTOSAR FlexRay Driver, we identified the following module slices: A *base slice* containing all the functional units that provide some kind of basic functionality to

the module (e.g., module initialization etc.). – This slice is always required when using any of the other slices of the module. All functional units dealing with FlexRay's global time (e.g., alarm timer services based on this global time etc.) have been allocated into a *time services slice*. Functional units dealing with FlexRay's wakeup service have been grouped into a *wakeup slice*, all functional units related to the handling of FlexRay's media test symbols have been combined in a *media test symbol slice*, and functional units for querying the operational status of the FlexRay CC (e.g., checking whether the CC is synchronous with the other CCs in the FlexRay network) have been combined in a *status slice*. The functional units dealing with the transmission of PDUs (in the FlexRay Interface) or frames (in the FlexRay Driver) have been combined in a *transmission slice* whereas the functional units dealing with reception have been allocated in a *reception slice*.

### 3.4 Use of Special Hardware Support

Further optimization mostly in terms of execution time can be achieved when special hardware support is used for the implementation of the specific modules.

Most of the MCUs used in todays automotive applications are equipped with special hardware like DMA controllers or co-processors (e.g., freescale's XGATE co-processor for the S12X family) which can perform data transfers between memory mapped I/O devices and the MCU's internal memory independently of the main CPU. Making use of this special hardware in the implementation of the AUTOSAR basic software modules reduces the processing load of the main CPU, thus saving more CPU bandwidth for the application.

Especially the lower AUTOSAR communication service modules (i.e, driver, interface and PDU router modules) are well suited for making use of this kind of special hardware. When looking at the driver modules for example the data transfer between the CC's hardware buffers and the MCU's main memory can easily be accomplished by a DMA controller.

When looking at a situation where driver and interface module are combined as described in Section 3.2, the benefit of the use of special hardware can be increased even more. In this scenario, the packing and unpacking of PDUs into and out of frames can be accomplished by a sufficiently smart DMA controller as well. The main CPUs job in that case is reduced to programming the DMA controller to perform the desired transfers (including the packing and unpacking much like the *scatter-gather I/O* in UNIX systems) and initiating the transfers at the correct point in time (according to the interface module's configuration).

### 3.5 Compliance to AUTOSAR

Note that all of these optimization approach are fully compliant with AUTOSAR, since AUTOSAR defines different *implementation conformance classes (ICCs)*, namely *ICC3* treating each single AUTOSAR module as a target for conformance tests, *ICC2* treating a cluster of modules as a target for conformance tests, and *ICC1* treating all modules underneath the RTE as a whole as target for conformance tests [1].

In conformance classes ICC2 and ICC1 AUTOSAR only requires that the the interfaces of a module cluster with other AUTOSAR modules (which are not a member of this cluster) have to adhere to the AUTOSAR specification. Those interfaces which are only module cluster internal (intra module cluster interfaces) do *not* have to adhere to the AUTOSAR specification.

Thus the optimizations presented in the previous sections result in a AUTOSAR FlexRay communication stack that is fully AUTOSAR compliant at the ICC2 and ICC1 level.

## 4 Evaluation

In order to assess the benefits of the proposed optimizations, we investigated the memory consumption and the execution time of the combined FlexRay Interface and FlexRay Driver with single controller and single driver optimization enabled in comparison to the respective ordinary FlexRay Interface and the ordinary FlexRay Driver.

As far as the memory consumption is concerned, both version (original and optimized) have been compiled with the same compiler using the same compiler settings. The memory (RAM and ROM) used has been derived from the compilation output (i.e., from the map file produced in the compilation process).

