05AE-340

Standard Software Modules - One Key for Future Distributed Systems

Roman Nossal, Thomas M. Galla
DECOMSYS

Jochen Olig
3SOFT

ABSTRACT

The key to the required level of quality and safety in future distributed systems is a comprehensive standardized ECU (Electronic Control Unit) software architecture and infrastructure.

INTRODUCTION

The number of electronic systems in cars is continuously growing. Electronic systems 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. The functional software heavily depends on the actual system and is a discriminating factor of competitive importance. This, however, does not apply to the software infrastructure. The software infrastructure, which is comparable to the BIOS or the operating system of a PC, does not influence competition – it is merely the foundation on which functions that are relevant for competition are built.

Keeping this in mind, the obvious approach is to standardize this software infrastructure for the sake of cost-effectiveness, reliability, and interoperability. This paper will provide a brief overview of the current status with respect to software infrastructure. This will be followed by a definition of goals to be achieved, as well as some insights into the AUTOSAR initiative [1], whose purpose is to define and standardize a comprehensive software infrastructure. The paper concludes with the introduction of TimeCore, a set of standard software modules for use in series applications.

STATE OF THE ART

Historically, if an ECU had some software infrastructure at all, it used to be some proprietary implementation. This was different for every ECU – not even parts, which were independent from the actual hardware, would be reused in another ECU.

Of course ECU suppliers – especially those with higher volumes and many different types of ECUs – discovered the inefficiency of this approach. As a result, they started to implement some platform software that was used in an entire family of ECUs. Hardware drivers and similar modules became company standards by this approach. This solution eased the implementation of new ECUs for the supplier considerably, yet it did not provide any gains for the system integrator, be it the OEM or a system supplier.

Consequently, OEMs and system suppliers took the next step. They no longer wanted to accept a company-specific platform, but to have one common software platform for all ECUs of a system. Thus the OEMs started to implement their own software infrastructure and to demand that their suppliers use this basic software in any ECU supplied to them. While this improved the situation for the OEMs, it diminished the advantages of platform software for the suppliers. They could no longer reuse the software for different OEMs.

Both developments, the platform software and the OEM-specific infrastructure, show that a standardization driven by either of the two main groups in the automotive industry will not be able to achieve the goal and provide benefits for all participants. Any standardization that does not involve the entire industry will have some obvious drawbacks:

- The software failure rate is high, because software has to be re-implemented more often than necessary. A supplier cannot reuse his implementation for another OEM, but instead has to implement it again to match this OEM's software infrastructure. Similarly, OEMs or their software infrastructure providers have to develop a specific implementation for each OEM, thus diminishing the effects of reuse.

- For the same reason, it is more or less impossible to argue that the implementation is field-proven, an important fact in safety-relevant systems.
- Interoperability is threatened. Integrating ECUs from different suppliers with different software platforms always carries the risk of incompatibilities at the interfaces.
- Most important of all, any reimplementation of the same functionality is inefficient. All the more so if it is not even relevant to competition.

To overcome these problems the automotive industry has one answer – industry-wide standardization.

OBJECTIVES

Standardization ensures a known, fixed functionality together with a stable API. The advantages are obvious:

- Standardization allows the re-use of functions and infrastructure.
- Standardization facilitates exchangeability on the OEM as well as on the supplier side.
- Standardization ensures interoperability. A service that has been standardized has a well-defined interface, which will be accessible by any other standard-conforming implementation.

RE-USE

Standardized software modules provide for better re-use than their proprietary counterparts. OEMs can easily re-use the functional software in different product lines if they apply the same software infrastructure in them. No porting is required as long as the functional software components can access an unaltered interface.

Suppliers, on the other hand, can re-use their software for different OEMs. Assuming that all OEMs agree on a common software architecture, the suppliers' software modules will use the same interfaces for each of the OEMs, so no porting is required.

Finally, the software and tool vendors can also re-use the software infrastructure they have implemented and sell it to several OEMs and suppliers.

Apart from the increased efficiency of re-using existing software, there is also a consideration of quality. Software that is used in several applications can claim to be field-proven. All applications that use the same piece of software can be considered field tests for this software. Thus, the more widely the same software is used, the better is its test coverage. The probability that there are dormant errors in the software decreases with each application. Likewise, the confidence in the quality of a specific software module increases with each application, whether it is part of the software infrastructure or the functional software.

