Efficient Memory-Protected Integration of Add-On Software Subsystems in Small Embedded Automotive Applications

Akram Khan, *Member, IEEE*, Achim Schäfer, and Markus Zetlmeisl

***Abstract*—Current innovations in the automotive industry evolve mainly in the electronics and software domain. This leads to an increasing integration of additional software subsystems into already existing electronic control units (ECUs) to cope with the raised amount and complexity of present ECUs in modern high-end vehicles. This paper discusses different approaches which are required to integrate such add-on software subsystems in an isolated memory domain, and considers particularly the special needs of small embedded systems—including the limited hardware support. Special focus is brought to the efficient detection of malicious memory accesses, as well as the benefits of a thereupon possible and adaptable failure-handling strategy. All investigations are based on a developed memory-protection framework which has been tailored to the special needs of a sample vehicle dynamics control system. Its usage allows the combination of. integrating additional subsystems without reducing the main application's availability.**

***Index Terms*—Add-on software integration, embedded automotive systems, memory-protection unit, subsystem partitioning.**

I. Introduction

The TREND in today's automotive systems is to integrate several independently developed software and hardware subsystems into larger electronic control units (ECUs). Subparts like engine management, cruise and gearbox control, sensors, and actuators grow together to a global vehicle control system. The advantage of this approach is a limitation of the extensively increased amount of ECUs, which reduces the costs significantly (e.g., reduced installation space). Furthermore, the more powerful ECUs now make it possible to develop and integrate additional add-on functionalities—mainly realized in pure software algorithms—which require only the already available system internal signals. Such subsystems naturally have a small interface to the main application and can therefore be developed by external suppliers (third-parties), which have, for example, deep knowledge in special market niches. This gives the automotive manufacturers and suppliers the possibility to provide additional add-ons (e.g., comfort values), which are not interesting to develop in-house but fit well to their offered portfolio.

However, to integrate and run such subsystems in parallel to the core application raises additional issues which must be handled explicitly. The main application, which was often designed to run exclusively onto the ECU [(with only a few limited network interfaces to other ECUs; e.g., a controller area network (CAN)], must now share its resources with additional independent subsystems. To prevent possible subsystem malfunctions, which could have impact on the core application's stability, and thus availability, requires the usage of additional partitioning concepts.

This paper focuses on memory-protection issues and describes different approaches and mechanisms which are required to integrate such subsystems in an isolated memory-protection domain. Prevention, detection, and proper reaction to any illegal memory accesses outside the assigned domain gain several benefits which obtain the required system's availability. All investigations are based on a developed memory-protection framework which considers the special constraints of such "small" high-end vehicle dynamics control systems where common virtual memory concepts are often too heavyweight and costly to realize (resources and runtime). Runtime monitoring (e.g., time budgets for different subsystems)—which is also a quite important issue for an overall partitioning concept—is not considered here.

The paper starts with an overview of the target applications and highlights their constraints especially regarding hardware and software design. Section II introduces the developed memory-protection framework with its realized internal mechanisms, and in Section III, the achieved failure-handling strategies are described. The paper ends with a performance evaluation and conclusion about the investigations.

II. Systems Constraints of Current Automotive ECUs and Design Requirements

This section describes some general design issues which must be considered in the integration of add-on software subsystems. It highlights the main differences between running exclusively the main application on the ECU and the sharing of the controller resources with supplier software functionalities.

A. Overview

Former vehicle-dynamics control systems had a strong emphasis on hardware functionality and the developed software was tailored on its special needs. Tight cost constraints due to strong market competition have been reflected to the software design so that many systems have used the available microcontrollers in the most efficient way and utilized the resources (RAM, ROM, runtime) at the upper limit. The combination of

Manuscript received December 5, 2005; revised May 10, 2006 and July 20, 2006; accepted October 30, 2006. Paper no. TII-05-12-0078.R2.

A. Khan is with Brunel University, Uxbridge UB8 3PH, U.K. (Akram.Khan@brunel.ac.uk).

A. Schäfer and M. Zetlmeisl are with Robert Bosch GmbH, Business Unit CC (Chassis Systems Control), Stuttgart D-70174, Germany.

