



## The world's smallest OSGi solution

**ProSyst mBS, the world's smallest OSGi solution fully compliant to OSGi revision 4.2, has been ported successfully to an embedded low power hardware with an ARM9 CPU at 156 MHz and 8 MB RAM and flash memory. While leaving sufficient resources for sophisticated applications and services, the OSGi framework meets all non-functional requirements of a mass-market automotive telematics system. It therefore also qualifies for optimal use in many other embedded markets, such as the home automation space.**

The results described in this paper were achieved in a real world automotive project. Innovation cycles of next generation Car2X services (e.g. eCall, Remote Control, Car2Car messaging, diagnostics, etc.) are shorter than product lifecycles, fostering the challenge rapid development and in-life deployment of new services. OSGi was introduced as a manageable service runtime environment capable of hosting multiple services concurrently while decoupling the software layer from the underlying hardware.

This technical white paper demonstrates that, if done right, OSGi can be applied on much smaller systems than commonly assumed. It provides a detailed solution overview, outlines the most important non-functional measurements like startup time and memory consumption, discusses how these achievements impact the scope of OSGi applications and closes with an outlook about where embedded OSGi is heading to.

### Platform Details

The target hardware is a micro processor platform specifically designed for low cost and low power consumption. It consists of an ARM9 based baseband processor running at 156 MHz, hosting a proprietary RTOS, the GSM stack as well as the application runtime environment based on a Java VM. The board is equipped with 16 MB of SRAM and 32 MB of flash memory, from which only 8 MB RAM and flash are available for applications. Energy efficiency is a key design goal. The CPU and peripheral chipsets consume a total of 450 mW

in active state, 97 mW in idle state and 25 mW in sleep mode.

The Java execution environment on this target is surprisingly powerful, consisting of the IBM J9 virtual machine with Java 1.4 compliant CDC/FP configuration plus a number of JSRs. However, it is configured to only support one Java process, i.e. one application at a time as the system cannot carry the overhead of multiple VMs. OSGi was chosen to turn this single-process environment into a multi-application platform. The OSGi framework provides a sophisticated and

#### OSGi in a nutshell

OSGi is the global standard for creating and assembling modular software built with Java technology. It provides a modular services framework and a component based architecture. OSGi has its roots in embedded home gateways but matured to a technology that is being adopted in various markets such as automotive, telematics, mobile, eHealth/assisted living and last but not least, Java enterprise servers.

Software components (i.e. middleware, apps, services, etc.) can be plugged into the OSGi framework at any point in time and each component has its own lifecycle. Thus OSGi enables concurrent multi-services and multi-application execution in just one VM. Moreover, all components and APIs are fully manageable and accessible from remote, which opens the door for true application lifecycle management, remote diagnostics and configuration scenarios.

By means of a sophisticated meta data model as well as a mature set of APIs, components can share functionality among each other. Complex application no longer need to be monolithic but can be modularized which results in a significant advancement for code reusability, development efficiency, feature extensibility and clean functional abstractions. In simplified terms, OSGi can be considered a mini application server comprising a rich Service Oriented Architecture (SOA).

In addition to these generic OSGi benefits, the standard defines a host of value adding APIs and services like UPnP, Logging, Configuration, Events, Remote Management, Security, etc.



standardized infrastructure to host and execute multiple Java components in one VM process simultaneously, thus adding multi-app and multi-service support while reducing the total cost per service.

The OSGi stack on the system is a derivative of ProSyst mBS 7.0, tailored to the needs of the project. Modifications compared to the standard product configuration include various optimizations throughout the stack as well as the exclusion of OSGi specification features not required for this particular case. The result is a highly optimized, yet complete OSGi stack compliant to the OSGi revision 4.2. The text box “Features Supported” lists all supported OSGi and additional ProSyst features.

### Resource Management

In embedded environments with limited memory, energy and computing resources, the system robustness and uptime directly depend on the application resource consumption strategy. If the platform supports concurrent execution of multiple resource-competing applications, which is one of the propositions OSGi offers, reaching high stability is particularly hard.