EXCHANGEABILITY

With the possibility to re-use code there is a certain degree of exchangeability. OEMs are no longer bound to a specific software vendor. Any vendor who is able to apply the standardized software infrastructure can supply this to the OEM.

The OEM, however, also becomes exchangeable for the software vendors. As mentioned above, the vendors can provide the same implementation to several different OEMs.

INTEROPERABILITY

A service that has been standardized offers a well-defined interface. Hence, any other standard-conforming implementation can use this service without any alterations to the implementation.

For this reason, in order to be fully effective, the standardization needs to cover all aspects of an infrastructure: the operating system, hardware drivers, interface layers, communication drivers, and service layers like Network Management, a Transport Layer etc.

INITIATIVES

HISTORICAL OVERVIEW

In the recent years there have been several consortia, initiatives, and funded projects with a common goal – to standardize software infrastructure or at least parts of it. There are OSEK/VDX, HIS, and the EASIS project, to name but a few. The OSEK/VDX consortium [2] took action to standardize an operating system (OSEK OS) for embedded real-time applications. A communication access layer (OSEK COM) and a Network Management service (OSEK NM) complement the operating system. HIS [3] went a step beyond this, took the results of the OSEK/VDX consortium and added important software modules such as hardware drivers for input and output.

A somewhat different approach is taken in the EASIS project [4], which is funded by the European Commission. In this project a group of OEMs, suppliers, and tool vendors have gathered with the objective to define a common software architecture that is suitable for use in safety applications in vehicles.

The most recent development in this respect is the AUTOSAR initiative, which will be dealt with in the next section.

THE AUTOSAR INITIATIVE

The AUTOSAR initiative pursues the goal of establishing an open standard for automotive E/E architecture that will serve as a basic infrastructure for managing functions. This comprises the standardization of basic system functions and functional interfaces, the ability to

integrate and transfer functions and to substantially improve software updates and upgrades over the lifetime of the vehicle.

The objectives of AUTOSAR are [1]

- Management of E/E complexity associated with growth in functional scope
- Flexibility for product modification, upgrade and update
- Scalability of solutions within and across product lines
- Improved quality and reliability of E/E systems.

AUTOSAR will provide a common software infrastructure for automotive systems in all vehicle domains based on standardized interfaces for the different layers shown in Figure 1.

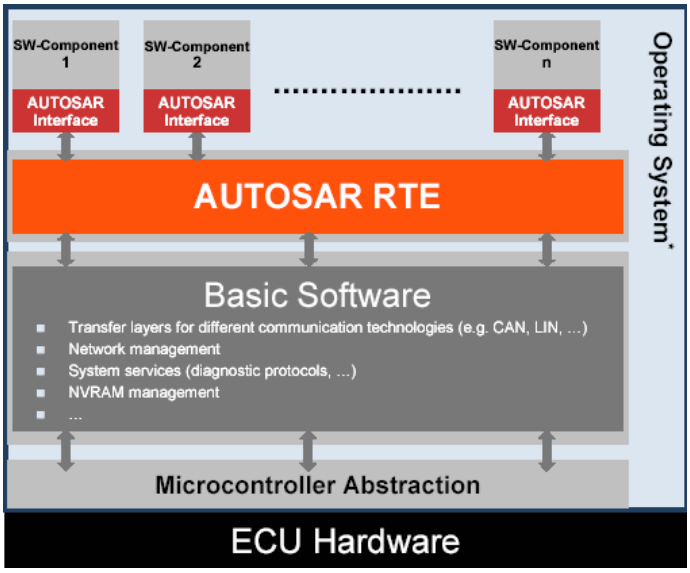

Figure 1: AUTOSAR Software Architecture (Source: www.autosar.de).

The scope of AUTOSAR is the entire software infrastructure of an ECU. It covers all layers, from the abstraction of the actual hardware, over the so-called “basic software”, the AUTOSAR Run-Time Environment (RTE), to the interfaces for the functional software components.

TIMECORE

While an AUTOSAR-compliant software infrastructure is still a product of the future, there is a solution available. With their joint product called TimeCore, 3SOFT and DECOMSYS present a software kit that contains a wide variety of standardized software modules. TimeCore offers a software infrastructure similar to the intended coverage of AUTOSAR.

