Real-Time Operating Systems: Some Examples

Prof. Dr. Jian-Jia Chen (and Colleagues)

Department of Computer Science, Chair 12
TU Dortmund, Germany

05.11.2019, Embedded Systems WS 19/20

Outline

Embedded Linux

OSEK/VDX

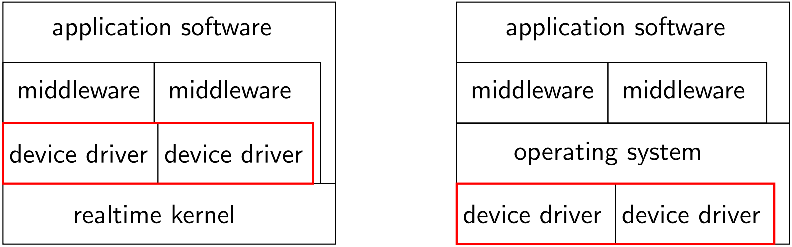
Reference

Embedded Operating System

Device drivers are typically handled directly by tasks instead of drivers that are managed by the operating system:

- This architecture *improves timing predictability* as access to devices is also handled by the scheduler.
- If several tasks use the same external device and the associated driver, then the access must be carefully managed. (shared critical resource, mutual exclusion etc.)

Embedded OS | Standard OS



Embedded Linux

- Adaptation of a well-tested code base with the required functionality to run in an embedded context.
- Linux has become the OS of choice for a large number of complex embedded applications following this approach.
  - However, integrating a number of different additional software components is a complex task.
  - May lead to functional as well as security deficiencies.
- These applications benefit from easy portability
  - Linux has been ported to more than 30 processor architectures, including the popular embedded ARM, MIPS, and PowerPC architectures
  - The system's open-source nature, which avoids the licensing costs arising for commercial embedded operating systems.

An Example: LibC Optimization

libC: the C library, which provides basic functionality for the file I/O, process synchronization and communication, string handling, arithmetic operation, and memory management.

libc version	musl	uClibc	dietlibc	glibc
Static library size	426 kB	500 kB	120 kB	2.0 MB
Shared library size	527 kB	560 kB	185 kB	7.9 MB
Minimal static C program size	1.8 kB	5 kB	0.2 kB	662 kB
Minimal static “Hello, World” size	13 kB	70 kB	6 kB	662 kB

- musl: optimized for static linking
- uClibc: designed for systems without MMU (memory management units)
- dietlibc: target for smallest possible size to compile and link programs
- glibc: standard Linux GNU libC

Busybox - All Linux Utilities in ONE Executable

Originally aimed to put a complete bootable system on a single floppy disk that would serve both as a rescue disk and as an installer for the Debian distribution

- Only one program for over 200 utilities, for example: sh, cat, tail, echo, vi, nc, tr, sed, ifconfig, dmesg, lsmod, insmod, fsck
- Share code for parsing args, common functions
- Usually statically built
- The binary is linked to several file names
  - When busybox is executed, it checks out its argv[0], and assumes this to be the applet to execute
  - The other arguments are parsed
- A whole “linux” userspace in a single command.

Challenges of Using Linux for Embedded Computing

Adopting Linux to typical embedded environments poses a number of challenges due to its original design as a server and desktop OS.

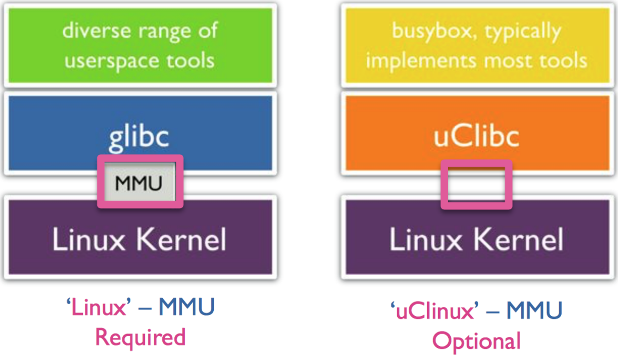
- Limited resources available within embedded system (CPU, storage, RAM, and so on).
- Complex structure and large size $\rightarrow$ optimization for the implementation of C library.
- Guarantee Real-Time properties is the most complex challenges $\rightarrow$ some Linux kernel extensions are available e.g., RTAI [3], RT-Linux [1], etc.