Digital Object Identifier 10.1109/TII.2006.890522

resource efficiency, required system specific complex degradation and fallback levels, and the support of different customer configuration requirements result mainly in a monolithic design structure of the core application. Hence, the system focus was rather on a sort of software product line approaches ([1]) than applying "costly" internal memory-partitioning concepts, which additionally do not gain enough benefits as long as the main application was running exclusively on the ECU.

However, with the increased need to integrate add-on software and run it in parallel to the main system, it is often mandatory to prevent—at least—the core application during runtime from any external malicious accesses and to clarify responsibility in case such a failure has happened. But it is important to distinguish between common public computer systems and the target embedded automotive systems. All processes in such embedded environments are designed to cooperate, which means they never try to intentionally harm each other—like a worm or a virus in a desktop or server system does. Another big difference is the missing dynamic addition and removal of functions during runtime, compared with public computer systems. However, small software bugs in a less important module, which are difficult to find and may occur only in some specific situations, could potentially corrupt the whole system (e.g., invalid pointer arithmetic in the C programming language).

B. Memory Management Constraints

With the introduction of high-performance microcontrollers and their integrated memory management units (MMU), many embedded systems benefit from the possible advantages of virtual address spaces or isolated partitions, and can easily set up these mechanisms ([2], [3]). However, as already mentioned, there exist a large amount of small embedded systems where the additional overhead is not worthy (e.g., explicit OS support; increased runtime and memory consumption). Such real-time applications are often allocated on the ECU in a predefined and static way, and even dynamic memory allocation is often not possible due to system and resource constraints. Partitioning the monolithic main application in several dedicated subsystems and providing a sort of protection would be too cumbersome and resource consumptive as failures in these small partitions affect the consistency of the whole system and require, anyway, a global shut-down strategy. Even if a MMU is available on the microcontroller, it is often either bypassed or configured in a straightforward way to save the rare resources.

Beyond the common and flexible MMU systems, there exist several microcontrollers which aim for the currently more and more important memory-protection issues and integrate special—for the target applications designed—memory-protection units (MPUs). They monitor memory accesses against given regions—specified by MPU channels—in a very efficient way and consider the application's possible address space, memory granularities, and peripheral interconnections (e.g., CAN mailboxes, general input/output registers). The overhead in applying such pure MPUs requires fewer resources than setting up a complete virtual memory system.

The developed memory-protection framework will apply an efficient protection scheme to isolate selected multitasking add-on software subsystems on top of such a MPU.

C. Operating System Concepts and Memory Protection

The OSEK standard (German: Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug; English: open systems and the corresponding interfaces for automotive electronics; [4]), published by a consortium of automotive companies, specifies operating systems which are adapted especially for static automotive applications. Even very small applications often use such operating systems to benefit from the specified and efficient implemented system services like multitasking scheduling or alarm and message handling.

Unfortunately, no subsystem protection features are considered in the current standard until now. However, research projects like APOS ([5]) face such issues and have developed different features of an OS supported memory protection and execution time monitoring. As these concepts move the whole memory-protection responsibility to the operating system, they restrict the protection granularity to the smallest available OS scheduling units—which are tasks according to the OSEK standard. However, the design of many small automotive applications is often based on a limited amount of cyclic activated tasks with different prioritized time bases. In-between theses tasks many independent functions run in a pure sequential way and with the same priority (sometimes called processes; [6], [7]). This OS configuration fulfils, on one hand, the cyclic activation requirements of the different independent controller applications and reduces, on the other, OS overhead since only a limited amount of scheduling units must be handled in the system. As a result, a subsystem may consist of several functions, which are executed in different tasks but share a single task with other subsystems.

The idea behind the developed memory partitioning framework is to work mainly independent of the OS, and to integrate any subsystem in a standard way without major changes in the scheduling mechanism. This gains not only more flexibility and high efficiency, but achieves also the protection of any arbitrary small execution unit instead of using OS tasks (i.e., protection on the function level).

However, developing a memory-protection framework independent from the used OS raises several system internal issues which must be considered explicitly in the design phase and are highlighted in the following sections.

1) Multitasking Scheduling Support: OSEK defines two different task models: The simpler one is the basic task state model and allows task activations in a cyclic way or by explicit calls, whereas the second concept—extended task state mode—allows the usage of an extended event mechanism. Operating systems which implement the extended task state model require a more complex task management and therefore more resources than systems supporting only the basic model.

