# An application software download concept for safety-critical embedded platforms

Christoph Dropmann, Drausio Linardi Rossi

**To cite this version:**

Christoph Dropmann, Drausio Linardi Rossi. An application software download concept for safety-critical embedded platforms. CARS 2015 - Critical Automotive applications: Robustness & Safety, Sep 2015, Paris, France. hal-01193016

**HAL Id: hal-01193016**
**https://hal.science/hal-01193016**

Submitted on 4 Sep 2015

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# An application software download concept for safety-critical embedded platforms

Christoph Dropmann
Fraunhofer IESE
Kaiserslautern, Germany
christoph.dropmann@iese.fraunhofer.de

Drausio, Linardi Rossi
Fraunhofer IESE
Kaiserslautern, Germany
drausio.rossi@iese.fraunhofer.de

Bastian Zimmer

***Abstract*—Application download is a promising concept for embedded systems in safety-critical domains such as automotive. Systems could be kept up to date without maintenance visits and new business models could be developed. However, the protection of safety-critical software against applications that are not known at system design time is rarely addressed. This paper presents a concept for downloading non-safety-critical software applications into a safety-critical system. The platform´s partitioning is realized via memory, execution time, and service protection.**

***Keywords—Embedded Application Download, Safety, Software Interferences, Segregation, Partitioning***

## I. INTRODUCTION

In recent years, we have witnessed the establishment of dynamic installation of software applications on electronic devices such as mobile devices and personal computers. Installing dynamic applications (new software at product lifetime) in the domain of critical embedded systems bears huge potential and could have a significant impact on our daily lives. Embedded systems could be kept up to date without visiting a shop. In the context of cyber-physical systems, where openness accompanies the need for software updates, application download to embedded systems could be an enabling technology. Furthermore, it could enable users to extend the system's functionality, allowing new business models; e.g., parking assistance or infotainment updates could be offered via an app (application) store. However, from a safety perspective, this trend also raises huge challenges since safety-critical software has to be protected. The state of the practice is to integrate applications and to argue about safety at development time. The current Automotive open system architecture (AUTOSAR) for instance allows only static configuration [1].

Deploying an untrusted application during runtime can interfere with safety-critical software. In addition, embedded systems for safety-critical tasks are often low-resource systems. These technical limitations (small amount of memory, processing power, etc.) are a further challenge for dynamic software installation. This paper presents an approach that allows dynamic deployment of applications to embedded systems, especially those with low resources, and automatic integration of non-safety-critical applications into a platform containing safety-critical software. To achieve this, freedom from interference between the non-safety-critical app and the safety-critical software running on the platform must be guaranteed. The presented solution uses three sets of protective measures to achieve this: a) measures against spatial memory corruptions, b) measures against temporal scheduling interferences and c) measures against corruptions via shared services.

## II. RELATED WORK

For decades, desktop computers – usually standard x86/AMD64 hardware – allowed dynamic installation of new software. Next on the consumer market were smartphones, e.g. with a Linux-based operating system, and mobile devices that allowed downloading software with restricted user rights in terms of system resources. Called applications or apps, these can be created using a software development kit, e.g. for Java, and can be installed from the desktop, from an app store, or from websites.

With regard to other embedded systems, Han et al. introduced the SOS operating system for sensor nodes [2]. The key idea of SOS is to provide support for dynamically loadable modules. SOS enables engineers to download new modules to the target system during runtime. Contiki is an operating system for sensor network nodes consisting of a small event-driven kernel. The approach of Dunkels et al. enables dynamic applications for resource-constrained wireless sensor nodes through dynamic linking and locating on the target node [3]. LyraOS is an operating system for embedded systems. It has been ported to several architectures (e.g., ARM, x86), supports its own file system (LyraFile), and provides a subset of the POSIX standard for applications. Shen and Chiang extended LyraOS to support dynamic software components by pre-linking on the server side [4]. In contrast, the solution presented here makes it possible to download untrusted applications dynamically to a low-resource embedded system. In addition, the system can contain safety-critical software.

## III. APPLICATION AND PLATFORM

In this section, we introduce the working environment and basic approach of the concept. The working environment includes the host system and the target system. The host system

contains applications that could be downloaded to an embedded system; e.g., the host system could be a standard desktop computer, additionally including a complete C development environment for embedded software for the purpose of developing applications. The platform is the target system that is selected to run this software, e.g. a microcontroller with an operating system. The connection between these two systems is established by a communication channel, e.g., via CAN (Controller Area Network).

