Resource Management for Multicore Aware Software Architectures of In-Car Multimedia Systems

Andreas Knirsch[1,2], Joachim Wietzke[1], Ronald Moore[1], Paul S. Dowland[2]

[1] In-Car Multimedia Labs, Faculty of Computer Science,
University of Applied Sciences Darmstadt, Schöfferstr. 8b, D-64295 Darmstadt
{andreas.knirsch, joachim.wietzke, ronald.moore}@h-da.de

[2] Centre for Security, Communications and Network Research (CSCAN),
University of Plymouth, Plymouth, PL4 8AA, United Kingdom
info@cscan.org

Abstract: With increasing hardware capabilities the demands on the functionality of user centric systems continuously expand. The next generation of automotive embedded systems is going to make use of multicore hardware architectures, which strongly enhances the computational power. This means a movement from concurrent to parallel computing. Although the competition for CPU time will decrease, other resources are not available in multiple instances. This raises the need for a management unit that controls access to resources other than the CPUs. Such a resource manager is able to utilise the capabilities of multicore hardware architectures for component based software systems more predictably. This paper builds a case for a resource scheduler, identifies requirements and provides details of a prototype implementation. As an illustrative example, the domain of automotive multimedia/infotainment systems is used.

1 Introduction

Automobiles can no longer be designed without an in-car multimedia (ICM) system, providing comprehensive functionality to their passengers. They are already used to enjoying digital audio and video from their personal media library, navigating to their destinations with the support of satellite-backed guidance, controlling their mobile phones using the car's human-machine interface (HMI), accessing content and functionality of the Internet using wireless access networks, and much more. Additional functionalities enrich those embedded computer systems with every new product generation. Most major car manufacturers have announced plans to introduce digital distribution platforms (application stores) to provide the capability to add or modify certain functionality on demand, which reinforces the need for solid architectures and frameworks. Despite the increasing complexity and the heterogeneity of the services provided (in terms of entertainment, control of comfort systems, and driver information), all functions have to be integrated into a homogeneous and trustworthy system, utilising a common hardware platform.

erschienen im Tagungsband der INFORMATIK 2011
Lecture Notes in Informatics, Band P192
ISBN 978-3-88579-286-4

Economic pressure within the highly competitive automotive domain requires a focus on new product development (NPD) and time-to-market. As a result the development process is parallelised by the use of an extensive division of labour. Tier 1 original equipment manufacturers (OEM) subcontract development tasks to several specialised Tier 2 OEMs. The integration process is tampered by drastic problems caused by the use of independently developed software artefacts [SVDN07]. Further, integrators loose control over the complete system due to downloaded components made available over the wireless access networks. Therefore, as a matter of principle, future systems cannot be built by using established architectures.

Additionally, the requirements of the automotive domain have to be considered, which includes cost pressure due to high product quantities, long product life-cycles, poor maintainability, and harsh operating conditions (in addition to the already mentioned short development cycle). Operating conditions are characterised by a wide range of climate conditions in terms of temperature and humidity, low voltage situations and uneven road surfaces [BRR11]. A dependable system has to be created to be used for more than 15 years, satisfy the quality demands of the vehicle's owner and justify actual cash values of up to several thousand Euro.

1.1 Architectural approaches

The focus here is set on the high number of different software functions, which have to be joined in such a way that the resulting system appears as an integral whole to its user, the passer nains (for exam

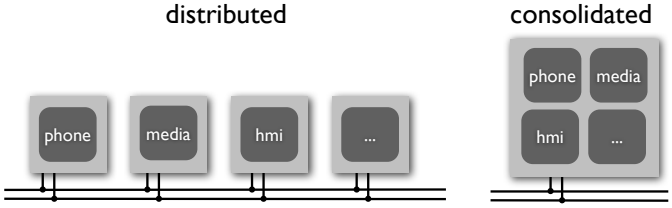

Figure 1: Architectural approaches.

In the past, having a separate physical platform for each domain that provides a defined subset of the overall functionality solved this issue, as depicted in Figure 1 ('distributed'). With this approach conflicting system constraints can be isolated to separate compartments, to respect concerns such as performance, CPU load, and time behaviour. Further, the probability of an erroneous behaviour of one single subsystem propagating to the others decreases, since shared resources are reduced to a minimum. However, there are significant disadvantages, namely: the need for multiple cases, power supplies, thermal dissipation facilities, and communication devices to connect the interdependent system

ISBN 978-3-88579-286-4

components.

