Model Based Development of Embedded Control Software

Part 4: Supported Target Platforms

Claudiu Farcas

Credits: MoDECS Project Team, Giotto
Department of Computer Science
cs.uni-salzburg.at

UNIVERSITÄT SALZBURG

Current execution platforms for TDL

- Java
- OSEK/VDX
- InTime
- POSIX
- Native* (microkernel*)

Java - Advantages

- General purpose OOP language
- Rapid prototyping
- Cross platform development – virtual machines
- Same development platform from GUI to embedded device
- Automatic memory management - No pointers ☺
- Reasonable performance on modern hardware
- Easy integration with data storage solution
- …

Java - Disadvantages

- Resource management (garbage collector)
- Not hard real-time
- Requires high run-time resources (memory, CPU) for embedded devices
- Performance adequate for simulation level/proof of concept
- Hard to interact with hardware for controlling applications

JAVA - TDL Runtime - Details

- Dynamic module loading = Java Reflection
- TDL Tasks = Java Threads
- Timing -> Thread.sleep
- Scheduling multiple modules -> CPU partitioning
- Dispatch table of each mode computed at load-time
- Preemption -> thread.suspend, thread.resume

Java - TDL Runtime – Usage

Scenario: application consisting of module: MyModule

- Implementing functionality code:
  - MyModule.java contains the MyModule class
- Writing & compilation of the TDL application:
  - TDL timing code compiles into MyModule.ecode
  - Generated wrapper code for functionality: MyModule$.java
- Java sources compilation: MyModule, MyModule$.class
- Execution of MyModule under Java E-Machine

OSEK/VDX – What is it?

- German: “Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug” (Open systems and the corresponding interfaces for automotive electronics)

+

French “Vehicle Distributed Executive”

- Standard in Automotive Domain - 1993
- Registered trademark of Siemens AG

UNIVERSITÄT SALZBURG

OSEK/VDX – Why?

- Portability and reusability of software:
  - Interfaces - abstract and application-independent as possible
  - User interface independent of hardware/network
  - Verification of functionality and implementation of prototypes
- Standardization
- Scalability
  - Efficient design of architecture - Configurable and scalable functionalities to enable optimal adjustment of the architecture to the target application

UNIVERSITÄT SALZBURG

OSEK/VDX – Covered areas

- Communication
  - Data exchange within and between ECUs
- Network Management
  - Configuration determination and monitoring
- Operating System
  - Real-time environment for ECU software
  - Time Triggered Environment* (OSEKtime)
- System Configuration Overview - OIL