The approach enables automated integration of new functionality. An interface description between the application and the underlying platform enables this automated integration. The interface is used to arbitrate between a downloaded application and the corresponding platform. It must be formalized in such a way that existing applications are not influenced inappropriately. To achieve this, we propose a vertical interface description according to [5]. The description follows a modular, contract-based approach for specifying demands and guarantees in order to form a so-called vertical interface between application and platform. VerSaI (Vertical Safety Interface) is our language approach for describing demands and guarantees [5].

### A. Application demands and platform guarantees

The demands define the safety-related behavior of the platform requested by the application. Note that the demands result from higher-level safety requirements of an application and have to be derived by the application developer. Consequently, a demand is linked to a specific application. An example of a safety demand is: “The application must be protected from temporal CPU interferences (ASIL C).”

The guarantees are related to a specific platform and define the actual safety-relevant capabilities of the platform. With regard to app downloads, a platform-guarantee satisfies an application demand depending on the service guarantees already assigned to other applications. An example of a safety guarantee is: “The platform is capable of protecting core0 from temporal interferences (ASIL C).”

When developing safety-critical systems, the requirements used to specify the safety concept of the existing applications must be considered as application demands that have to be covered by platform guarantees. Safety has to be re-evaluated whenever the system changes. An example of a requirement is a residual failure rate of a platform sub-system with respect to random hardware failures lower than or equal to 0.5 FIT (Failure In Time: number of failures that can be expected in $10^9$ hours of operation).

Zimmer [5] proposes the following safety-related demand-guarantee dependencies for the vertical interface: *platform service failures, health monitoring, service diversity*, and *resource protection*.

*Platform service failure* detection or avoidance deals with failures caused by the platform; for instance, a value failure of analog-to-digital converter signals larger than a specified threshold must be detected within a defined period of time.

The opposite of platform service failure detection is *health monitoring*. It focuses on trapping and encapsulating application and execution failures. As an example, the platform has to detect and arbitrate whether an application/task is scheduled too frequently.

*Service diversity*, also known as independency or dissimilarity, deals with systematic failures in redundant components. Service diversity methods reduce the likelihood of common-cause systematic failures via independence of communication links, input and output services; e.g., an application could request an analog input channel to calculate a physical value in different ways in order to avoid a systematic value failure.

*Resource protection* aims at achieving freedom from interferences. We define an interference as a cascading failure via a shared resource that potentially violates safety requirements. The interference propagates among several applications via a commonly used resource instead of a private resource for every application.

The presented work focus on resource protection, because protection regarding freedom from interference is a key issue. Service diversity for instance requires further investigation to be compatible with the notion of generic platform. Design patterns, e.g. with respect to failure detection and monitoring need to be elaborated. The potential solution space of patterns can be assumed as a platform resource that need to be mitigated between application demands and guaranties.

### Application software download

Once an application has been developed and its demands have been specified, the app can be downloaded to a platform with defined guarantees. A software component called app manager automatically evaluates whether the demands and guarantees for a specific platform are compatible. The app manager consists of two parts: on the host system, it builds the technical basis for registering applications and starting the download process. On the target system, the app manager receives and installs the application. In addition, the app manager on the target system needs to ensure at runtime that applications do not consume more platform resources than demanded by the application.

The compatibility arbitration between demands and guarantees can be located at the host as well as at the target. We propose keeping as little overhead as possible on the target in order to be efficient regarding low-resource embedded systems. Our aim is to keep as much information on the target system as needed to allow a target to connect with different host systems. Hence, we have to store information regarding the installed applications on the target.

The app download procedure is illustrated in figure 1. The first check is if it is possible to install further applications. The restriction results in the maximum number of applications that can be handled by the target system’s app manager, e.g., memory slots storing administrative application information such as a function pointer to the application’s main function or its execution time. Thereupon, the host system sends the resource demands, such as memory requirements and timing properties, to the target system. If the target has enough resources available, it will send the host system an

acknowledgment and the required information to generate the application executable. Such information could be information to resolve symbols for platform system services of a re-locatable object file mapping them to the correct target memory addresses. It depends on the embedded system used whether the addresses are physical or virtual addresses. The last activities of the host system are the generation of an application executable and its transmission together with the necessary information, e.g., regarding memory segments or the location of the application's main and init functions.