The memory-protection framework concept focuses on the more efficient basic task state model (so-called BCC conformity class) because the majority of the target systems are based on this strategy. So it must cope with these preemption mechanisms to configure the execution environment properly and in a transparent way to the OS.

2) Processor Mode Handling: Modern embedded microcontrollers are able to execute the code in several architecture dependent processor modes—basically differentiated in privileged and unprivileged. However, running only one main application (like the target automotive systems) makes it often inefficient to use both modes. At least the OS and the main drivers need unrestricted access to the whole microcontroller to configure the peripherals during runtime (e.g., CAN or DMA, direct memory access). Explicit mode switches for each privileged access would result in unnecessary resource overhead. Furthermore, since the main application has, for efficiency and configuration reasons, usually a monolithic structure it is neither feasible to run parts of it in an unprivileged mode nor would it gain any benefits because the whole core must be consistent to initiate one of the complex system degradations after a failure.

However, this strategy changes with the integration and isolation of add-on software subsystems. In that view, the main core application and the OS are of higher priority than the subsystem and must be protected against the unprivileged parts. The framework must cope with issues that arise due to different execution mode preemptions.

3) Stack Usage: To reduce runtime and memory overhead, as well as stack configuration effort, small embedded systems often use one large single stack for the whole application rather than having for each task a separate one.

The stack handling strategy must be considered when executing the add-on subsystem transparent to the OS because it raises issues concerning stack protection and restoration after an illegal access. There are mainly two approaches.

1) Share the single stack with the add-on software.
2) Provide an own stack for each subsystem, independent in which tasks it is running.

The first approach requires additional measures to protect dynamically parts of "core" stack during subsystem execution and needs more effort after a detected illegal access. The second, on the other hand, is more efficient concerning protection and restoration but requires some enhanced stack exchange mechanisms—which are normally in the OS's responsibility. System constraints due to a limited amount of available MPU channels enforce the realization of the second concept for the memory-protection framework (separate stack for each subsystem).

III. Memory-Protection Framework Overview

The next section describes briefly the realized concepts of the memory-protection framework as well as the subsystem integration. It starts with an introduction of the used environment (main system, microcontroller, OS) before topics regarding the MPU and the required OS extensions follow. The advanced failure-handling strategies will be highlighted in a separate section.

A. Existing System Environment

The sample memory-protection framework has been developed on top of an embedded vehicle dynamics control system, running an open market 32–bit RISC microcontroller with an ARM7 core (Texas Instruments TMS470; [8], [9]). The available hardware MPU has four independent MPU channels with a granularity of 32 bits. Access violations result in an immediate branch to the core's data exception handler. The application runs on top of the underlying OSEK oriented real-time operating system ERCOS^{EK} ([6], [7]).

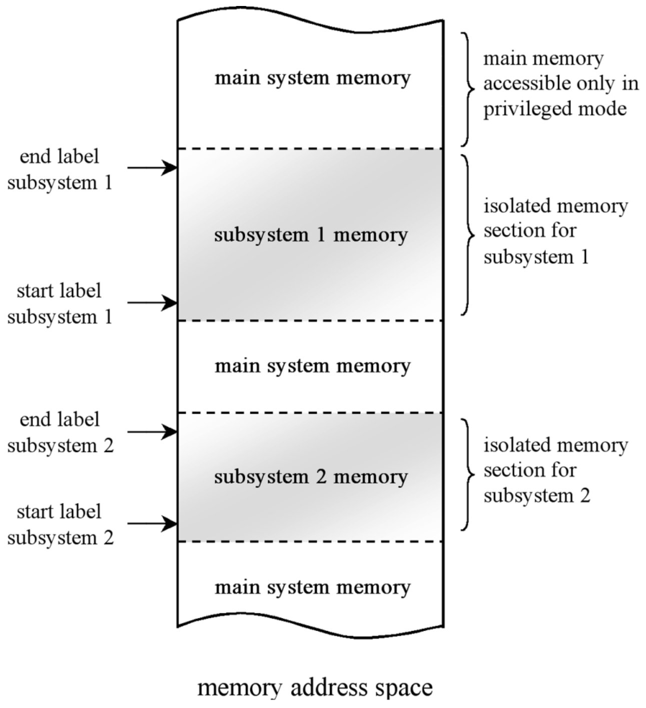


Fig. 1. Memory map with two independent subsystems.