Real-Time Properties in Linux

Since Linux version 3.14 (in 2014), a configuration option SCHED_DEADLINE has been added to Linux:

- Supports for the earliest-first-deadline (EDF) scheduler and different real-time schedulers (to be detailed later)
- Coexist with other non-real-time schedulers
- Tutorials are available in the Internet:
  - Basic knowledge of real-time schedulers
  - Constant bandwidth server (not covered in this lecture)
  - Multiprocessor scheduler (to be detailed later)
- Limitations:
  - not suitable for hard real-time systems (some routines do not have hard real-time bounds), although you may see hard guarantees in some documents
  - for EDF, applications must be modified to signal the beginning/end of a job (some kind of startjob()/endjob() system call)

μClinux



source: https://elinux.org/images/3/35/Austin-uClinux_ELC_43_small.pdf

- MMU can be optional in μClinux
- Is this a good thing or not?
  - COW (copy on write) is forbidden $\Rightarrow$ NO fork..... Use vfork
  - many limitations
- However, the OS size remains a few MB in RAM, which is too big for some micro-controllers

$LITMUS^{RT}$

Linux Testbed for Multiprocessor Scheduling in Real-Time Systems ($LITMUS^{RT}$) [2] is a real-time extension of the Linux kernel.

$$Linux\ kernel\ patch \begin{cases} RT\ schedulers \\ RT\ synchronization \\ [cache\ and\ GPU] \end{cases}$$

$$+$$

$$user\ space\ interface \begin{cases} C\ API \\ device\ files \\ scripts and tools \end{cases}$$

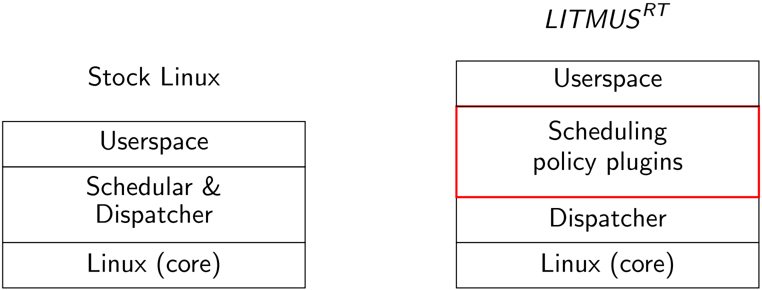
$$+$$

$$tracing\ infrastructure \begin{cases} overheads \\ schedules \\ kernel\ debug\ log \end{cases}$$

$LITMUS^{RT}$

$LITMUS^{RT}$ enables practical multiprocessor real-time systems research under realistic conditions.

- Allow implementation and evaluation of novel multiprocessor schedulers and synchronization protocols.
- Based on Linux, multiple useful tools are available (debug, schedule trace, and overhead trace).
- Flexible, fine-grained measurement of different overheads.



LITMUS^{RT}

However, *LITMUS*^{RT} is only a testbed for the researchers to develop and test their schedulers, resource sharing protocols, and other real-time properties, rather than a real real-time operating system!

More information, please refer to their website:
https://www.litmus-rt.org/documentation.html

Evaluating the Use of Linux in Embedded Systems

- Technical side
  - POSIX-like API which enables easy porting of existing code
  - free-of-charge development tools and integration tools
  - well-tested code base (thanks to many active users)
  - Complex code base for debugging and verification
- Legal/Business side
  - Benefits due to the availability of the source code free of cost
  - However, GPL License version 2 governs that the source code for modification has to be published as well $\Rightarrow$ secrete leakage?
- Security side
  - distributed denial of service (DDoS) attacks for non-updated Linux versions
  - updates (due to security vulnerabilities have to be planned for an embedded Linux

Outline

Embedded Linux

OSEK/VDX

Reference

OSEK/VDX

- OSEK/VDX stands for:
  "**O**ffene **S**ysteme und deren Schnittstellen für die **E**lektronik im **K**raftfahrzeug / **V**ehicle **D**istributed E**x**ecution"
- OSEK was started by german vehicle manufacturers in 1993.
- VDX was a similar project in France and joined OSEK in 1994.
- Definition in the OSEK Specifications 2.2.3:
  "The specification of the OSEK operating system is to represent a uniform environment which supports efficient utilisation of resources for automotive control unit application software. The OSEK operating system is a single processor operating system meant for distributed embedded control units."