OSEK/VDX – Development

Figure credits: © OSEK

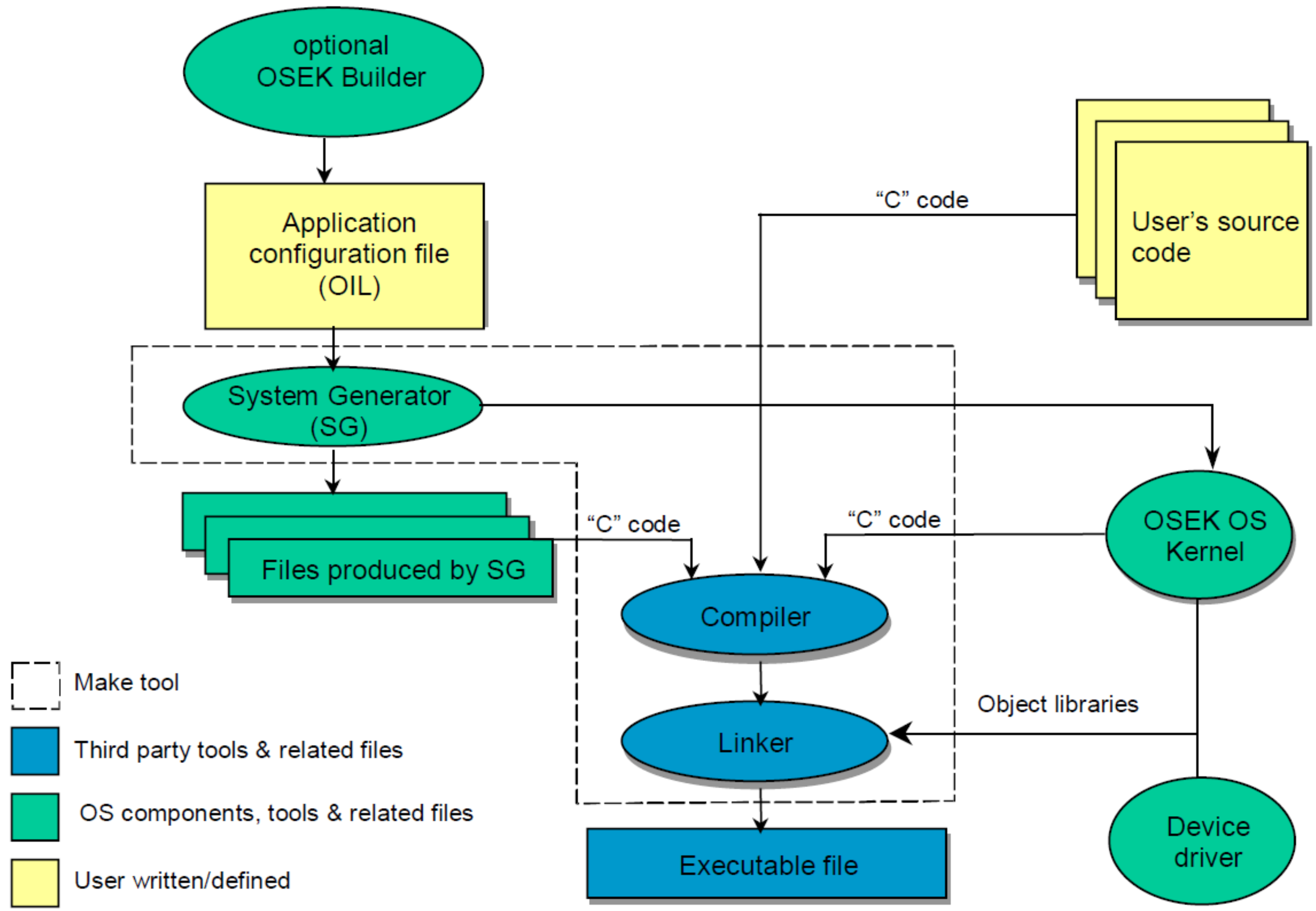


Figure 1-1: Example of development process for OSEK/VDX applications

OSEK/VDX - Advantages

- Savings in costs and development time
- Software quality – best product/company wins
- Standardized interfacing features for control units with different architectural designs.
- Reuse of resources (functionality code) – C language
- Absolute independence with regards to individual implementation (it's just a specification)

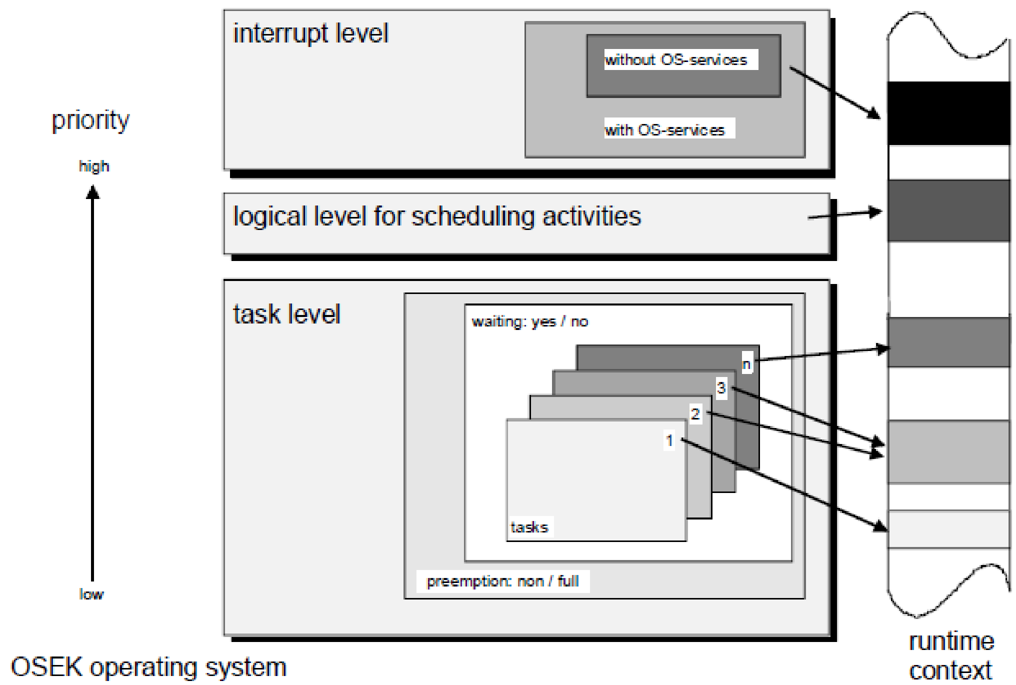
OSEK/VDX – The big NO(s)