As already mentioned, the main application is distributed over several tasks which share the same single stack. Furthermore, the operating system and the main application as well, run completely in the privileged processor mode to reduce explicit mode switch overhead.

B. Subsystem Memory Isolation and MPU Channel Usage

To apply the limited amount of available MPU channels in an efficient way, it is required to link the subsystem's required memory in one contiguous RAM section. This range is surrounded by two labels which allow a dynamic configuration of the HW MPU channels. Only the window between these labels is accessible for unprivileged memory accesses during subsystem execution. Furthermore, to cope with task preemptions, it is required that the whole memory is permanently opened for privileged accesses. This reduces the MPU channel reconfiguration overhead during runtime because otherwise neither the OS nor the privileged main application would have access to their private memory until the channels are set properly. Fig. 1 shows a sample configuration with two independent subsystems.

C. Framework Subsystem Wrapper

Since the memory-protection framework runs mostly independent of the operating system, it must explicitly configure the execution environment before starting each subsystem's function. This is done in a special wrapper which prepares the system (precall), invokes the function, and reconfigures the execution environment afterwards back to its original state (postcall). Required actions to isolate the subsystem properly are as follow.

- Preserving the previous MPU configuration and opening the current subsystem's dedicated memory for unprivileged access. The restored configuration in the postcall function ensures that only the subsystem's dedicated

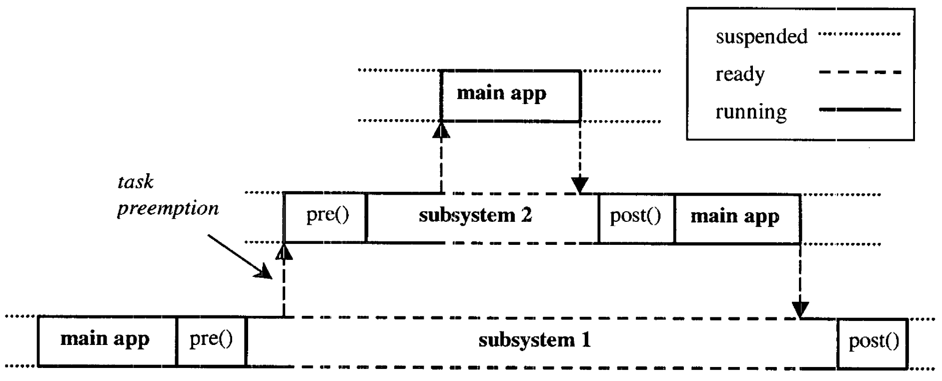


Fig. 2. Multitasking main application and subsystem execution.

memory is accessible when it proceeds (in case an isolated application has been preempted).

- Preserve all processor registers to ensure that no manipulations—even on register basis—are transported in case of an abort from the subsystem to the main application (see Section IV on failure-handling strategies, which covers this topic more in detail).
- Switch to the subsystem specific stack to keep the main stack free of any subsystem contents. This has the advantage that the main stack does not need to be restored after a detected malicious memory access and the following subsystem abort.
- Switch explicitly to the unprivileged processor mode and invoke the subsystem function. The MPU channels prevent access to any memory locations outside the predefined range, including peripheral and system-configuration registers.

The described measures have to be reversed in the corresponding postcall function after the subsystem has finished. Switching back to the privileged mode is done using a special OS service call which makes use of the microcontroller's internal mode switch mechanisms.

Fig. 2 shows a scenario where two different subsystems and the main application run with different priorities (i.e., in different tasks). The subsystems are invoked in-between the main software tasks, requiring no additional OS scheduling units (e.g., extra time table entries). The only extensions are the small framework's pre- and post-call functions which prepare the execution environment for each subsystem function call. The next section describes the subsystem preemption process more in detail because it requires additional framework mechanisms.