With evolving hardware capabilities, especially the performance capacity of the CPUs, the software components are increasingly consolidated onto a common physical platform, as shown in Figure 1 ('consolidated'). This has positive impacts on the energy consumption and thermal dissipation and allows a more efficient communication between the components by use of inter process communication (IPC) facilities like shared memory. Fieldbus systems and network components for interconnecting the subsystems become obsolete, which positively affects the bill of materials (BOM). Hardware platform abstractions and scheduling becomes necessary to portion the system resources in respect of the component's needs. Therefore a real-time operating system (RTOS) provides the needed support.

The result of this evolution is a very highly integrated embedded software system, which consists of heterogeneous components executed in parallel and competing for the platform's resources [DNSV10]. For single core hardware architectures the competition for resources can mainly be reduced to CPU time, which is controlled through the RTOS by use of a scheduler and based on configurable priorities.

The underlying RTOS has to balance the diverse timing requirements, which is demanding due to the co-operation of time and event triggered tasks. The result is a trade-off between determinism and reactivity by use of appropriate prioritisation schemes and scheduling algorithms. Nevertheless, this dilemma is not solvable due to the concurrent utilisation of a single CPU [Kai06]. With the knowledge gathered through an inter-institutional research project it can be stated that a current ICM system makes use of more than 1,000 threads at runtime, issued from a source base with around 86,000 files (or 3.1 GByte), and produced by 235 software engineers at 13 locations. Rising complexity and the growing functionality, which must be incorporated, has led to an increasing interest in the issue of integration.

With multiple general purpose processing cores (multicore architectures) it is possible to cluster certain software components into execution domains and assign those to predefined cores. Domains with incompatible prioritisation can be grouped in such a way, that they execute in parallel (in contrast to concurrent execution) and do not interfere in terms of time constraint runtime behaviour. Such an approach and implementation for an embedded software framework is presented in [KWMD10].

1.2 Problem statement

The use of multicore platforms reduces the competition for computational resources of the CPU. But other resources are still available only once, or can only be accessed by one software component at a time. This includes memory, network interfaces, devices attached via serial line, or various other I/O devices. For single core architectures, the RTOS's scheduler grants time slices for CPU utilisation to threads, which implicitly control the access to resources by use of the configured priorities for the threads. This is not applicable for multicore architectures, because differently prioritised threads are able to execute at the same point in time, as long as they are scheduled for different CPU cores. This applies for both,

ISBN 978-3-88579-286-4

static thread/CPU affinity and dynamic assignment based on load balancing algorithms as facilitated for symmetric multiprocessing (SMP) by an RTOS scheduler as depicted in Figure 2. A further scheduling instance is needed, which grants access to shared resources other than the CPU for defined groups of tasks (execution domains), based on predefined priorities. This allows an integrator of the system's software components, to set priorities that are based on the semantics of the components, independent of their internal prioritisation of threads and implementation details. Resource control can be implemented at integration time to reflect the required behaviour of the system as derived from the specified use-cas

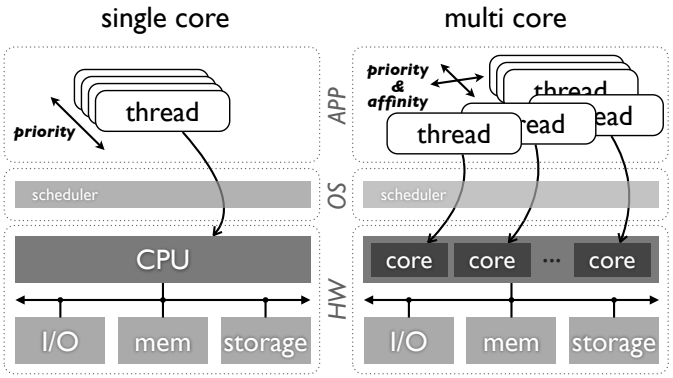

Figure 2: Thread scheduling for different architectures.

1.3 Related work

For shared resources, only limited priority-based access control is usually provided by common embedded operating systems. The Linux kernel allows the use of an I/O scheduler for block-oriented devices, mainly targeted to improve the performance for concurrent disk access. The driver of such random access devices implements a job scheduler to hide latencies, for example for a hard disk: the I/O scheduler virtualises the disk among multiple outstanding requests from different client applications to reduce disk seeks and improve overall performance [Lov10, p. 297-304]. Therefore different implementations are available, even with the ability to adjust request orders by use of priorities. They all have in common that they are only targeted for block devices and do not support stream oriented resources.

