# LEO/P4 - A µ-Kernel Based OSEK Implementation

Robert Kaiser
Sysgo Real-Time Solutions GmbH, Klein-Winternheim. Germany
rkaiser@sysgo.de
January 18, 2000

## 1. Abstract

This article introduces LEO/P4, an implementation of the standardized OSEK OS API as a server on top of the new µ-kernel P4. Together with the OSEK emulation environments LEO/Lynx or LEO/Linux, it provides an OSEK run-time environment that can seamlessly support OSEK applications from the early stages of development unto the final production version in a vehicle.

The P4 µ-kernel is a new, advanced re-implementation of the L4 µ-kernel as specified by Liedtke /4/. It features a processor independent API and is internally structured for better portability. Platform specific support routines are provided to P4 by an external module (PSP), so porting P4 to another platform within the same processor family can be achieved without modifications to the µ-kernel's binary code.

In contrast to traditional monolithic implementations of OSEK, the µ-kernel-based approach allows for multiple operating system APIs and instanciations to coexist. Specifically, it is possible to have multiple instances of OSEK OS running independently in a single machine, each one in it's own protected address space.

Moreover, a Linux server can also be added to run in parallel with the OSEK OS instance(s), thus turning the system into a full-featured UNIX workstation. This does not affect OSEK's real-time characteristics, so the resulting system is a very comfortable and efficient environment for developing, debugging and testing OSEK-based code under real-world conditions.

All interaction between the servers is based on the µ-kernel's inter process communication (IPC) mechanism, which features the capability of transparent re-routing of messages across a network. Therefore, servers can be migrated from the development system to the embedded controller without any modification to their code. This not only provides an easy way to implement distributed systems, it also makes the process of moving OSEK applications from the development system to their designated target ECU far less painful than usual.

We will start with brief introductions of the OSEK OS standard and of LEO, SYSGO's OSEK OS implementation and will then focus on the benefits of the µ-kernel-based implementation, as well as on the P4 µ-kernel itself.

## 2. What is OSEK ?

*OSEK/VDX* is a standardization effort mainly driven by the European automotive industry to provide a common programming environment for automotive control software.

OSEK stands for "Offene Systeme für Elektronik im Kraftfahrzeug" (Open Systems for Electronics in Vehicles), VDX stands for "Vehicle Distributed Executive".

The OSEK committee provides publicly available documents defining the OSEK application program interfaces and corresponding system behavior. Software manufacturers are invited to implement systems conforming to these definitions.

The OSEK/VDX standard specifies APIs for:

- a real-time operating system (OSEK OS)
- communication (OSEK COM)
- network management (OSEK NM).

In this article, we will focus on the real-time operating system API, OSEK OS.

### 2.1 Distinguishing Features of OSEK OS

The OSEK OS standard specifies a real-time, multi-tasking operating system. It covers functionality for task management, task synchronization, interrupt management, alarm handling and error treatment. OSEK OS differs from traditional real-time OSes in several ways:

#### 2.1.1 Static Design

OSEK does not support dynamic creation/destruction of system objects (i.e. tasks, events, alarms) at run-time. The number of tasks and their associated resources has to be defined at system generation time.

The rationale behind this is that the temporal behavior of dynamic allocation and de-allocation tends to depend largely on factors which are hard (sometimes impossible) to predict (e.g. memory fragmentation, etc.). In a real-time environment, the system does not have the freedom to simply suspend a task requesting resources until these resources have become available. Consequently, existing real-time applications typically pre-allocate all resources at start-up time in order to avoid unpredictable delays later on. It makes sense to enforce this policy by making it part of the system definition.

OSEK also does not specify any functions to change a task’s priority at run-time. This allows precomputation (at system configuration time) of parameters needed for the "priority ceiling" policy which is used by OSEK to avoid priority inversions.

#### 2.1.2 System Generator Tool

Any OSEK application consists of the actual application code and an application definition. For the latter, a special "OSEK Implementation Language" (OIL) has been developed by the OSEK standardization committee. The application definition defines all system resources which are required by the application.

At system generation time, a system generator  tool which is part of the OSEK implementation reads the application definition. This tool then produces " C" code containing all prerequisite data structures and initialization code required by the application. This "C" code, combined with the actual application code and the OSEK kernel is then used to generate the executable binary image.