D. Unprivileged Subsystem Preemption

Preemption of a lower priority task (e.g., currently executing an unprivileged software subsystem), which is in the used operating system ERCOS$^{\mathrm{EK}}$ always initiated by the same single interrupt, requires two explicit OS features to interact later-on with the framework, as Fig. 2 shows. Firstly, it is required that each new task is started explicitly in the privileged mode, which can be configured in the OS setup. This allows access of the higher priority SW to the whole address range until a new subsystem is started, using the framework's pre-call wrapper. Secondly, it must be ensured that the operating system preserves and restores the interrupted unprivileged processor mode, so that the subsystem can proceed in the correct mode after the higher priority task has finished. This is generally part of every multitasking OS' interrupt context handling which considers also further runtime context information (e.g., processor status flags).

The decision to spend a separate stack for each isolated subsystems requires the installation of a short routine in the interrupt handler which switches to the main stack in case of an unprivileged preemption. The action must be reversed (switch to the dedicated subsystem stack) before returning to the interrupted subsystem software.

E. Configuration Structure

The framework sets-up and maintains for each integrated subsystem a single configuration structure which consists in the current implementation of the accessible memory range, the assigned private stack pointer, and—as explained in the next section—information about the software's execution status. This status shows, among other things, if the software forced an illegal memory access, which is especially important for the required failure-handling strategies.

IV. Failure-Handling Strategies

The failure-handling strategy after a detected malicious memory access depends strongly on the integrated subsystem. If it is only responsible for some minor comfort functionalities, a less restricted strategy can be applied compared with an important information processing algorithm. Consequently, the framework provides mechanisms which allow a flexible system integration and achieve therefore project specific independence.

A. Implemented Framework Mechanisms

The most important requirement is the immediate abort of the running subsystem because its consistency after the abnormal memory access cannot be ensured. A branch directly to the specific postcall wrapper (behind a subsystem call) is initiated by the exception handler to prevent any malicious side effects to the main system and to restore the execution environment. The following advantages result from this behavior.

- *MPU channel reconfiguration*: The restoration of the previous MPU channel configuration ensures that other integrated and currently preempted subsystems can proceed without any inconsistencies. It allows an independent

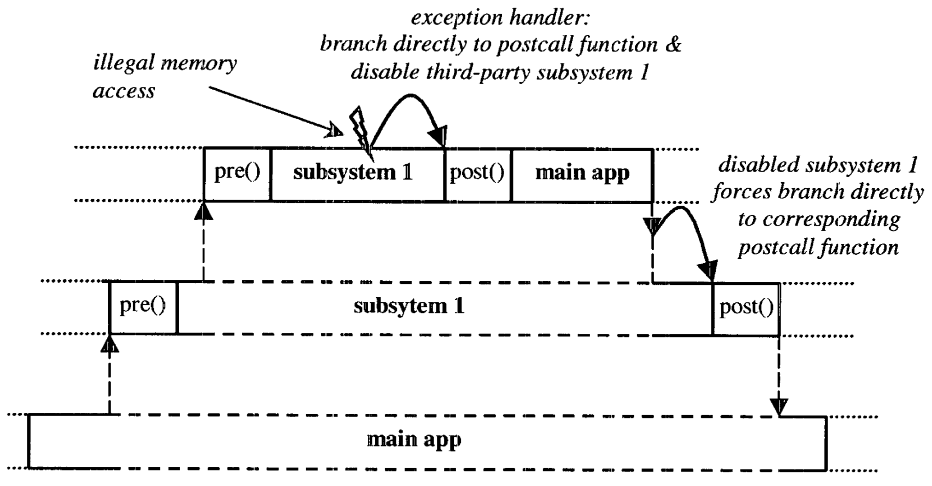

exception handler:
branch directly to postcall function &
disable third-party subsystem 1
illegal memory access
pre()
subsystem 1
post()
main app
disabled subsystem 1 forces branch directly to corresponding postcall function
pre()
subsytem 1
post()
main app

Fig. 3. Cascading subsystem abort.

failure handling of each subsystem in which only the malicious one will be aborted.

- *Register set restoration*: The restoration of the controller's register set to the state before invoking the function prevents any inconsistencies on register basis to the main application. This concerns mainly the compiler specific register usage—like save-on-entry registers ([10]).
- *Stack exchange*: The usage of a separate stack for each subsystem allows an efficient switch to the protected and therefore consistent main stack without any additional required clean-up work. The application can proceed without worrying about manipulated stack values (e.g., function return addresses).