The following paragraphs provide an overview of the main constituents of the TimeCore package. While the description naturally highlights certain features contained in the TimeCore product, the modules presented and their functionality are valid for any software infrastructure in an automotive ECU.

OSEKTIME OS

OSEKtime OS is designed as a time-driven single chip operating system for distributed applications like X-by-Wire systems, which need to have synchronized access to various ECUs.

OSEKtime executes functions either as interrupt service routines or as tasks. Interrupt service routines (ISRs) are triggered by interrupts and execute interrupt-related services. Tasks on the other hand are started at defined points in time. They can have one of three states. In the *running state*, the CPU is assigned to the task and its commands are executed. The *preempted state* is reached by a task that has been in the running state and was preempted by another task that is to be activated. A task can leave the preempted state only if the preempting task changes into the suspended state. An inactive task that can be activated is in the *suspended state*. These state changes are shown in Figure 2.

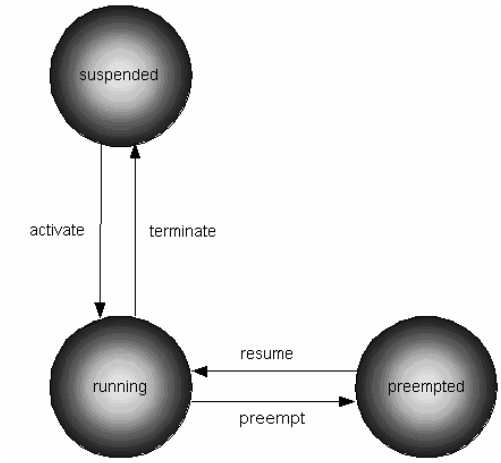

Figure 2: OSEKtime Task States

The activation times of the tasks are stored in a so-called dispatcher table during system configuration. A dispatcher uses the information stored in this table to activate the tasks according to their activation times. The dispatcher itself is usually triggered from a non-maskable interrupt.

The dispatcher table is processed cyclically. If a time-driven task is still in the *running state* when the activation time of another time-driven task is reached, the first task passes over to the *preempted state* and stays there until the interrupting task is finished. This kind of scheduling is called stack-based scheduling; no priorities are assigned to the tasks - only the activation times define the precedence between different tasks. The later a task is started the less likely it is that another task can interrupt it while it is being executed.

Depending on the design of the application, a dispatcher round can include times during which no tasks or interrupt service routines are active. A task named ttIdleTask is assigned to the CPU during these idle times. This task is provided by the OS, has no entry in the dispatcher table and no deadline assigned. Its sole purpose is to take up CPU time otherwise unused.

The occurrence of interrupts is also supervised and controlled by the operating system. An interrupt source is only switched on through an entry in the dispatcher table. After an interrupt is raised, its interrupt service routine will be executed. Then the operating system switches off the interrupt source and it can only be switched on again by a new entry in the dispatcher table. In this way, the whole dispatcher round can be divided into unconnected time intervals where a particular interrupt can occur at most once within one of these intervals.

The dispatcher's sole duty during run time is to cyclically execute the dispatcher table. It does not check for any task constraints etc. The configuration of the distributed system has to take care of these issues. Besides information about existing constraints between tasks, e.g., a mutual exclusion due to access to a common resource, the knowledge of the worst case execution time (WCET) [5,6,7,8] of every task and interrupt service routine is essential to plan the course of a dispatcher round.

OSEK/VDX SUBSYSTEM

If an application consists of a time-critical part and a more complex, non time-critical part, then OSEKtime offers an interesting solution for it: Instead of the idle task described above, one can integrate an OSEK/VDX subsystem. This OSEK/VDX subsystem acts as the idle task for the OSEKtime OS. Whenever the OSEKtime OS is idle, i.e., no task is being executed, the OSEK/VDX subsystem is run.

OSEK/VDX is a standard for an operating system that was designed for applications in the area of car body electronics or power train. It provides the user with system services like tasks, interrupt service routines, counters, alarms, and resources. It also offers an event mechanism for the signaling between tasks and interrupt service routines.