Goals of OSEK/VDX

- OSEK is designed to:
  - offer necessary functionality to support event driven control system with stringent real-time requirements,
  - keep resource requirements minimal,
  - support a wide range of hardware,
  - ensure portability of application software,
  - realize standardised interfaces (ISO/ANSI-C-like),
  - be scalable, and
  - support for automotive requirements.
- In this lecture we focus on:
  - features of OSEK,
  - task management,
  - event mechanism,
  - resource management, and
  - alarms.

Two special features of OSEK kernels

- All kernel objects are statically defined
  - No dynamic memory allocation (most of them)
  - No dynamic creation of jobs (most of them)
  - OIL file specifies the objects off-line (# of tasks, size of stack)
- Stack Sharing support
  - RAM is expensive on micro-controllers
  - Persistent state is not stored in the stack
  - Related to how task code is written:

```
Task(x){
  int local;
  initialization();
  for(;;){
    do_instance();
    end_instance();
  }
}
```

Listing 1: Extended Task

```
int local;
Task x(){
  do_instance();
}

System_initialization(){
  initialization();
}
```

Listing 2: Basic Task

OIL: OSEK Implementation Language

```
TASK Task1 /* Definition of tasks */
{
  AUTOSTART = FALSE;
  PRIORITY = 7;
  ACTIVATION = 1;
  SCHEDULE = FULL;
  STACKSIZE = 4096;
};
ALARM Task1_Alarm
{
  COUNTER = SysTimerCnt;
  ACTION = ACTIVATETASK
  {  TASK = Task1;  };
  AUTOSTART = TRUE
  {
    ALARMTIME = 1;
    CYCLETIME = 8000;
  };
};
```

Task management

- In OSEK tasks are subdivided parts of control software.
- Tasks can be specified according to their real-time requirements.
- OSEK introduces two different task concepts:
  1. Basic tasks only release the CPU, when
     - they terminate,
     - the OSEK-OS loads a higher-priority task, or
     - an interrupt occurs.
  2. Extended tasks are additionally allowed to use the system call `WaitEvent`.

Task state model

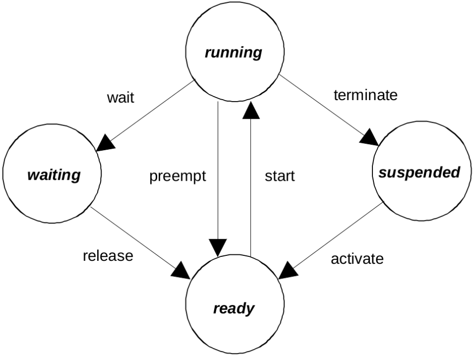


Figure: Task state model of extended Tasks.

Note: Basic tasks do not have the waiting state.

Scheduling policy

- Fully preemptive scheduling
  - A task in the running state will be put into ready state, as soon as a higher-priority task gets ready
  - In fully preemptive systems, the programmer shall constantly expect preemption of his/her task
- Non-preemptive scheduling
  - The activated higher-priority tasks have to wait for the running task to terminate
  - Rescheduling only takes place after the task termination, waiting or the scheduler gets called by the currently running task

Event mechanism

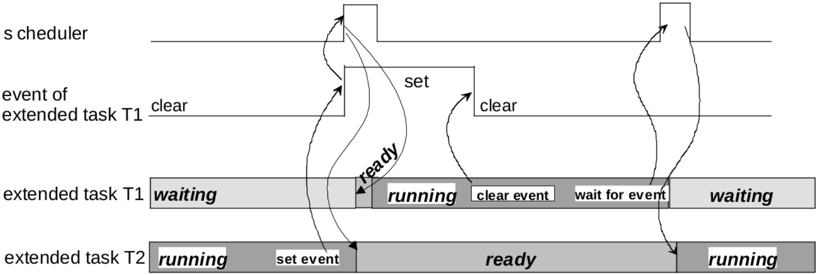


Figure: Task synchronization with fully preemptive tasks using an event.