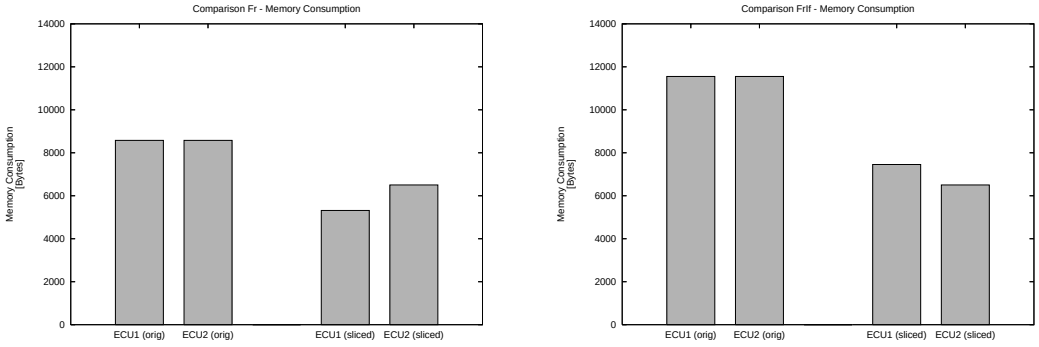
For the runtime comparison the FlexRay CC's timer (which ticks with a granularity of 1 $\mu s$) has been used to time stamp the invocation and the termination of the respective API functions of the transmission and the reception slices.

## 4.1 Tailoring to the Application

The first evaluation has been targeted at showing the benefit of slicing the ordinary AUTOSAR modules into distinct module slices and placing these slices in separate translation units in order to allow for tailoring of the modules to the specific application. – For this purpose the slicing of the FlexRay Driver and the FlexRay Interface modules has been performed as described in Section 3.3.

This first evaluation has been conducted on an ARM922T CPU running at 166 MHz and providing 16 bit access to an external ERAY 1.0 FlexRay communication controller. For the evaluation a rather beneficial sample application consisting of two ECUs has been used, where ECU 1 only transmits PDUs/frames whereas ECU 2 only receives frames/PDUs.

Figure 5(a) shows the memory consumption (sum of RAM and ROM consumption) of the used module slices of the FlexRay Driver for ECU 1 and ECU 2 of the sample application compared to the memory consumption of the ordinary FlexRay Driver module for both ECUs. Figure 5(b) shows the same data for the FlexRay Interface. Hereby the following module slices of the FlexRay Driver and the FlexRay Interface have been deployed to the respective ECUs: The base slice and the transmission slice have been deployed to ECU 1, whereas the base slice and the reception slice have been deployed to ECU 2. The other slices of the FlexRay Interface and the FlexRay Driver are not required for this particular application and have thus not been deployed.




(a) FlexRay Driver (b) FlexRay Interface

Figure 5: Memory Consumption Comparison

The benefit of this slicing is obvious when looking at Figure 5: For the FlexRay Driver (Figure 5(a)) as well as for the FlexRay Interface (Figure 5(b)), the memory consumption of the sliced version is significantly lower than the memory consumption of the ordinary modules (approximately by 30%).

## 4.2 Combining Adjacent Modules

The second evaluation has been targeted at showing the benefit of combining adjacent modules. – For this purpose an integration of the FlexRay Interface with the FlexRay Driver (as described in Section 3.2) has been performed and single controller/single driver optimization (as described in Sections 3.1) has been enabled.

For this evaluation an NXP SJA2510 N1B MCU from NXP Semiconductors[3] with integrated FlexRay CC NXPFRDLC running at 80 MHz and providing 32 bit access to the FlexRay CC has been chosen. – Program execution took place from the MCU's internal flash.

Table 1 illustrates the key figures, namely the number of PDUs contained in the frame and the total frame length in units of bytes, and the typical use cases in FlexRay networks for the frame layout used in