By introducing resource management to its OSGi implementation, ProSyst pushes platform stability to unreached levels and once again assumes leadership in embedded OSGi innovation. The ProSyst resource manager is capable of monitoring per application runtime resource utilization metrics such as

- number of active threads
- number of open sockets
- allocation of data storage space
- allocation of memory
- CPU usage

Based on configurable policies the resource manager intervenes by stopping, disabling, uninstalling or blacklisting an application that exceeds its pre-configured resource utilization limits. To cater for more sophisticated recovering strategies, custom activities can be developed and plugged into the resource manager, i.e. to generate a user prompt or a message to a customer care center, to reboot the entire system, etc. OSGi resource management grants predictable platform behavior in cases the system runs out of whatever resource, thus contributing significantly to robustness and uptime parameters of the system.

Automotive use cases foster a series of challenging non-functional requirements. Startup times of electronic control units (ECU) and their contained applications must be as fast as possible to offer safety and a responsive experience to users. For obvious reasons, system robustness and stability are to meet highest standards. Moreover, RAM memory and CPU cycles being scarce resources call for optimized resource consumption designs. How does OSGi fit into this environment?

### Measurements

In the context of these resource constraints, OSGi is to be considered a piece of infrastructure adding computational “overhead”, as it isn’t an application itself. The faster it boots, the less memory it occupies and the fewer CPU cycles it burns, the more resources are left for the actual applications and services. The key measurements of the OSGi stack used in this project are:

#### **Startup time: 4,3 s**

With just 4.3 seconds OSGi boots amazingly fast, considering the amount and complexity of infrastructure services it adds to the Java process. The startup time depicts the elapsed time from entering the framework main method until the framework started event is fired. The unique ProSyst lazy loading mechanism is activated to postpone some of the class loading and initialization times of OSGi functionality until first used by applications.

#### **Allocated RAM memory: 1,13 MB**

One of the design goals of the mBS OSGi implementation is to minimize the creation of temporary objects and heavy data structures. This reduces the work load of the garbage collector and thus boosts startup and available memory for applications. In this particular case, approximately 5 MB of RAM can be allocated by applications.

#### **Flash memory occupation: 765 kB**

The OSGi stack, including all class and configuration files and scripts, takes up 765 kB of memory on the local 32 MB flash drive, leaving plenty of room for applications.



### **No Runtime performance penalty**

Once fully loaded, the OSGi framework goes into idle mode and does not burn CPU cycles in a significant order. Nonetheless, to investigate the OSGi stack behavior from an application perspective, a test application was developed and executed on stack configurations with varying levels of OSGi feature support. The results reveal that there is no direct relation between application execution time and the number of supported OSGi features, in other words, the OSGi runtime has no significant impact on the application performance.

### **Robustness**

Using Java for the application layer greatly improves robustness due to language features like exception handling, pointer management or garbage collection. To cater for robustness in the multi-service context of OSGi, the Java runtime has been extended by a powerful in-VM resource management logic. A resource manager running in OSGi monitors application bundles and takes pre-defined actions in case that applications exceed their resource consumption limits, thus enhancing robustness and predictability of the system. Further details can be found in the text box "Resource Management".

### **Application Scenarios**

The unique combination of low cost / power hardware and OSGi opens the door for an unlimited number of new mass market application use cases. The benefits of a standardized, remotely manageable, sophisticated multi-service application runtime environment like ProSyst mBS are highly attractive for scalable systems with distributed, intelligent and even heterogeneous devices.

Traditional markets like smart home, automotive, telematics, industrial applications, building automation, mobile or M2M benefit as much as emerging markets such as smart metering, smart grid, eHealth or, more generally spoken, the Internet of Things (IOT). In terms of target devices there are no limitations for device categories that could benefit from this high performance

yet super-light OSGi solution: Router, IADs, gateways, smart phones, femto cells, M2M modules, ECUs, telematic boxes, PLCs, MUC controller (multi utility communication), USB sticks, etc.

#### **Features supported in this particular project**

##### **OSGi Features:**

Module Layer  
Life Cycle Layer  
Service Layer  
Framework API  
Package Admin Service  
Start Level Service  
URL Handlers Service  
Log Service  
Config Admin Service  
Preference Service  
IO Connector Service  
Declarative Services  
Event Admin  
Monitor Admin Service  
Tracker  
XML Parser Service  
Position  
Measurement and State

##### **ProSyst Features:**

Lazy Bundle Loading  
Resource Management  
Event Thread Monitor  
Context Class Loader

ProSyst Software GmbH  
Dürener Straße 405  
50858 Cologne  
Germany  
info@prosyst.com  
Tel +49 221 6604 200