Beyond these issues, a detailed look at the multitasking support and the possible integration of one subsystem in several tasks shows another problematic situation. It could be possible that a higher priority process forces a failure while a lower priority function of the same subsystem has been preempted. This would result in the correct immediate abort of the present one but the operating system would later-on proceed with the interrupted process, which would come across with an invalid environment (e.g., malicious SW had overwritten subsystem specific variables or stack contains invalid values). This scenario shows that an additional mechanism is required to realize a so-called cascading abort where the whole chain of all functions belonging to the same subsystem has to be aborted. The framework implements a small and efficient logic which is invoked before the operating system proceeds with the execution of the interrupted software to cope with this situation. It decides, evaluating the integrated subsystems execution status, if the interrupted software is allowed to proceed or if a branch to the corresponding post-call wrapper is required.

Fig. 3 shows this cascading abort scenario where a preempted function of the same subsystem aborts after a higher process has forced the failure.

B. Main-System Failure Handling

The automated framework mechanisms with its immediate abort and the restoration of the execution environment have the advantage that the main subsystem can proceed without any side effects and does not need to consider any special system internal properties (register set; switch to main stack; preemption issues). After returning from the subsystem call, it can instantly evaluate the subsystem's execution status, check if it has been aborted, and react properly.

The first and most obvious action is to prevent the future execution of the subsystem. This depends on the integrated subsystem's timing requirements and can include cyclic calls as well as invocations via asynchronous events. Furthermore, the calling function—which is normally a wrapper who prepares the interface for the subsystem—must ensure that the interface to the main system is set to a defined state. Otherwise, the initiated abort would prevent memory manipulations correctly, but the inconsistent interface would transport errors in form of invalid data to the main system.

The next steps after these low level operations (interface handling, function call) are the reactions on system level. As already stated, there exist several degraded ways which are dependent on the subsystems responsibility.

If the main application requires necessarily the subsystem's information (e.g., a deeply integrated information processing) a strict, but defined, shut-down strategy is maybe the appropriate solution. The advantage in using the framework anyhow—even if such a hard reaction follows—is the availability and consistency of the mandatory required system functionalities, like global error handlers or watchdog mechanisms. Otherwise, it would not be ensured that these parts work properly and could result, in the first state, in an undefined system behavior which will later force the external safety trigger mechanisms and radically shut down the system.

Another way is to bring the main application into a reduced operating mode. This can be useful if the provided information is only necessary for some special system layers and does not make the whole application inoperative.

Furthermore, it is possible to proceed with the main application in normal mode and keep the subsystem completely switched-off. Examples are independent subsystems with no effect/feedback to the main application (e.g., flat-tire detection in a brake-control ECU).

The framework offers also the possibility to set up the subsystem from scratch for another trial. A central coordinator could reinitialize the add-on subsystem and manage afterwards the reintegration to the main system.

Particularly small and tight designed embedded applications benefit from the proposed approaches in offering higher main system availability even in case of memory violations out of an integrated add-on subsystem.

V. Implementation and Performance Results

A reference implementation based on the underlying system environment with the 32-bit TI TMS470 microcontroller (running at 60 MHz), the ERCOS$^{\text{EK}}$ operating system, and a typical vehicle-dynamics stability system (see also Section III-A, existing system environment) has been done to demonstrate the effectiveness of the described mechanisms. The system is basically working in six cyclically executed tasks with rate-monotonic assigned priorities and fix activation scheme. The software subsystems are executed within these tasks in a fix order per task, so-called processes. Most parts of the system run without active memory protection, i.e., with access to the complete address space. Single processes or even subsets of these are memory access restricted in this experimental reference implementation. Note that in this realistic embedded control application the scheduling differs from PC-based server/desktop application approach, where a system consists of memory separated applications using an MMU with virtual memory, each of which usually has a pool of threads with different priorities but are working on a shared application memory.

The experimental framework uses three out of four available hardware MPU channels, in which two are used to configure the subsystem base protection and one for application specific adaptations like subsystem inter-communication via shared memory.