- No malloc() , free()
- No heaps
- No memory regions / partitions
- No dynamic memory allocation facilities whatsoever
- No specific device I/O support
- No file system support
- No on-the-fly
  - creation of operating system objects (tasks, events, resources, alarms,...)
  - deletion of operating system objects
  - programmed changes of task priority or preemptibility
- No round-robin time-slicing of tasks
- No application mode switch at run-time

UNIVERSITÄT SALZBURG

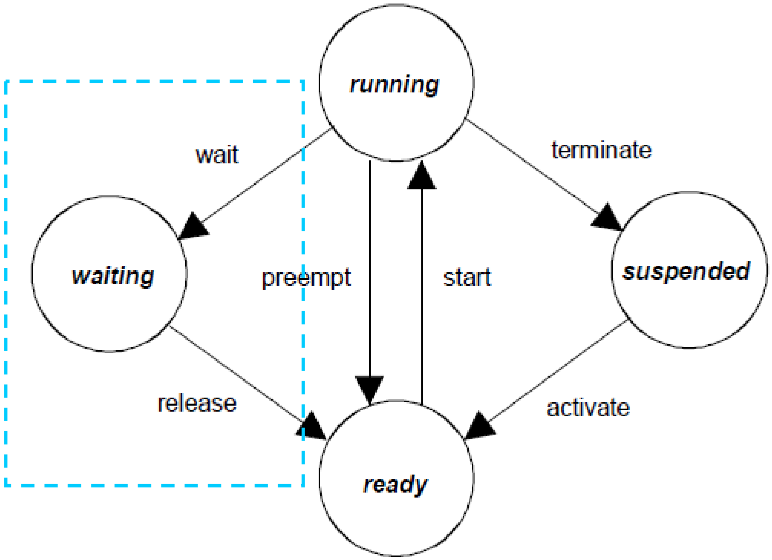
OSEK/VDX standard

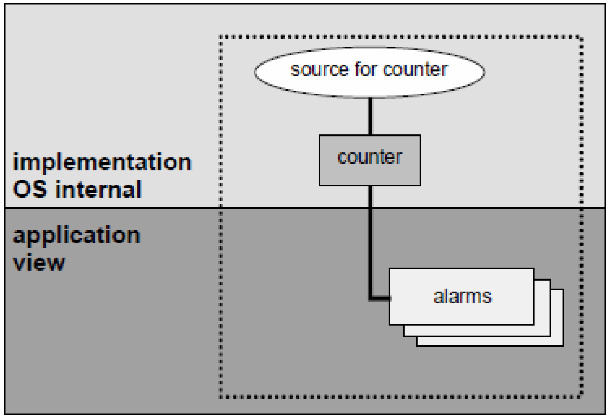
- Functional entities
  - Interrupts – two categories (with/without API calls)
  - Tasks – groups of tasks, support for application modes



OSEK – Runtime concepts

- Task conformance classes
  - Basic
    - terminate, preempt via task priority or interrupt
    - Sync points only at start/end
  - Extended
    - wait permitted
    - higher run-time resources requirement
- Recurring events
  - Counters – ticks
  - Alarms – trigger task or alarm callback routine