When events are used with non-preemptive tasks, the scheduler should be called after clearing an event.

Resource management

- OSEK uses the Immediate Priority Ceiling protocol (PCP) to prevent deadlocks and improve data integrity.
  - The resource usage has to be specified in the OIL configuration files.
  - The calculation of the priority ceiling is done via the OIL compiler.
- In OSEK resources can also be used to call the scheduler in non-preemptive tasks.
- Resources can also be used by interrupt service routines (ISR) and can prevent interrupts during task run time.
- The task is not allowed to terminate, wait or call the scheduler while it holds resources.

Alarm management

- Alarms manage reoccurring events in the OSEK-OS.
- Alarms are always bound to counters.
  - Counters are represented by a value "ticks".
  - OSEK doesn't standardise an API to manipulate counters.
  - The OSEK-OS takes care of advancing the counters "ticks".
  - OSEK-OS's must provide at least one counter deriving from a timer.
  - more than one alarm can be attached to a counter.
- Alarms can activate tasks, set events or call an alarm-callback routine (user defined).
- Alarms can be single alarms or cyclic.

Alarm model

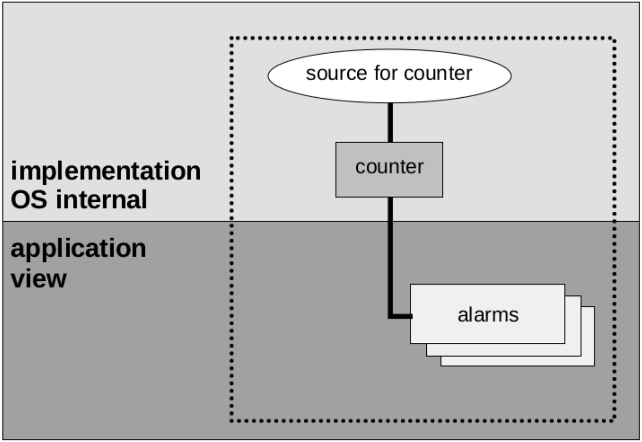


Figure: Layered model of alarm management.

ERIKA Enterprise

- An open-source OSEK/VDX Hard RTOS
- v2.x (**certified OSEK/VDX compliant**)
  - Hard Real-Time with FP-Scheduling and Immediate PCP
  - Support for *EDF* and Resource Reservation Schedulers
  - Support for *stack sharing* among tasks
  - 1-4KB Flash footprint, for 8-32 bit microcontrollers
- v3.x
  - Support Limited Preemption
  - Support for manycore platforms (Partition and Global Scheduling)
  - Single copy of RTOS among all cores, whereas v2.x requires one copy of per core
  - 1-4KB Flash footprint, for 8-64 bit microcontrollers

Conformance Classes

- Supported by the OSEK/VDX standard (also ERIKA)
- BCC1: Smallest class supporting 8 tasks with different priorities and one shared resource
- BCC2: BCC1 + one task with multiple activations
- ECC1: BCC1 + Extended tasks that can wait for an event
- ECC2: BCC1 + the above two additional features

- ERIKA provides additional two classes:
  - EDF (earliest-deadline-first scheduling) optimized for small micro-controllers
  - FRSH: EDF extension providing resource reservation scheduler

Direct Interrupts Control

- Tasks are scheduled by the scheduler
- Interrupts are scheduled by hardware
- Two types of Interrupt Service Routines (ISR):
  - Category 1: simpler and faster, does not implement a call to the scheduler at the end of the ISR
  - Category 2: this ISR can call some primitives that change the scheduling behavior. The end of it is a rescheduling point
- Only a subset of system API services are allowed in ISR