The QNX Neutrino RTOS provides a concept for resource managers with its driver architecture. Each driver abstracts the access to a system resource, represents an individual domain of authority, adds functionality to the operating system, but lives outside the kernel space [Hil92]. This approach would provide an appropriate base for implementing a resource scheduler, but also would imply incompatibility with other operating system architectures, e.g. the Linux kernel.

The Multicore Association is establishing a set of standards for the efficient use of multicore platforms [HAB+09]. They advocate with their Multicore Resource API (MRAPI) specification a comprehensive interface for the cooperative use and management of shared resources in embedded systems. As for their self-set goals, "MRAPI intentionally stops short of being a full-featured dynamic resource manager capable of orchestrating a set of resources to satisfy constrains on performance, power, and quality of service" [Mul10, p. 9]. This means that the MRAPI is not usable as a resource scheduler, but could support the development of one.

The separation of heterogeneous software components can be pushed further by using virtualisation, where each software domain or set of tasks with a different class of time constraints is assigned to a dedicated virtual machine (VM). This field of research is being pursued at the ICM Labs [VWSD10]. Such system structure comes with additional overhead for the virtualisation and does not resolve the concurrent use of shared resources, which has to be managed at the level of the virtual machine monitor (VMM). Additionally, several virtual machines do compete for shared resources as well, which makes the approach of an resource scheduler also applicable to a system architecture that relies on virtualisation, in particular to the VMM layer. An architecture utilising both multicore and virtualisation is able to offer a combination of advantages for structuring heterogeneous systems [VKW].

With the AUTomotive Open System ARchitecture (AUTOSAR) a consortium consisting of automobile manufacturers, suppliers and tool vendors is developing a standardised automobile software architecture, a development methodology, and standardised application interfaces [Hei11]. With release 4.0 AUTOSAR specifies a multicore OS architecture. An emphasis is placed on mechanisms to enable communication between applications located on different cores. As of the current requirements resources can not be shared between tasks that are placed on different cores [AUT10, p. 45]. This limitation affects the degree of freedom for allocate independent software components on different CPU cores, while the components are utilising shared resources. Such a requirement reduces the complexity introduced with concurrent resource access. For ICM systems consisting of multiple independent components the disadvantage caused through that restriction prevails.

Within the following sections, both requirements and a prototype implementation show the applicability of a resource scheduler for highly integrated embedded systems based around the example of ICM systems. The remainder of this paper is structured as follows:

Section 2 outlines requirements to be fulfilled by a resource manager.

Section 3 provides details of a prototype resource manager, which reflects the requirements of the previous section, as well as a first evaluation of the achieved results.

Section 4 summarises the information provided and enumerates open issues to a comprehensive framework supporting the development of a heterogeneous and component based embedded software system.

ISBN 978-3-88579-286-4

2 Requirements for a resource scheduler

To provide a beneficial resource management facility that can be located between the operating system and application components within a layered system architecture, a set of requirements has to be fulfilled. These may represent the starting point for a successful implementation. As the basis for the requirements and implementation, an SMP based multicore system with heterogeneous and general-purpose cores utilising a shared memory domain for efficient inter-component communication is used.

Interoperability is one major goal to achieve for interdependent heterogeneous system components. Therefore a common set of standards for multi threaded application development has to be chosen. The Portable Operating System Interface (POSIX) system API is of mature quality, proved applicability, and provides adequate portability to facilitate a reuse for software components [POS08][Wal95]. Furthermore, POSIX provides comprehensive support for parallel programming with the threading library 'pthread', which is also proven in current ICM system implementations. This means an implementation of both the resource scheduler and the application components can be independent of a certain (UNIX like) operating system, as long as it complies with the POSIX programmers interface. Additionally, the resource manager appears transparent to the application level programmer. This means that a single application component should not notice whether a device access abstraction is used or not (at least from the functional point of view). This enables the component supplier to also make use of the application software without the resource manager, as long as an uncontrolled access does not interfere with the application's implemented logic.