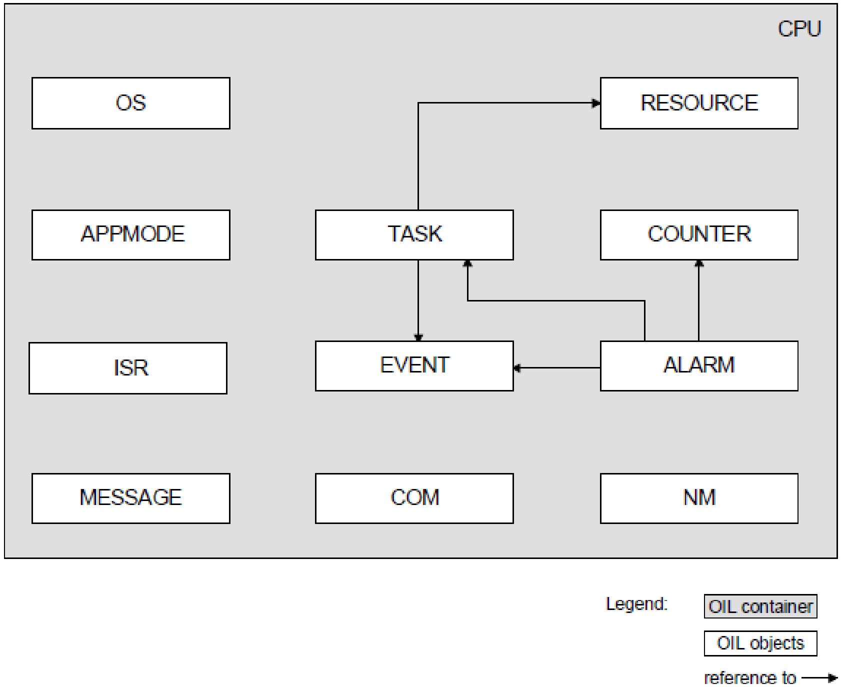


UNIVERSITÄT SALZBURG

OSEK OIL Configuration File

- Sample

```
CPU MyCPU {
...
    TASK TaskA {
          PRIORITY = 2;
          SCHEDULE = NON;
          ACTIVATION = 1;
          AUTOSTART = TRUE;
          RESOURCE = resource1;
          EVENT = event1;
    };
    COUNTER MyTimer {
          MINCYCLE = 16;
          MAXALLOWEDVALUE = 127;
          TICKSPERBASE = 90;
    };
    ALARM WakeTaskA {
          COUNTER = MyTimer;
          ACTION = ACTIVATETASK {
               TASK = TaskA;
          };
...
};
```



- Automatically generated by TDL compiler plugin for OSEK

OSEK - TDL Runtime - Details

- Dynamic module loading = not supported
- TDL Tasks = OSEK Basic Tasks
- Timing -> Counter + Alarm + TDL Scheduler
- Scheduling multiple modules -> RM offline or EDF offline + runtime
- Preemption – ActivateTask, ChainTask + OSEK scheduler
- Mode changes supported in TDL sense, not original OSEK*
- Resource management – TDL internal
- Interrupts – ONLY when required via OSEK ISR