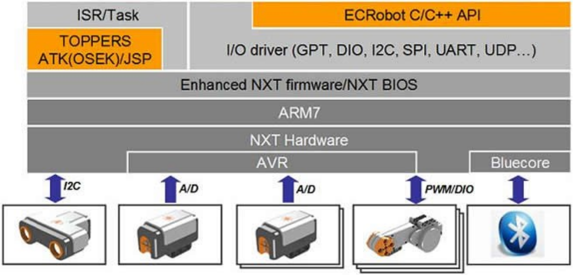
EV3OSEK

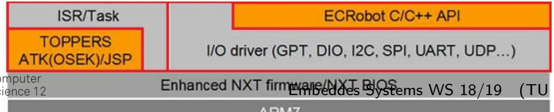
- EV3OSEK is an OS for Lego Mindstorms EV3 (2013).
  - Aims to fulfill the OSEK standard.
  - NXTOSEK port by Westsächsische Hochschule Zwickau.
  - Used in exercise sessions.

- SoC TexsInstruments AM1808
  - ARM926EJ-S
  - ARM9
  - 300MHz
  - 64 MB RAM
- LEGO motor and sensor compatible

NXTOSEK and EV3OSEK

- NXTOSEK is an OS for Lego Mindstorms NXT (2006).
  - uses Toppers/JSP or Toppers/ATK(OSEK) kernel.
  - has to be flashed on the brick.
- Only the Toppers ATK(OSEK) kernel has been ported to EV3.
- ECRobot API in EV3OSEK supports less hardware in EV3.





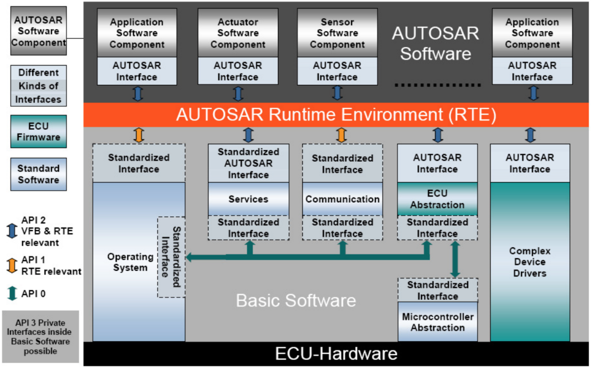
AUTOSAR

AUTOSAR - **AUT**omotive **O**pen **S**ystems **AR**chitecture

- Middleware and system-level standard, jointly developed by automobile manufacturers, electronics and software suppliers and tool vendors. More than 100 members
- Motto: “cooperate on standards, compete on implementations” Reality: current struggle between OEM and Tier1 suppliers
- Target: facilitate portability, composability, integration of SW components over the lifetime of the vehicle
- AUTOSAR provides a set of specifications based on standardized exchange format for
  - Basic Software modules,
  - application interfaces, and
  - a common development methodology.

Three Layer Architectures

- Basic Software: standardized software modules
- Runtime environment(RTE): Middleware which describes information exchange between the application software components and between the Basic Software and the applications.
- Application Layer: application software components that interact with the RTE



AUTOSAR: Timing Extension

- Release 4.3.1 in Dec. 2017 (now free of charge for download)
- Created as a supplement to the formal definition of the Timing Extensions by means of the AUTOSAR meta-model
- Support constructing embedded real-time systems that satisfy given timing requirements and to perform timing analysis/validations of those systems once they have build up
  - Configure and specify the timing behavior of the communication stack.
  - However, the specification of analysis and validation results (e.g. the maximum resource load of an ECU, etc.) is not addressed in AUTOSAR Timing Extension.

Note: OSEK/VDX, AUTOSAR, and AUTOSAR Timing Extensions are standards for interfaces and format exchange. The validation of the correctness is not part of the specifications.

Reference

M. Barabanov and V. Yodaiken.
Real-time linux.
Linux journal, 23(4.2):1, 1996.

J. M. Calandrino, H. Leontyev, A. Block, U. C. Devi, and J. H. Anderson.
LITMUS^{RT}: A testbed for empirically comparing real-time multiprocessor schedulers.
In *Real-Time Systems Symposium (RTSS)*, pages 111–126. IEEE, 2006.

L. Dozio and P. Mantegazza.
Real time distributed control systems using rtai.
In *Object-Oriented Real-Time Distributed Computing, 2003. Sixth IEEE International Symposium on*, pages 11–18. IEEE, 2003.