[3] formerly a division of Philips

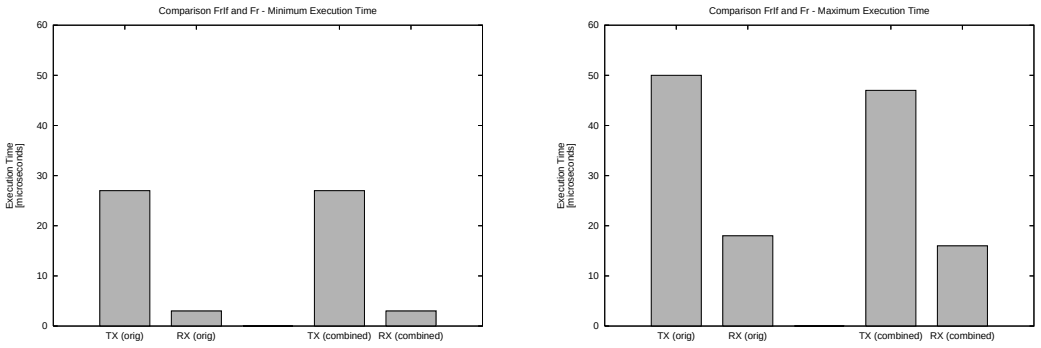
this evaluation.

| Type | # PDUs | Length | Use Case |
|---|---|---|---|
| 1 | 1 | 8 | Transmission of a CAN-like frame (consisting of 8 bytes) |
| 2 | 1 | 16 | Transmission of frames with signals for a distributed control loop |
| 3 | 1 | 32 | Transmission of frames with signals for a distributed control loop |
| 4 | 3 | 8 | Transmission of multiple CAN-like frames packed into a single FlexRay frame[a] |
| 5 | 1 | 254 | Transmission of diagnostic frames for in-system flash programming of ECUs |

[a] This is a use case for a FlexRay backbone tunneling several CAN frames.

Table 1: Frame Layout Overview

Figure 6 illustrates a comparison of the sum of the execution times for the plain FlexRay Driver (Fr) and the plain FlexRay Interface (FrIf) to the execution times for the combined FlexRay Driver and FlexRay Interface.




(a) Minimum (b) Maximum

Figure 6: Comparison Execution Time Fr & FrIf

Hereby Figure 6(a) depicts the minimum executions times whereas Figure 6(b) shows the maximum executions times of the transmission/reception paths of the ordinary FlexRay Interface and the ordinary FlexRay Driver (TX/RX (orig)) as well as of the combined transmission/reception paths (RX/RX (combined)). – It can be seen that as far as the minimum execution time is concerned, the combination has no effect (neither positive nor negative) on the execution time. In the charts depicting the maximum execution times however (Figure 6(b)) the benefit of combining the respective slices becomes obvious (approximately 10%).

The effects of this combination with respect to memory consumption can be seen in Figure 7. Looking at the third and fourth bar we see that the memory consumption of the combined FlexRay Interface and FlexRay Driver lies slightly below the sum of the stand-alone ordinary modules.

Note that this decrease in memory consumption gained by combining both modules is independent from the decrease gained by the dividing of the two modules into module slices. – Thus combining both approaches results in increased benefit.

# 5 Conclusion

In this paper we presented possible optimization strategies for the AUTOSAR system software stack. We demonstrated the feasibility of these optimizations by implementing the optimizations for two sample AUTOSAR modules, namely FlexRay Driver and FlexRay Interface, and by deploying these optimized versions in a simple application. We pointed out the benefits of these optimized versions by comparing the memory

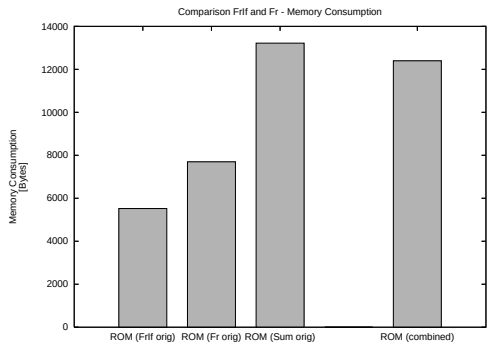



Figure 7: Comparison Memory Consumption Fr & FrIf

consumption and the required execution time of the optimized versions to the ordinary AUTOSAR versions. This comparison yielded an approximately 10% reduction in execution time as well as an average 30% reduction as far as the memory footprint is concerned.

The reduction in execution time is on the one hand due to the combination of the FlexRay Driver and the FlexRay Interface module, resulting in a much tighter integration and thus in an elimination of the function call overhead at the interface between these originally separated modules. On the other hand the reduction in execution time is caused by the tailoring of the modules to the ECU hardware, i.e., by enabling optimizations for a single FlexRay controller and a single FlexRay Driver and thus eliminating the need for passing controller indices as parameters and using lookup-tables when calling FlexRay Driver API functions in the FlexRay Interface.