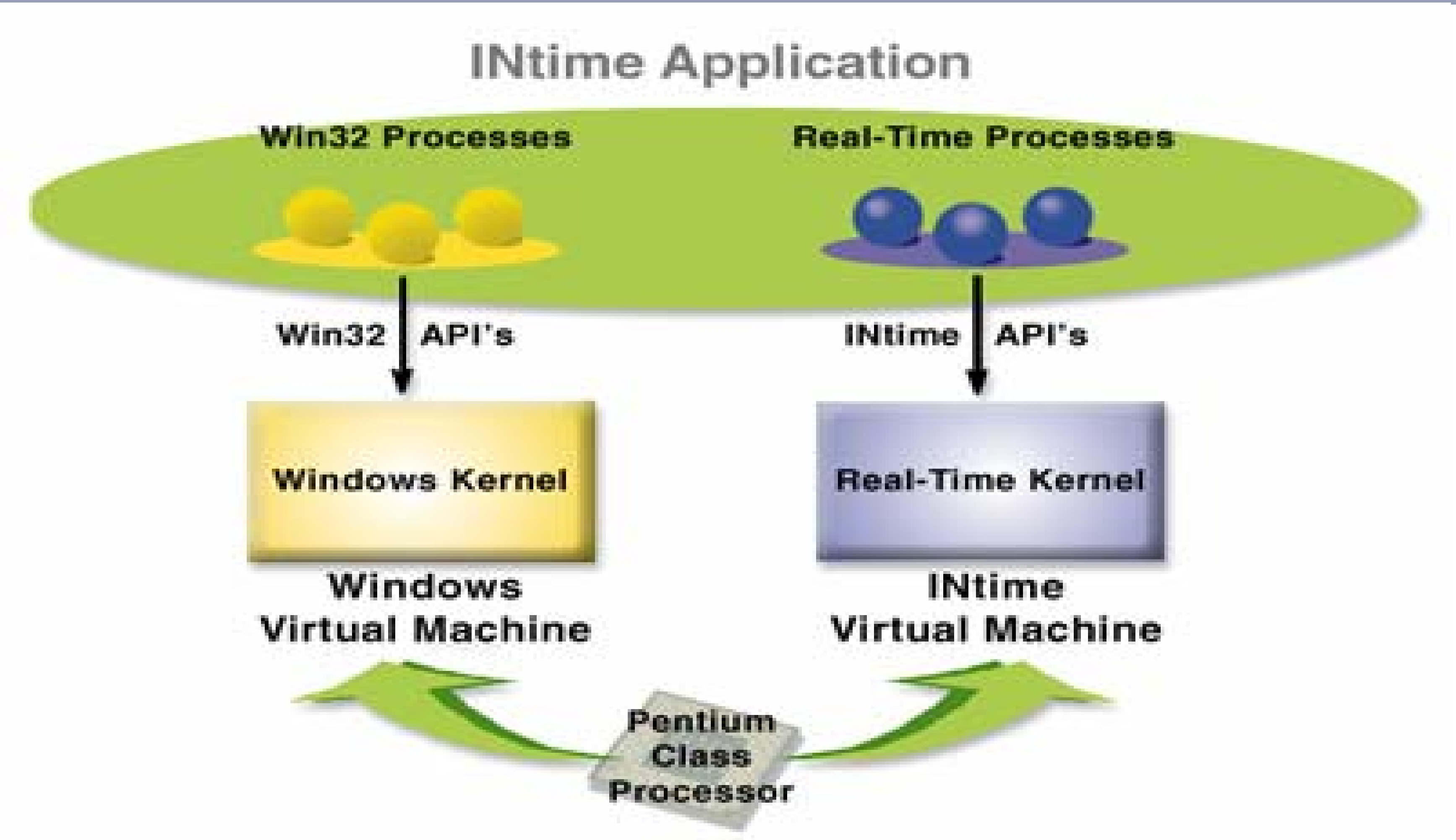
OSEK - TDL Runtime - Usage

Scenario: application consisting of module: MyModule

- Implementing functionality code:
  - MyModule.c contains the MyModule functionality
- Writing & compilation of the TDL application:
  - Generated timing (E-code)+ wrapper code for functionality: MyModule_TDL.c
  - Generated OIL files: NodeName.oil, MyModule.oil
- OIL files + C sources compilation: MyModule, MyModule_TDL.c + TDL Runtime libraries + *.oil => MyApplication.elf
- Transfer/flash MyApplication.elf onto embedded platform

UNIVERSITÄT SALZBURG

InTime – What is it?



- RTOS developed by TenAsys (iRMX succesor)

InTime - Advantages

- Enhancement of Windows 2000/XP with RT capabilities
- Seamless integration with MS Visual Studio (C/C++)
- Same development platform from GUI to embedded device – C/C++
- Rapid prototyping
- Cheap and fast hardware
- Memory protection for real-time processes from Windows apps
- Debugging facilities for real-time processes from Windows space
- Real-time TCP/IP networking* (limited HW devices)
- RT object browsing + performance analysis

InTime - Disadvantages

- Only Intel Pentium/Pii/P3/P4/Xeon supported
- Constricted design because of PC architecture
- High cpu/memory/power requirements – industrial design only
- Proprietary OS, API – vendor lock-in
- Slow time related primitives (mostly because of PC RTC)
- Single CPU support only* (multi-cpu in progress)
- Only 256 priority levels, some reserved for OS or HW (RM becomes complicated on complex designs)

InTime – TDL Runtime - Details

- Dynamic module loading = possible but not supported yet in TDL tool-chain
- TDL Tasks = Pre-allocated InTime real-time threads
- Timing -> kernel level Alarm + TDL Scheduler
- Scheduling multiple modules -> Offline RM or Hybrid EDF (offline+runtime)
- Preemption -> thread priorities manipulation/thread suspend/resume
- Debugging – console
- Fast, up to 10KHz task freq (50KHz with APIC tweak)