To ensure the portability of the scheduler, no changes to the operating system must be made. This implies that the resource scheduler has to be implemented in 'user space' (in contrast to a 'kernel space' implementation), which follows the concept of microkernel-based service architectures, as for example followed by the QNX Neutrino RTOS or the L4 microkernel [Hil92][Ruo08]. Further, the loose coupling from the internals of the operating system's kernel avoids the need for the maintainer of the resource scheduler to keep track of modifications of the implementation of the kernel.

For software architectures that are linked at runtime to system libraries (as for example libc, librt, or libpthread), the code base of the resource scheduler has to be separated from those libraries with a focus on portability. This means that the resource scheduler must not rely on any custom modifications within system libraries maintained by third parties. Although dynamic linking may conflict with a deterministic and fast system start-up, it supports the integration process by making use of a single system API for delivered software. ICM systems are composed of various artefacts provided by subcontractors, including binary executables and legacy software that cannot be modified. Further the system probably has to be optimised with a focus on the size needed for the executable code. Therefore dynamic linking can be considered as a trade-off for embedded systems, based on heterogeneous components. To make use of the resource scheduler, no modification within the source code of the software components and no rebuild of binary application files should be necessary.

ISBN 978-3-88579-286-4

Further, the targeted resource scheduler has to comply with the constraints of embedded systems, which implies a lightweight implementation in terms of latency and memory footprint. Device access has to be abstracted while maintaining a deterministic overhead to support timing constraints of the application layer.

A central feature of the resource scheduler is the ability to grant access to shared resources. Therefore statically defined priorities are necessary to reflect the specified time behaviour of the overall system. These may be defined based on the information tuple 'execution domain' and 'resource' for being able to reflect the specified behaviour of the targeted system.

The requirements for a resource scheduler outlined above can be condensed to the following list. In no particular order:

- The resource scheduler should conform to the POSIX programming interface.
- It should appear transparent to the application level programmer.
- It should not require any modifications to the kernel of an operating system.
- It should not require any modifications to deliverables provided by suppliers, which includes source code, compiled object files, and linked executable files.
- It should make use of deterministic overhead.
- It should grant access to registered resources based on predefined priorities.

3 Implementation and evaluation of a prototype resource scheduler

Derived from the presented requirements, a prototype resource scheduler was implemented in order to prove their applicability. The prototype was incorporated into the OpenICM, which is an embedded framework to facilitate support for application programmers developed at the ICM Labs [WT05][ICM10].

Initially, the implementation is targeted for the operating systems GNU/Linux and QNX Neutrino RTOS, by using the POSIX programming interface. Therefore portability to other software platforms is ensured. Further the prototype provides the functionality by interposing the relevant POSIX calls. The list below does not claim to be exhaustive, but enumerates the necessary calls to realise a proof-of-concept implementation by use of a character device and complies with the requirements defined in the previous section (see [POS08] for a detailed documentation of the calls):

- open/close (open/close I/O device)
- read/write (read/write data from/to I/O)
- select (synchronous I/O multiplexing)

ISBN 978-3-88579-286-4

- ioctl (device control functions; selective request types only)
- tcsetattr/tcgetattr (set/get parameters associated with a terminal)

Further, calls affecting the access of shared resources can be intercepted to prevent misuse which interferes with the resource scheduler, or passed through to the primary implementation of the standard library provided by the underlying system.

3.1 Interposing

To interpose the new functionality, the preload capability of the dynamic linker at runtime is used to overload the relevant symbols. This technique still allows fall back to the primary implementation, as for example provided by the system's C library (libc). The resource manager is applicable even for binary executables without the need to re-compile or re-link. As an alternative to dynamic interposing at runtime, the wrapping ability of the linker during compile time is usable, as long as sufficient artefacts to re-link are available to the

although
ts to the
ow us to
lification

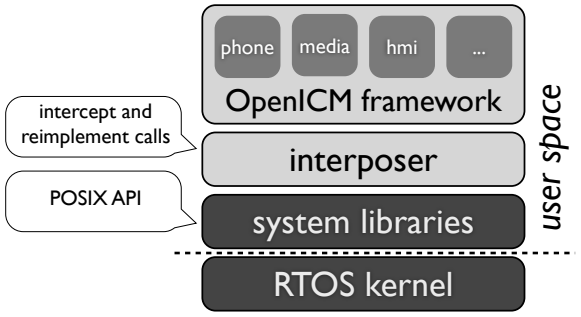

Figure 3: Layered system overview.