OSEK/VDX is a configurable and static system. Since every system service that an application wants to use must be defined and configured before compile time, there is no dynamic management of system resources. This results in an operating system kernel that is tailored to the needs of the corresponding application and that can therefore be highly optimized.

OSEKTIME FTCOM

FTCom (Fault Tolerant Communication) [2] provides all associated ECUs with a global time to which they synchronize their local time base. Furthermore, it is responsible for the fault tolerant sending and receiving of messages between the distributed communication partners in a temporally deterministic way.

The fault tolerant mechanisms of FTCom consist of the replication and reduction of messages. On the sender's side, the FTCom layer replicates a single application message into several instances and transmits these via redundant communication paths (e.g. on different communication channels). On the recipient's side, FTCom reduces the several instances of the received message to a single application message that is provided to the application.

FLEXRAY

In order to achieve the above-mentioned determinism at the communication level, OSEKtime FTCom in TimeCore uses the FlexRay time-driven communication protocol [9,10].

The FlexRay communication medium is accessed synchronously in a slot-based manner. Time is divided into equally sized slots that are owned exclusively by ECUs for transmitting their data. Slot assignment is done off-line during the system planning stage, thus eliminating any on-line competition for the communication medium. FlexRay integrates two different communication paradigms, time-driven and event-driven media access. In the so-called static segment, communication access follows a TDMA scheme with equally sized slots. In the dynamic segment an event-driven minislotting scheme is applied. The dynamic segment includes a prioritization scheme, thus enabling variable bandwidth distribution during run time.

Both segments strictly adhere to the above-mentioned exclusive ownership of slots providing a key foundation for an important FlexRay feature, composability. As each ECU owns slots exclusively, it is possible to develop ECUs autonomously and to integrate them later without side effects.

FlexRay also supports fault-tolerant clock synchronization and provides a global time base to OSEKtime FTCom.

FLEXRAY NETWORK MANAGEMENT

Network management (NM) in the automotive area handles the controlled startup and shutdown of the communication across a shared communication infrastructure. In particular, Network Management coordinates the common view of the network where each node can initiate a shutdown or a wakeup of the communication system during normal system operation. In TimeCore, NM is based on the FlexRay communication system.

FLEXRAY TRANSPORT LAYER

In general, a transport layer supports the transmission of messages whose length is greater than the maximum payload length of the underlying communication system. On the sender's side, the Transport Layer will split such long messages into packets that the underlying communication system can handle. The transport layer reassembles these packets on the receiver's side.

The support of direct addressing (sending a message to a specific node), broadcasts (sending a message to all nodes in the network), and multicasts (sending a message to a specific subset of nodes) is also implemented in the Transport Layer. Consequently the Transport Layer provides a unique address to every node addressable via the Transport Layer, addresses for groups of nodes, and a broadcast address.

CAN DRIVER AND CAN TRANSPORT LAYER

A CAN driver is also included in TimeCore. The driver can be operated in interrupt and polled mode. The number of message objects is user-definable, and a priority based message queue is also supported.

The working principle of the included CAN transport layer is similar to the transport layer for FlexRay.

KWP2000

KWP2000 is a standard for a diagnostic layer. It describes a set of functions with which an ECU can be diagnosed from a tester tool. The diagnostic layer is based on the Transport Layer. Within TimeCore, the KWP2000 layer can be accessed via CAN and FlexRay.

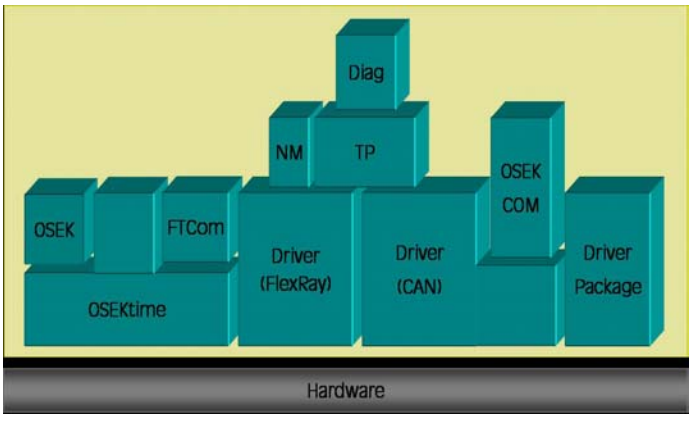

Figure 3: TimeCore Architectural Overview

TOOLING

All components of TimeCore require configuration before run time. The software modules are set up to match the requirements of the application exactly. Services that are not used by the application are not configured into the run-time software of TimeCore. This leads to highly optimized code with respect to execution time and memory consumption.