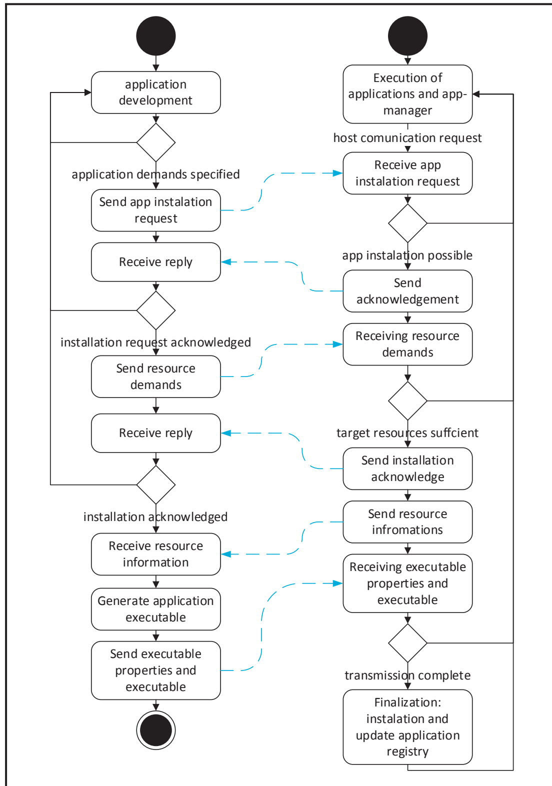


Fig. 1. *Applicastion download:, the host system is shown in the left part of the activity diagram and the target system on the right part.*

## IV. SHARED RESOURCE PROTECTION

Our app download concept scales to all four of the vertical safety requirement categories presented in the previous section. However, our work focuses on resource protection as resource protection is the technical basis from both the app download perspective and the mixed-criticality perspective. In our concept, we consider platform service failure detection, health monitoring, and platform components for supporting service diversity as special platform services and consequently as a shared resource.

The safety demands and guarantees in relation to shared resources focus on freedom from interferences, which is also called segregation or partitioning. In [6], we presented an analysis method for detecting interference channels and, based on our work related to analysis methods, we identified three categories of shared resources for an app download.

### A. *Memory protection*

Spatial memory interference, i.e., memory corruption, occurs if an (application) task erroneously writes to a memory region belonging to another task. Potential memory regions belonging to a task include the task's stack, its data and code, as well as registers for controlling peripherals that are exclusively assigned to the task of the application, respectively.

The effect of arbitrary memory interference on the behavior of the affected program is not always predictable, which is the reason why memory interferences are typically prevented before the memory is modified. The corresponding mechanisms for protecting the system against memory corruptions usually demand close interaction between the operating system and dedicated memory protection hardware. There are two ways to support memory protection in hardware: the memory protection unit (MPU) and the memory management unit (MMU). The latter does not only provide memory protection but also the very flexible tool of memory virtualization. This facilitates the creation of relocatable apps and simplifies app download. On the other side, MMUs have several disadvantages since they make timing predictability more difficult, require more complex handling by the OS, and increase hardware costs. Although MMU in combination with an extended OS like Linux simplifies app download, the described implementation uses an MPU, which is common in platforms for safety-critical embedded systems.

Using an MPU or an MMU to separate different programs' memories has one other major consequence. If an application task needs to call a module using memory that the current task cannot access, we need a system call feature to make the call. This is, for example, necessary if an untrusted task wants to call a shared service such as the OS and corruptions of the shared services are to be avoided. In the app download scenario, both concepts are required. The MPU is required to protect the legacy platform from the app, especially its safety-critical parts, such as the OS. As a consequence, the applications require a system call interface in order to interact with the system as well.

### B. *Execution time protection*

In real-time systems, the computing time of a process is as important as its computational results. In this context, a deadline violation means a timing fault occurring when a task or interrupt service routine takes more time to complete than allowed, delivering the result at a later time. To ensure that the amount of processing resource of a platform is enough, a sufficient and necessary response time test analysis, made according to the scheduling algorithm implemented, must be done. Besides the response analysis, a run time monitor must also ensure that the task does not take more computational time during run time, than it was considered by the response time analysis. Our platform implements the rate monotonic scheduler algorithm, with periodic fixed priority tasks. To develop a reliable system, the system designer specifies certain timing requirements, such as the worst-case execution time and period for each task. Based upon this information, the platform analyzes whether all tasks are able to meet their deadlines [7]. To allow dynamic app download, the response time analysis is shifted to the app manager as part of the check on whether the

target resources are sufficient. In addition, a concept for guaranteeing that an application task cannot extend its execution time is needed. Possibilities include monitoring approaches and preemptive scheduling algorithms. We will present our implementation in more detail in section V.