3.2 Internals

The application domains subscribe to certain shared resources by use of the 'open' call. This issues a registration at the resource scheduler, which manages a list of all subscribers, and is able to connect them to the related device abstractions provided by the underlying operating systems. Depending on a predefined prioritisation configuration, it grants access

for calls (for example: read, open, select, ioctl) which access the appropriate resource to the registered application component. The configuration of the priorities is set up during the integration process, independent from the development of the application components. This is facilitated through the concept of component contexts which are provided by the OpenICM framework [WT05, p. 287 ff.]. The software components (and their threads) can be identified through information stored within a shared memory structure that is populated on system startup. The resource priority is assigned to pairs of 'component' and 'resource'. When accessing a resource by use of a system call, the interposer identifies the caller through the component context information in order to determine the predefined priority related to the called resource. This allows the integrator to arrange the calls using a predictable order.

The parallel programming capabilities of the underlying software system provides the necessary support by the use of the POSIX threading library. As described above, resource priorities can be mapped to thread priorities for servicing the registered subscribers, utilising the available scheduler of the kernel. This allows the usage of mechanisms of the up-

in
er
u-
al

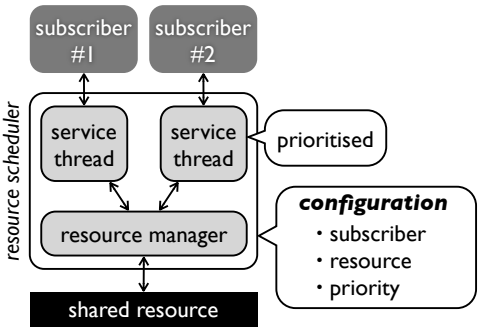

Figure 4: Concurrent access to shared resource using the resource scheduler.

By the use of the standardised POSIX API the resource scheduler is not limited to a finite number of resources, which makes it as versatile in usage as the primary API. The application programmer does not encounter any additional limitations.

3.3 Benchmarking

Due to the transparency of the application layer, an evaluation of the resource scheduler can be managed by use of common and available benchmark tools. Figure 5 show the latencies that were recorded with the micro-benchmark tool LMbench [Sta05] (test envi-

ronment: GNU/Linux on Intel Xeon E5504). Using the application 'lat_syscall' as part of the aforementioned benchmark suite, 'open' causes an overhead of 2.0% (0,042 µs) and

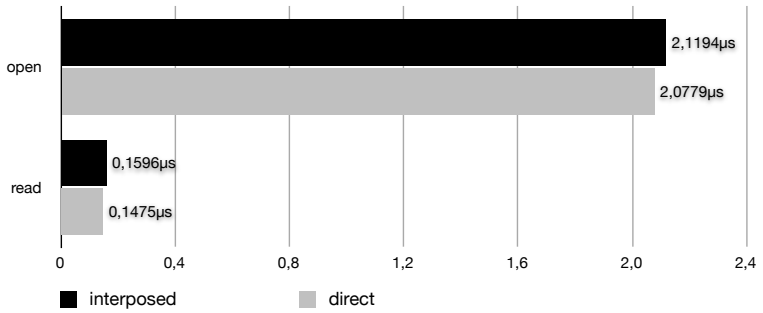

Figure 5: Call latencies in µseconds.

The code base of the prototype implementation's source is less than 90 kByte (<290 kByte binary shared object file; not optimised for size; reusing some components of the OpenICM framework). This positively affects the maintainability and keeps the requirement for persistent memory at a decent level.

4 Conclusion and outlook

The availability of multicore architectures within the automotive multimedia domain provides new opportunities to integrate heterogeneous software components on a single hardware platform. With future applications, in part driven by connectivity to wireless access networks, new solutions are needed for large-scale software projects such as ICM systems. This paper raises the issue of the need for priority-based access to shared resources with different time constraints, introduced by the parallel execution of software domains. Therefore a resource scheduler is proposed, defined by a set of substantial requirements. The applicability has been proven by a prototype implementation, including the results of a first evaluation using micro-benchmarks.

The evaluation performed using LMbench provides a first impression of how a system using the proposed resource scheduler might behave. Given that micro-benchmarks only measure a very specific unit of the system, more elaborate test scenarios, derived from real-world use-cases, have to be established in order to provide additional evidence on

ISBN 978-3-88579-286-4

applicability.