To configure TimeCore the development engineer can make use of the comprehensive toolset from DECOMSYS and 3SOFT, which is included in TimeCore. The DECOMSYS::DESIGNTOOLS tool suite with its graphical user interface allows the user to design the architectural model of his distributed system. Based on this model, the DECOMSYS::DESIGNER configures the FlexRay communication system and builds a communication schedule. Finally it generates FTCom code, which takes the particularities of each signal into account. *tresos*, a configuration tool that is used to configure OSEK/VDX, OSEKtime and all CAN components, is also part of TimeCore. During configuration, *tresos* resolves inter-component dependencies automatically. For example, an interrupt service routine used by the CAN Driver will be added automatically to the OSEK/VDX configuration. Both tools, DECOMSYS::DESIGNER and *tresos*, are well-integrated and thus support the user in configuring all modules during the development cycle.

CONCLUSION

This paper has stated the case for a standardized software infrastructure in automotive ECUs. We have argued that the objectives of re-use, exchangeability, and interoperability can only be achieved if all players in the automotive market, OEMs and suppliers alike, agree on one standard for software modules that are to be used in any ECU.

With its joint product TimeCore, DECOMSYS and 3SOFT deliver a set of software modules that meets all requirements of automotive applications. TimeCore comprises a time-driven as well as an event-driven operating system, communication layers, network management service, etc.

While TimeCore is readily available, several automotive companies have teamed up in the AUTOSAR consortium with the goal of standardizing the software infrastructure for automotive electronics. As soon as a set of AUTOSAR standards becomes available, DECOMSYS and 3SOFT will provide an updated, AUTOSAR-compliant version of TimeCore.

REFERENCES

1. AUTOSAR Website. www.autosar.org
2. OSEK/VDX Website. www.osek-vdx.org
3. HIS Website. www.automotive-his.de
4. EASIS Website. www.easis-online.org
5. P. Puschner. Timing Analysis for Real-Time Programs. PhD thesis, Technische Universität Wien, Institut für Technische Informatik, Treitlstraße 3/3/182-1, 1040 Vienna, Austria, 1993
6. Raimund Kirner. Extending Optimising Compilation to Worst-Case Execution Time Analysis. PhD thesis,

Technische Universität Wien, Institut für Technische Informatik, Treitlstraße 3/3/182-1, 1040 Vienna, Austria, May 2003
7. C. Y. Park and A. C. Shaw. Experiments with a Program Timing Tool based on a Source-Level Timing Schema. IEEE Computer, 24(5):48–57, May 1991
8. P. Puschner and A. V. Schedl. Computing Maximum Task Execution Times – A Graph-Based Approach. The Journal of Real-Time Systems, 13:67–91, 1997
9. FlexRay Consortium Website. www.flexray.com
10. A Manager's Guide to FlexRay. Automotive Design and Production, September 2002. http://www.flexray.com/publications/a_manager_guide.pdf

CONTACT

Please direct questions or comments about this article to Roman Nossal at DECOMSYS - Dependable Computer Systems, Hardware und Software Entwicklung GmbH, Stumpergasse 48/28, A-1060 Vienna, Austria; nossal@decomsys.com; www.decomsys.com

DECOMSYS and 3SOFT are development members of the FlexRay® consortium.

Roman Nossal is product manager at DECOMSYS. He has a PhD in computer science from the Vienna University of Technology and an MBA in General Management from Danube University in Krems, Austria.

Thomas M. Galla is the technical coordinator of all DECOMSYS developments. He has a PhD in computer science from the Vienna University of Technology.

Jochen Olig is senior software engineer with 3SOFT GmbH. He has studied engineering at the University of Erlangen-Nuernberg.

Jochen Olig can be contacted at

3SOFT GmbH Frauenweiherstrasse 14, D-91058 Erlangen, Germany. Jochen.Olig@3SOFT.de