### 2.1.3 Deliberate Omissions

Some functional areas which are traditionally covered by operating systems have been deliberately left out of the OSEK specification.

Specifically, OSEK does not say anything about I/O functionality. I/O is expected to be performed by modules outside of OSEK’s scope.

OSEK does however specify an interface for interrupt handling. This facilitates the writing of device drivers as part of the OSEK application code.

Also, in accordance with the static design principle mentioned above, no memory management functions (e.g. dynamic allocation/de-allocation of buffers) are defined by OSEK.

### 2.1.4 Conformance Classes

Target machines addressed by OSEK range from small eight-bit controllers up to high-end 32-bit or 64-bit processors with multi-megabyte memories. In order to support such a broad range, OSEK specifies a set of four "conformance classes". Each conformance class limits the amount of available system resources such as task priorities, scheduling capabilities and system services.

### 2.1.5 Error Checking Levels

The system can be configured for two different levels of error checking: the "standard" level is designed for minimum error checking overhead. It is intended for use in a production version of the software. The "extended" level, on the other hand, is intended for debug and test. It performs extensive checks and thus requires more hardware resources (both memory and computational power).

## 3. LEO

### 3.1 Emulated OSEK

Based on the OSEK specification SYSGO created a development environment for rapid system prototyping running under UNIX: LEO runs an OSEK application and the OSEK kernel as a single user-level process.

Today, LEO is available for several host operating systems, including Linux, FreeBSD, OpenBSD and LynxOS, where the LynxOS version, due to the real-time capabilities of the host operating system, can provide guaranteed inter-task communication and switching times < 15 µs. It allows not only the emulation, but the real-life simulation of a complete ECU in the running car.

From the host operating system's point of view, the LEO process is just another task running at user level confined to it's own addressing space. Therefore, the host operating system can also serve other requests while running the OSEK system. It can even run multiple OSEK instances independently.

LEO maps the OSEK task functionality to thread primitives provided by the host operating system. Typically, there is one thread per OSEK interrupt, one thread running the OSEK kernel code and one thread serving as a "container" for the OSEK application code. The OSEK kernel uses it's own scheduler to switch between OSEK application task contexts, thus the host OS's scheduler does not get involve here.

### 3.2 System Generator Tool

The system generator tool that comes with LEO is called CLEO. To the customer, its usage is the same regardless of the host operating system in use. Besides generating code, CLEO also performs consistency checks on the application definition. It automatically assigns event and interrupt masks and generates symbolic labels for use in the application.

### 3.3 Conformance Classes

LEO supports all available conformance classes. The system generator CLEO insures that the required conformance class matches the requested set of system resources. CLEO can also automatically determine the minimum required conformance class for a given application definition.

### 3.4 Kernel Monitor

As an extension to the OSEK standard, LEO also features an optional kernel monitor: LEMON can be used interactively to list any OSEK objects and display their detailed state in a running OSEK system. It is also able to halt ("freeze") and resume ("unfreeze") the OSEK system interactively. Enabling/disabling support for LEMON is conveniently achieved by setting a (non-standard) attribute in the system definition File.

## 4. The µ-kernel Approach

### 4.1 Motivation: Moving to the ECU

Having successfully developed OSEK-based applications with the LEO emulation environment, SYSGO's customers required a new environment to support their algorithms on embedded controller units (ECU). The straightforward approach to fulfil this requirement would have been to re-implement LEO as a stand-alone operating system for the ECU. However, in order to provide portability and being open for future enhancements SYSGO chose to develop a new, highly efficient and portable µ-kernel that would allow to run LEO as a server in user space.

### 4.2 Introducing the P4 µ-Kernel

A *µ-kernel* is an operating system with only the essential services. Based on these services, the functionality of a traditional monolithic operating system can be implemented as a user space process.
The basic idea is to tolerate concepts in the kernel only if moving them outside the kernel would make them impossible to implement (e.g. privileged machine instructions required), would cause significant overhead (e.g. excessive context switching) or would compromise the system's security. The traditional job of a monolithic operating system (i.e. Memory management, I/O, Task scheduling, etc.) is performed by user programs (so-called servers).