InTime - TDL Runtime - Usage

Scenario: application consisting of module: MyModule

- Implementing functionality code:
  - MyModule.c contains the MyModule functionality
- Writing & compilation of the TDL application:
  - Generated timing (E-code)+ wrapper code for functionality: MyModule_TDL.c
- C sources compilation: MyModule, MyModule_TDL.c + TDL Runtime libraries => MyApplication.rta
- Execution of MyApplication.rta under InTime RTOS

POSIX - Advantages

- Big standard covering multiple areas
- Wide range of implementations available
- UNIX* derivatives available on most embedded platforms (Linux, QNX, RTLinux, RTAI, RTEMS, …)
- C/C++ based
- Free tool-chains available for most platforms
- Flexible
- …

POSIX - Disadvantages

- Ancient design
- Not all features actually used/required for embedded platforms
- C usage may result in hard to maintain code
- No automatic memory management/garbage collection
- …

POSIX – TDL Runtime - Details

- Dynamic module loading = possible
- TDL Tasks = Pre-allocated POSIX real-time threads, or plain user-level threads for simulations
- Timing -> nanosleep + TDL Scheduler
- Scheduling multiple modules -> Offline RM or Hybrid EDF (offline+runtime)
- Preemption -> thread priorities manipulation
- Debugging – console
- Fast and lightweight

POSIX – TDL Runtime - Usage

Scenario: application consisting of module: MyModule

- Implementing functionality code:
  - MyModule.c contains the MyModule functionality
- Writing & compilation of the TDL application:
  - Generated timing (E-code)+ wrapper code for functionality: MyModule_TDL.c
- C sources compilation: MyModule, MyModule_TDL.c + TDL Runtime libraries => MyApplication (ELF – simulation), alternate MyApplication kernel module
- Execution of MyApplication under POSIX compatible RTOS

UNIVERSITÄT SALZBURG

OSEKtime – Brief overview

- Adjacent standard to OSEK
- Target: Distributed Embedded Control Units
- Design philosophy:
  - Predictability
  - Modularity
  - Dependability
  - OSEK compatible
  - Minimal resource requirements