The decision of the first prototype framework implementation was, in some cases, pro architecture and compiler independence rather than on heavy runtime optimizations. This involves mainly the interaction of written assembler and C routines to hide data-structure alignments. Future enhancements in these fields will bring additional optimizations of at least 20% in runtime and ROM consumption. The results are classified in runtime and memory overhead.

A. Runtime Overhead

The fix runtime overhead in preparing the system to execute a single subsystem function requires approximately 570 processor cycles whereas the phases precall and postcall are one half each (see Section III-C, framework subsystem wrapper). This is about 9.5 μs assuming a 60-MHz processor system.

The variable runtime overhead which is needed during task preemption depends on the interrupt load of the system (see Section III-D, unprivileged subsystem preemption). Processing of each interrupt requires the following additional resources.

1) Isolated subsystem is active/running: Overhead is required to switch to the default stack and to implement the cascading abort logic which needs approximately 192 processor cycles (about 3.2 μs).
2) Isolated subsystem is not active: Only a minimal interrupt overhead is needed in checking the current system state which requires 25 processor cycles (about 0.4 μs).

B. Memory Overhead

The framework requires a static overhead of about 1-kbyte ROM and 80-bytes RAM to implement the protection mechanisms. Additionally, each subsystem requires for internal data structures about 30-bytes ROM and 10-bytes RAM—which scales very well with the number of integrated subsystems.

C. Example Application

The isolation of a typical, for such small systems, example "dummy" application with an execution time of about 500 μs, a cyclic activation every 20 ms, and a global system interrupt load of 4000 INTs/s, results therefore in a runtime overhead during subsystem execution of nearly 16 us which is 3.2% based on the subsystem's execution time and an overall drop of 0.24% (47 μs) based on the cyclic task (including all interrupts). The internal structure of the example subsystem is comparable to a normal control application even if it does not perform any useful work during the tests—which has no impact concerning the performance results. This involves memory (including stack) and runtime usage as well as internal function decomposition.

The static runtime overhead of 9.5 μs can be directly charged to the subsystems pre- and postprocessing phase. Since these explicit phases are needed (and mostly accepted) in any case to serve the subsystem interface when integrating add-on software subsystems to a core application, only a small additional system overhead with 0.16% remains (minimum interrupt overhead).

Hence, the main advantage compared with a fully operating system supported approach with an estimated 30%–50% OSEK-OS overhead ([5], however with integrated timing protection) is the flexibility in isolating arbitrary small execution units with a minimum overhead for the remaining system, which can furthermore use the optimized OS version. This makes it particularly useful when only single subsystems of a larger applications have to be isolated.

VI. Conclusion

This paper highlighted different approaches which are required to integrate independent add-on software subsystems in small high-end embedded automotive applications. It showed that with the increasing need of such additional features, it is inevitable to set up mechanisms which prevent impacts to the core application. The focus in this paper was on the design of an efficient memory-based subsystem isolation which is able to prevent memory accesses outside the assigned domain and thereupon provides possibilities in initiating proper system reactions. Advantages are a higher main system availability, even in the case of hard memory-access violations, either forces by software or hardware failures. No side effects based on memory accesses, including stack and register usage, are transported from the malicious subsystem to the core application. Different failure-handling strategies can be configured in a project specific way, allowing flexible reactions dependent on the integrated subsystem's responsibility.

The introduced memory-protection framework shows that it is possible to set up flexible and quite efficient mechanisms with respect to tight and cost sensitive designed embedded system. It considers modern multitasking support regarding subsystem integration, as well as during the abort process (cascading abort). Independence and transparency to the operating system allows an efficient partitioning of arbitrary small protection units (i.e., on function call level), rather than using "heavyweight" OS scheduling parts (tasks). The achieved advantages regarding flexibility, isolation granularity, and configurable abort handling are more important for the target small embedded systems, where only a few dedicated subparts have to be isolated, than using pure and fixed OS support.

REFERENCES

[1] P. Clements and L. M. Northrop, *Software Product Lines: Practices and Patterns*, 1st ed. Boston, MA: Addison-Wesley, 2001.

[2] L. Tietz, "Virtual memory in embedded systems—Marketing hype or engineering reality," in *COTS J , On the Softer Side*, Sep. 2001, pp. 26–29.