P4 is a new µ-kernel developed by SYSGO. Like a traditional operating system, P4 is invoked by a system call or trap mechanism. The small number of system calls handled by P4 all deal with the following basic concepts:

- Address spaces

  Address spaces are mappings which associate virtual pages to physical pages.

- Tasks and Threads

  Tasks and threads are activities executing inside an address space. Each task has it's own address space. Each task consists of one or more threads, where all threads share the same address space.

- Inter-process Communication (IPC)

  IPC is the basic mechanism for tasks and threads to communicate with each other. Specifically, the system call mechanism of a traditional monolithic system is replaced by IPC in P4: in order to direct a server to deliver a certain service, an application program sends an appropriate IPC message to that server and waits for it's response message. The P4 kernel just acts as a transport agent here, it does not really care about the contents of the messages being transferred. IPC is also used to transfer mappings (access rights to pages) from one task's address space to another.

### 4.3 Benefits of the µ-kernel Approach

The µ-kernel approach has several advantages over traditional monolithic operating systems:

- The system is more flexible and tailorable. Different strategies and APIs, implemented by different servers, can coexist in the system. Commonly required services such as device drivers, file systems or network stacks can be implemented as servers too and can consequently be shared by the different APIs.

Based on P4's primitives, it is possible to implement an OSEK server as a task that works in much the same way as the LEO/Lynx or LEO/Linux emulation environment does. Moreover, these primitives are also sufficient to implement a workstation-type operating system (such as Linux) as a server task. It is even possible to run multiple server tasks (i.e. multiple instances of OSEK and/or Linux) independently at the same time on a single machine.

- The amount of code running in kernel mode (i.e. code which, when faulty, will cause a catastrophic system failure) is drastically reduced: A server malfunction is as isolated as any other user program's malfunction. Furthermore, server code can be tested and debugged like any other user program (i.e. in a live system and with all the luxury of a window-oriented HLL debugger).

For example, in a system running multiple OSEK instances, a fault in one OSEK application can not affect the other OSEK instances. Even if one OSEK instance crashes, all the other instances remain fully functional.

- The µ-kernel hides architecture specific details of the system and offers a common, processor independent API to the servers.

Porting the system to a new architecture means to port the (small) µ-kernel. Servers only need to be re-compiled.

- IPC messages exchanged between servers can be transparently re-routed across a network. Thus, since all interaction between servers is based on the IPC mechanism, a distributed system can be constructed fairly easily.

Apart from opening a path to distributed systems in general, this feature can also come in handy when migrating code from one platform to another: If the code is implemented as a number of separate server modules, these modules can be migrated one by one, while the complete system remains functional all the time.
For example, it is possible to run an application on a prototype ECU, where a device driver on another machine substitutes for a piece of hardware that the ECU does not have yet. The IPC message redirection can be made fully transparent for the interacting servers, so application need not be aware at all that it is using a substitute driver.

### 4.4 Disadvantages of the µ-kernel Approach

The µ-kernel approach also has a downside which we shall not conceal here:

- Context switching overhead: in a µ-kernel system, context switches, both between kernel and user space and between different user spaces occur far more often than they do in a monolithic operating system.

This is the main reason why operating systems built on top of Mach (probably the most prominent µ-kernel architecture) perform far worse than comparable monolithic implementations.
However, the L4 µ-kernel developed by Jochen Liedtke at GMD has shown that by applying appropriate design, context switching overhead can be reduced by an order of magnitude /1/,/2/. An implementation of the popular monolithic Linux as a server on top of L4 shows that the overhead is almost negligible: The Linux/L4 server's performance is reported to be about 6%-7% slower than a monolithic Linux running on the same hardware /3/.

### 4.5 P4, an advanced L4

The P4 µ-kernel has been designed with the same principles as L4, in fact, it implements an API almost identical to L4's. Therefore, servers originally developed for L4 are easy to port to P4. There are a few differences though:

- Unlike L4, P4 is intended to be portable, while maintaining L4's high speed and low overhead characteristics. It's source code is mostly written in "C" and the platform independent code portions are kept separate from the platform/processor dependent portions with clearly defined interfaces (PSP/ASP, see below) in between.

- P4's memory footprint is approximately 4-5 times larger than L4's, due to the use of a high-level language. Nevertheless, P4 still qualifies as a "micro"-kernel, given that it's code size is usually in the 30KB to 60KB range.

