Introductory

Designing Small-Scale Embedded Systems with μITRON Kernel

1st Apr. 1998

Hiroaki Takada

hiro@ertl.ics.tut.ac.jp
http://www.ertl.ics.tut.ac.jp/~hiro/

Toyohashi Univ. of Technology

Background

- Application fields of embedded systems continue to expand to *small-scale systems*.
- Most embedded systems are *real-time systems*, in that the systems have some *timing constraints*.
- Software for (even small-scale) embedded systems becomes *larger and more complex*.

↓

How to raise software productivity?

- high-level programming language
- real-time operating system (RTOS)
- advanced development environments and tools

Consumer Applications

TVs, VCRs, digital cameras, settop boxes, audio components, air-conditioners, microwave ovens

Office Applications / PC Peripheral

printers, scanners, disk drives, copiers, FAX

Communications

answering machines, ISDN telephones, cellular phones, modems, terminal adapters

Other Applications

automobiles (engine management, *etc.*), game gear, vending machines, electronic musical instruments, (some components for) factory automation

Typical Applications of Small-Scale Embedded Systems

Contents of this Class

- What is a real-time kernel? What is the advantage?
- application status of real-time kernels to small-scale embedded systems
- introducing some real-time kernels for small-scale embedded systems
  - μITRON
  - OSEK/VDX OS
  - μC/OS
- basic approaches and techniques in designing a small-scale embedded system with a real-time kernels

Small-Scale Embedded Systems

Following features are common to *many* (but not *all*) small-scale embedded systems.

- produced in great number and in cheap
  - ➡ The producing cost is a larger issue than the development cost.
- short development life-cycle
  - short time-to-market
  - The software is seldom modified once the product is shipped.
- limited hardware resources
  - small memory (*esp*. RAM) capacity
  - required to fit in a single-chip MCU
- high reliability (for some systems)

What is a Real-Time Kernel?

- real-time kernel (*or* real-time operating system kernel)
  also called *as* a real-time monitor
  or a real-time executive
  - is the basic run-time software on which a *real-time system* is realized.
  - is the *core module* of a real-time operating system.
  - manages only the essential hardware resources of a computer system (*i.e.* processor and memory.

↑

There are only *a few common I/O devices* to be supported in case of small-scale embedded systems.

- functions supported by a real-time kernel
  - multitasking
    - *priority-based preemptive scheduling*
  - inter-task synchronization and communication
    - semaphore, eventflag, mailbox, ...
  - basic memory management
  - interrupt handling
  - timer handling, time management
  - ✗ *no external device handlings*
- Real-time kernels *were* difficult to apply to small-scale embedded systems

  because of the *overhead* of real-time kernels in
  - execution time
  - memory consumption

What is Multitasking?

- task
  - A task is a unit of concurrent processing.
  - The programs within a task are executed sequentially, while programs of different tasks are executed *concurrently*.
- task dispatching (*or* task switching)
  - changing the executed task
  - The context of the old task is saved, and that of the new task is restored.
- scheduling
  - *selecting* the executed task among the executable ones
- scheduling algorithm

Priority-Based Preemptive Scheduling

- priority-based scheduling
  - Each task is assigned a priority.
  - The task having the highest priority is *selected* as the executed task.
  - Lower priority tasks are never executed until the highest priority task is blocked (*or* suspended).
- preemptive scheduling
  - If a higher priority task is started while a lower priority task is being executed, the execution of the lower priority task is suspended and the higher priority task starts execution.
    - *preemption*

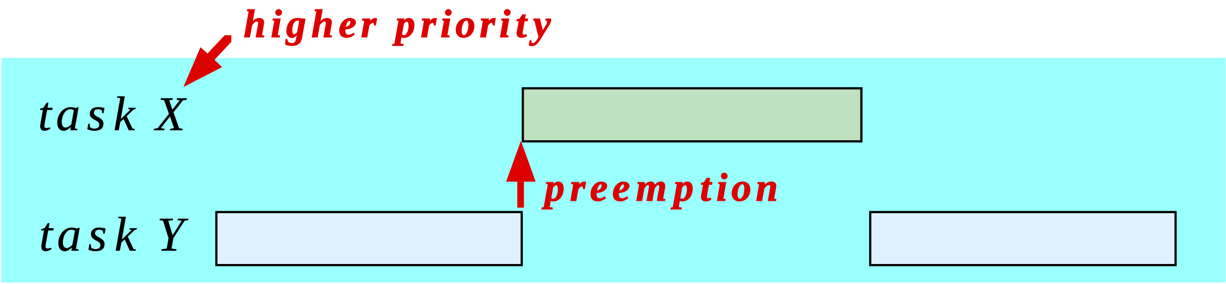
Why You Need Multitasking?