### *C. Service Protection*

We define a service as software when it abstracts from the platform and is provided to applications as part of the infrastructure. An example is a driver for the peripheral ADC (analog to digital converter). The ADC driver can be used by several applications to access the shared ADC. A protection service provides mechanisms for segregating the service in a way that guarantees its functionality. This service functionality must also be ensured for the applications to detect a faulty application that could misuse the service. Such a service misuse should not lead the service to fail or make it unavailable. The state of the art is that the system architect analyzes the system's services w.r.t. interferences. Regarding the app download, we aim to achieve service protection via plausibility checks of the service call parameters. How to develop a service that is free of interferences and is intended for app download is our ongoing research. In general, services have to be protected against spatial, temporal, and behavioral interferences [6]. We use memory protection to ensure spatial service protection. An application only has access to a reduced part of the memory map. The service itself is typically located in a protected memory region. If an app needs to access a service via the corresponding service function, there has to be an interaction between the application's and the service's memory regions. We use the concept of the so-called Service Call (SVC), which is to use an interrupt called from the non-privileged application code to access the protected memory region. The interrupt is a software interrupt designed for a user (application) piece of code to invoke the operating system and request the execution of a routine in privileged mode that has access to the protected memory region.

## V. IMPLEMENTATION ASPECTS

The prototypical implementation uses the STM32F103RF microcontroller. The microcontroller has a 72MHz Cortex-M3 CPU with MPU and 768 Kbytes of Flash, 96 Kbytes of SRAM, and various peripherals. Our prototype software comprises the FreeRTOS open source real-time operating system and the STM32 peripheral library. The host system is realized in eclipse with the Eclipse and OpenOCD for developing and testing the embedded software. In addition, the toolchain contains the GNU (cross) Compiler Collection, the GNU Linker, and the GNU Binutils.

### *A. Execution time protection*

To install a new application, this application must have sufficient time to be executed. Conclusions before runtime are calculated by the Response Time Analysis if all tasks assigned to the system meet their deadlines. The scheduling algorithm implements the rate monotonic scheduler. Tasks with a lower period have higher priority. The algorithm runs at a base time (quantum) of 1 ms (the scheduler interrupt). At each interrupt, the scheduler checks which is the highest-priority task to run. If more than one task has the same highest priority, and they are ready to run, they will share the processing time in a round robin fashion.

Our implementation extends the FreeRTOS kernel and enables it to calculate the schedulability described in the previous section. Two new parameters were added to the task structure: the period and the worst-case execution time. Now the task extends the former functionality and the kernel is able to calculate the response time for all tasks in the system based on the rate monotonic scheduler algorithm. For each new app to be installed, a new response time analysis must be performed to check if the new set, including the new app task to be installed, is still schedulable (i.e., checking whether all tasks from the set meet their deadlines).

If a task cannot be executed within its deadline (execution time exceeds period), the function returns with a false answer, meaning that the task set is not schedulable. Only if the new task set is schedulable is the new task included on the permanent task system list and the FreeRTOS scheduler can run again, now with the new task included. The execution time monitor is realized through a down-counter timer interrupt that is set when its timer reaches the value zero, and a variable that contains the value that must be loaded into this down-counter. If the current executing task exceeds its time budget, it is preempted and another task is scheduled. This avoids temporal interferences.

## VI. CONCLUSION

In this paper, we presented a concept for downloading applications to an embedded system containing safety-critical software. The concept reacts to low-resource systems typical for safety-critical systems in the automotive domain. Furthermore, the demand and guarantee relation of safety interfaces (between applications and a platforms) is regarded to enable downloading safety-critical applications. As future work, we plan to further develop the idea of runtime arbitration between application demands and platform guarantees and fairly automated platform service interference analysis in combination with a service protection assignment.

## ACKNOWLEDGMENT

This research was funding from the DNT – 'DIGITAL ENGINEERING FOR COMMERCIAL VEHICLES' project.

## REFERENCES

[1] Wagner, M., et. al., "Towards runtime adaptation in AUTOSAR: Adding Service-orientation to automotive software architecture," ETFA 2014

[2] Chih-Chieh Han et al. "A Dynamic Operating System for Sensor Nodes", MobiSys 2005

[3] Adam Dunkels et al. "Run-time Dynamic Linking for Reprogramming Wireless Sensor Networks", SenSys 2006

[4] Bor-Yeh Shen and Mei-Ling Chiang. "A Server-Side Pre-linking Mechanism for Updating Embedded Clients Dynamically, 2007

[5] B Zimmer, "Efficiently Deploying Safety-Critical Applications onto Open Integrated Architectures",Fraunhofer IESE 2014

[6] Bastian Zimmer et. Al. "A Systematic Approach for Software Interference Analysis", ISSRE 2014

[7] Hansson, Hans, Jan Carlson, and Damir Isovic, "Real-Time Systems", Fraunhofer IESE 2010