A goal of the research at the ICM Labs is to combine the necessary techniques and units into a comprehensive software framework, which provides flexibility to the developer and mitigates the risk of non-deterministic integration efforts. The resource scheduler represents an essential unit of such a framework that utilises current multicore hardware architectures.

References

[AUT10] AUTOSAR. Specification of Multi-Core OS Architecture. V1.1.0, Rel 4.0, Rev 2, 2010.

[BRR11] Manfred Broy, Günter Reichart, and Lutz Rothhardt. Architekturen softwarebasierter Funktionen im Fahrzeug: von den Anforderungen zur Umsetzung. *Informatik-Spektrum*, 34:42–59, 2011. 10.1007/s00287-010-0507-6.

[DNSV10] Marco Di Natale and Alberto Sangiovanni-Vincentelli. Moving From Federated to Integrated Architectures in Automotive: The Role of Standards, Methods and Tools. *Proceedings of the IEEE*, 98(4):603 –620, April 2010.

[HAB+09] Jim Holt, Anant Agarwal, Sven Brehmer, Max Domeika, Patrick Griffin, and Frank Schirrmeister. Software Standards for the Multicore Era. *IEEE Micro*, 29:40–51, 2009.

[Hei11] Khosrau Heidary. AUTOSAR Technical Overview and Future Development Roadmap. 3rd AUTOSAR Open Conference, May 2011.

[Hil92] Dan Hildebrand. An Architectural Overview of QNX. In *USENIX Workshop on Microkernels and Other Kernel Architectures*, pages 113–126, 1992.

[ICM10] ICM Labs. OpenICM Framework. Technical report, Faculty of Computer Science, University of Applied Sciences Darmstadt, 2010. http://fbi.h-da.de/~openicm, last checked 19.04.2011.

[Kai06] Robert Kaiser. Koexistenz unterschiedlicher Zeitanforderugen in einem gemeinsamen Rechensystem. In *Echtzeitsysteme im Alltag*, Informatik aktuell, pages 16–25. Springer Berlin Heidelberg, 2006.

[KWMD10] Andreas Knirsch, Joachim Wietzke, Ronald Moore, and Paul S. Dowland. An Approach for Structuring Heterogeneous Automotive Software Systems by use of Multicore Architectures. In *Proceedings of the Sixth Collaborative Research Symposium on Security, E-learning, Internet and Networking (SEIN 2010)*, pages 19–30, Plymouth, UK, November 2010.

[Lov10] Robert Love. *Linux Kernel Development*. Developer's Library. Pearson Education, 3rd edition, 2010.

[Mul10] The Multicore Association, Inc., El Dorado Hills. *Multicore Resource API (MRAPI) Specification*, 1.0 edition, 2010.

[POS08] IEEE Standard for Information Technology - Portable Operating System Interface (POSIX) Base Specifications, Issue 7. *IEEE Std 1003.1-2008*, January 2008.

[Ruo08] Sergio Ruocco. A Real-Time Programmer's Tour of General-Purpose L4 Microkernels. *EURASIP Journal on Embedded Systems*, 2008:7:1–7:14, April 2008.

[Sta05] Carl Staelin. lmbench: an extensible micro-benchmark suite. *Software: Practice and Experience*, 35(11):1079–1105, 2005.

[SVDN07] Alberto Sangiovanni-Vincentelli and Marco Di Natale. Embedded System Design for Automotive Applications. *Computer*, 40(10):42–51, October 2007.

[VKW] Sergio Vergata, Andreas Knirsch, and Joachim Wietzke. Die Integration zukünftiger In-Car Multimedia Systeme unter Verwendung von Virtualisierung und Multi-Core Plattformen. In *Eingebettete Systeme*, Informatik aktuell. Springer Berlin Heidelberg. forthcoming.

[VWSD10] Sergio Vergata, Joachim Wietzke, Alois Schütte, and Paul S. Dowland. System Design for Embedded Automotive Systems. In *Proceedings of the Sixth Collaborative Research Symposium on Security, E-learning, Internet and Networking (SEIN 2010)*, pages 53–60. University of Plymouth, November 2010.

[Wal95] Stephen R. Walli. The POSIX Family of Standards. *StandardView*, 3:11–17, March 1995.

[WT05] Joachim Wietzke and Manh Tien Tran. *Automotive Embedded Systeme*. Xpert.press. Springer, 2005.

ISBN 978-3-88579-286-4