The decrease in the memory consumption is caused by vertical sub-structuring of the original modules into finer-grained modules slices, by grouping these slices into separate translations units and by selecting only those translation units required for a particular application during the link process.

## References

[1] AUTOSAR Consortium. AUTOSAR – Layered Software Architecture. Technical Report Version 2.2.1, Release 3.0, Rev 0001, AUTOSAR Consortium, February 2008.

[2] LIN Consortium. LIN Specification Package. Technical Report Version 2.1, LIN Consortium, November 2006.

[3] H. Fennel, S. Bunzel, H. Heinecke, J. Bielefeld, S. Fürst, K.-P. Schnelle, W. Grote, N. Maldener, T. Weber, F. Wohlgemuth, J. Ruh, L. Lundh, T. Sandén, P. Heitkämper, R. Rimkus, J. Leflour, A. Gilberg, U. Virnich, S. Voget, K. Nishikawa, K. Kajio, K. Lange, T. Scharnhorst, and B. Kunkel. Achievements and Exploitation of the AUTOSAR Development Partnership. In *Proceedings of the Convergence 2006*, Detroit, MI, USA, October 2006.

[4] T. Führer, F. Hartwich, R. Hugel, and H. Weiler. FlexRay – The Communication System for Future Control Systems in Vehicles. In *Proceedings of the SAE 2003 World Congress & Exhibition*, Detroit, MI, USA, March 2003. Society of Automotive Engineers.

[5] P. Hansen. AUTOSAR Standard Software Architecture Partnership Takes Shape. *The Hansen Report on Automotive Electronics*, 17(8):1–3, October 2004.

[6] P. Hansen. New S-Class Mercedes: Pioneering Electronics. *The Hansen Report on Automotive Electronics*, 18(8):1–2, October 2005.

[7] ISO. Road Vehicles – Diagnostic Systems – Keyword Protocol 2000 – Part 3: Application Layer. Technical Report ISO/DIS 14230-3, ISO (International Organization for Standardization), 1, rue de Varembe, Case postale 56, CH-1211 Geneva 20, Switzerland, 1999.

[8] ISO. Road Vehicles – Controller Area Network (CAN) – Part 1: Data Link Layer and Physical Signalling. Technical Report ISO/DIS 11898-1, ISO (International Organization for Standardization), 1, rue de Varembe, Case postale 56, CH-1211 Geneva 20, Switzerland, 2003.

[9] ISO. Road Vehicles – Controller Area Network (CAN) – Part 2: High-Speed Medium Access Unit. Technical Report ISO/DIS 11898-2, ISO (International Organization for Standardization), 1, rue de Varembe, Case postale 56, CH-1211 Geneva 20, Switzerland, 2003.

[10] ISO. Road Vehicles – Diagnostics on Controller Area Networks (CAN) – Part 2: Network Layer Services. Technical Report ISO/DIS 15765-2.2, ISO (International Organization for Standardization), 1, rue de Varembe, Case postale 56, CH-1211 Geneva 20, Switzerland, April 2003.

[11] ISO. Road Vehicles – Unified Diagnostic Services (UDS) – Part 1: Specification and Requirements. Technical Report ISO/DIS 14229-1, ISO (International Organization for Standardization), 1, rue de Varembe, Case postale 56, CH-1211 Geneva 20, Switzerland, 2004.

[12] R. Mores, G. Hay, R. Belschner, J. Berwanger, C. Ebner, S. Fluhrer, E. Fuchs, B. Hedenetz, W. Kuffner, A. Krüger, P. Lohrmann, D. Millinger, M. Peller, J. Ruh, A. Schedl, and M. Sprachmann. FlexRay – The Communication System for Advanced Automotive Control Systems. In *Proceedings of the SAE 2001 World Congress*, Detroit, MI, USA, March 2001. Society of Automotive Engineers.