- Like L4, P4 provides full virtual memory support, but unlike L4, it can also run without an MMU if so desired.

- P4's API includes a number of extensions that are not part of the original L4 API definition. These extensions were made to address issues of API portability, practical usability and the option to run without MMU.

- Unlike L4, P4 is fully preemptive, i.e. real-time capable

### 4.6 Layers inside P4

#### 4.6.1 Generic Layer

About two thirds of P4's source code is independent of both platform and machine architecture. This code implements IPC, task & thread scheduling and address space management. This layer makes calls into the PSP and ASP layers (see below) to perform processor or platform specific actions.

#### 4.6.2 Architecture Support Package (ASP)

The ASP provides the machine architecture (i.e. processor family) specific functionality required by the generic layer. It consists of a number of functions implemented in "C", assembler or as preprocessor macros, whichever is most appropriate. The ASP's interface is clearly specified and it is possible to write an ASP for virtually any given processor type according to the specification.

Unlike the PSP (see below), the ASP is compiled/linked into the µ-kernel binary. The separation between the ASP and the generic layer exists in the source code, but not in the binary code.

### 4.6.3 Platform Support Package (PSP)

The PSP is a set of routines providing all the functions that an operating system kernel requires and that can not be implemented in a platform independent way. Functional areas typically covered by the PSP are:

- Bootstrap and initial system hardware setup
- System shutdown and reboot
- Determining the system's memory map
- The system clock ('ticker')
- Interrupt dispatch
- Interrupt prioritization/nesting
- Cache handling
- Optional polled serial channels (for diagnostics and debug)

P4 does not deal with I/O and consequently, the PSP does not contain any device drivers. Device drivers are to be implemented as user space servers on top of the µ-kernel. They can (and should) be implemented in a platform independent way, so they do not belong to the platform specific part of the µ-kernel.

The PSP is the first code to be invoked after a system reset. Thus, it will initially be running without any address translation. Later, when the operating system has been started, virtual addressing may be used.

The PSP's functions are accessed via a standardized jumptable. Therefore, the PSP code does not have any external references. It can be built as a separate binary. Thus, it is possible to run (and test) a PSP stand-alone, i.e. without interacting with other P4 components. A PSP test suite exists that allows to develop and test a new PSP without any µ-kernel involvement. Once the new PSP runs the test suite successfully, one can be very confident that it will also run flawlessly under the P4 µ-kernel.

## 5. Linux on P4

Using the µ-kernel technology opens the door for a seamless software development cycle from the first prototype to the ECUs in production. The P4 µ-kernel is fully preemptive and contains a real-time interrupt handler and scheduler. It is the only part which has direct access to the CPU. Besides the OSEK Server process, it can also host other operating systems at the same time. For example a Linux server can coexist with the OSEK kernel on the same machine. Code generated under Linux can then run -possibly on the same machine- under full control of the OSEK kernel in a true real-time environment. At the same time the OSEK server still has access to the UNIX services on the Linux side, thus the Linux server can perform non-real-time tasks in parallel such as GUI, visualization, web access, etc.
This is a big step ahead from the simulation/emulation inside an operating system (like LEO/Lynx) and is extremely helpful to validate the real-time properties of the OSEK application on a real OSEK kernel.

Literature:


/1/ J.Liedtke: *On µ-Kernel Construction.* In 15th ACM Symposium on Operating System Principles (SOSP), pages 237-250, Copper Mountain Resort, Colorado, December 1995

/2/ J.Liedtke, K.Elphinstone, S.Schönberg, H.Härtig, G.Heiser, N.Islam and T.Jaeger: *Achieved IPC Performance (still the foundation of extensibility)* . In 6th Workshop on Hot Topics in Operating Systems (HotOS), pages 28-31, Chatham (Cape Cod), Massachusetts, May 1997

/3/ H.Härtig, M.Hohmuth, J.Liedtke, S.Schönberg and J.Wolter: *The Performance of µ-Kernel-Based Systems*. In 16th ACM Symposium on Operating System Principles (SOSP), pages 66-77, Saint-Malo, France, October 1997

/4/ J. Liedtke: *L4 Reference Manual 486, Pentium, Pentium Pro.* Research Report 20549, IBM T. J. Watson Research Center, Yorktown Heights, NY, Sep 1996