OSEKtime - Architecture

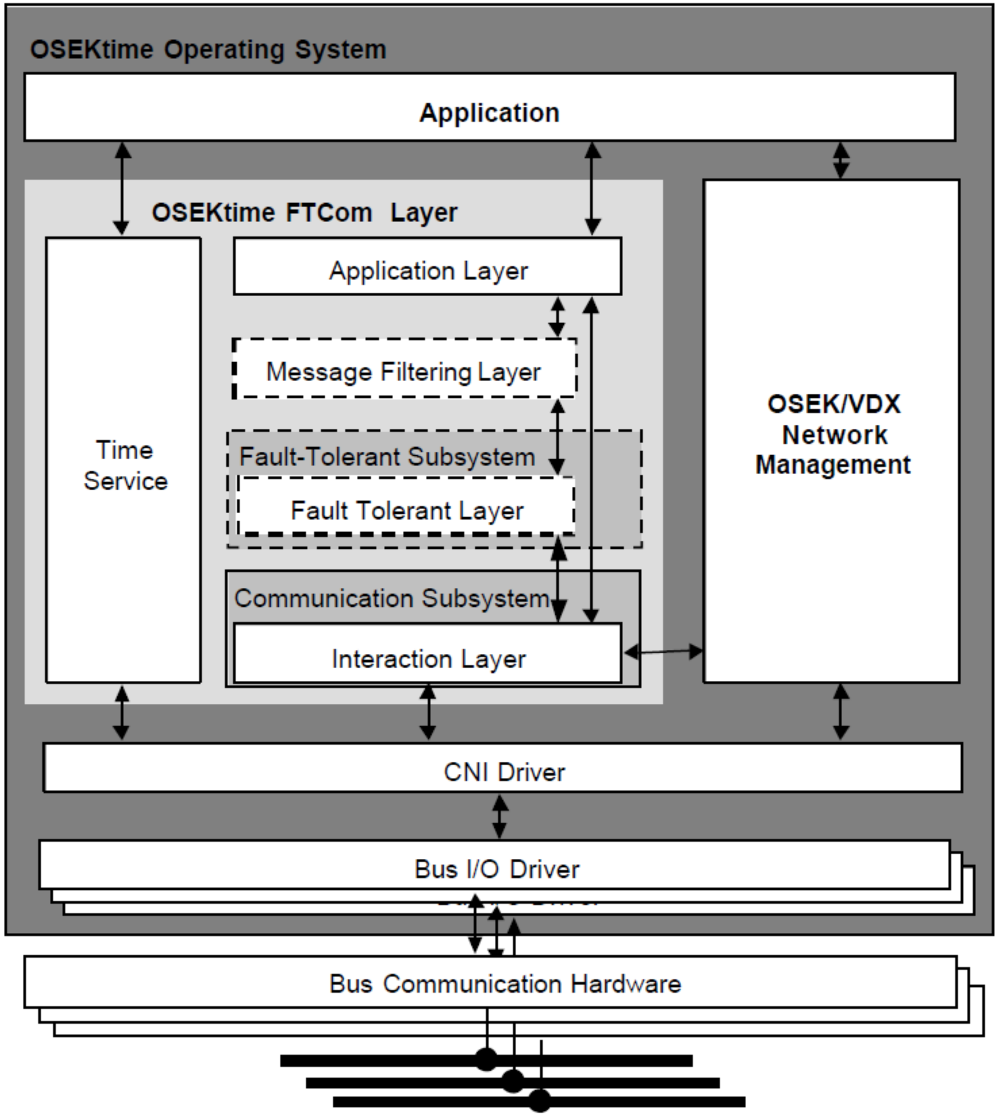
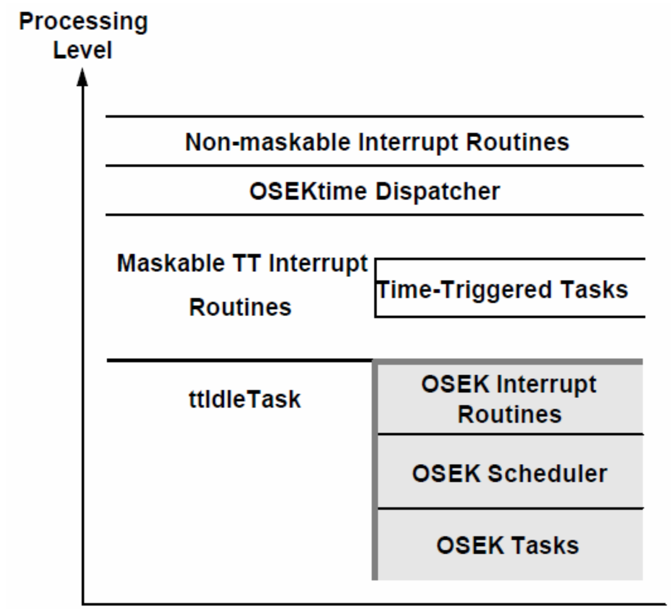
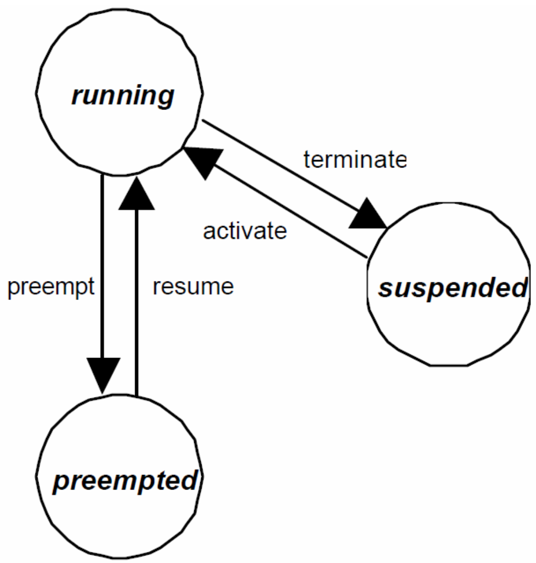


Figure © OSEKtime



OSEKtime - Advantages

- Static scheduling
- All basic RT services
  - Clock synchronization
  - Task management
  - Interrupt handling
  - Error detection
  - Fault Tolerance – via FTCom



- Preemptive multitasking + Time-triggered tasks may preempt OSEK non-preemptive tasks
- Support for mode switches

OSEKtime - Disadvantages

- Compared with OSEK, Tasks cannot wait for resources!
- Static dispatch table
- Difficult multi-application multi-mode support (because of static dispatch table)