[3] L. Tietz, "Under the hood of the ideal microkernel," in *COTS J., On the Softer Side*, Nov. 2001, pp. 25–31.

[4] *Open Systems and the Corresponding Interfaces for Automotive Electronics, Specification Version 2.2.2*, OSEK/VDX 2004 [Online]. Available: http://www.osek-vdx.org, Available

[5] K. Tindell, H. Kopetz, F. Wolf, and R. Ernst, "Safe automotive software development," in *Design, Automation and Test in Europe Conf. and Exhib. (DATE'03): Proceeding of an IEEE Computer Conference*, Munich, Germany, Mar. 03–07, 2003, pp. 10616–10623.

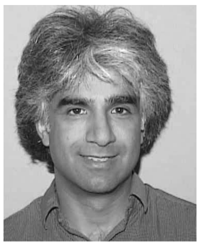
[6] S. Poledna, Th. Mocken, J. Schiemann, and Th. Beck, "ERCOS: An operating system for automotive applications," presented at the SAE, The Engineering Society for Advancing Mobility—Design Innovations in Engine Management and Driveline Controls (SP-1153), Detroit, MI, Feb. 26–29, 1996, Paper presented at the, unpublished.

[7] *ERCOSEK V4.0 User's Guide*, ETAS 2000, ETAS GmbH & Co. KG, 2000, Stuttgart, Germany, Document Number: EC110001 R4.0.2 EN.

[8] *Texas Instruments, Microcontrollers, TMS470*, [Online]. Available: http://www.ti.com, Available

[9] *ARM Architecture Reference Manual*, ARM 1996 [Online]. Available: http://www.arm.com, Advanced RISC Machines Ltd (ARM), 1996, Cambridge, U.K., Document Number: ARM DDI 0100B. Available

[10] *TMS470R1x Optimizing C/C++ Compiler User's Guide*, Texas Instruments 1999, Texas Instruments Incorporated, Apr. 1999, Dallas, TX, Literature Number: SPNU151A.

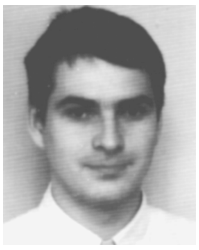


Akram Khan received the Ph.D. degree in experimental particle physics from University College, University of London, London, U.K.

He is a Reader in Grid Computing and in Electronic and Computer Engineering at Brunel University, Uxbridge, U.K. He is a Chartered Physicist with over 15 years of experience of working in large collaborative projects, which required the development of distributed computing technologies for the purposes of data analysis. He has held senior positions in some of the leading international centres for fundamental research (DESY, Femilab, and CERN) and is currently on leave of absence at Stanford University, Stanford, CA, as Director of the International Operational Computing Team. He is a member of the EU-funded EGEE, and the PPARC-funded GridPP collaborations. He is the Principal Investigator for the GridPP project for the development of middleware to enable the large distributed data and Monte Carlo analysis on the Grid. He was the Technical Director of the ScotGRID project for three years and is now the Deputy Director of the newly created Centre for e-Science and Multimedia Research, BITLab, Brunel University.

Achim Schäfer received the Dipl.-Ing. (FH) degree in computer engineering from the University of Applied Sciences Esslingen, Esslingen, Germany in 2003, and the M.Sc. degree (with distinction) in distributed computing systems engineering from Brunel University, Uxbridge, U.K., in 2005.

He is currently working for Robert Bosch GmbH, Business Unit CC (Chassis Systems Control), Stuttgart, Germany, as an Embedded Software Developer for vehicle dynamics control systems.



Markus Zetlmeisl received the M.S. (Hons.) degree in computer science from the Friedrich-Alexander University, Erlangen-Nuernberg, Germany, in 2002.

He was with Siemens Med and Siemens Corporate Research. He then joined 3Soft GmbH, Erlangen, Germany, as a Developer and Researcher in the fields of Windows CE and OSEK operating systems for automotive embedded applications. Currently, he is with Robert Bosch GmbH, Business Unit CC (Chassis Systems Control), Stuttgart, Germany, as a Software Architect for vehicle dynamics control systems. His main interests are in software architecture and design principles for embedded systems.