- modular design
  - ➡ *relatively unimportant in case of small-scale systems*
  - ◆ Modular design is effective for raising the maintainability and reliability of the system.
  - ◆ Different groups of I/O devices should be handled with different tasks, for example.
- real-time property
  - ➡ *major concern in this class*
  - ◆ The development and maintenance of a system with *real-time constraints* can be facilitated.

real-time constraints

… typically represented with *deadlines*

eg.) *task X* ... short execution time & short deadline
task Y ... long execution time & long deadline



→ easily realizable with a real-time kernel

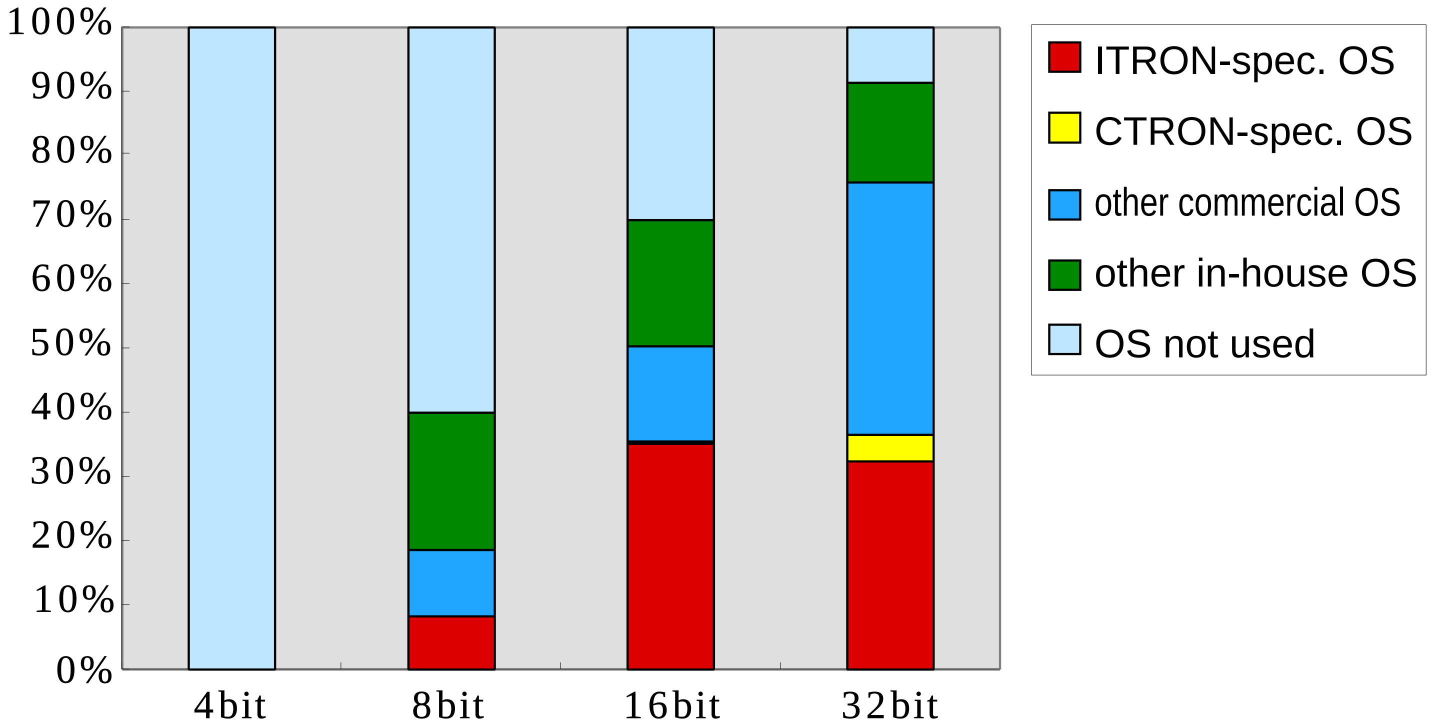
without a real-time kernel .....

- *Task Y* must repeatedly check if *task X* should be executed.

→ degraded response; overhead for the checking

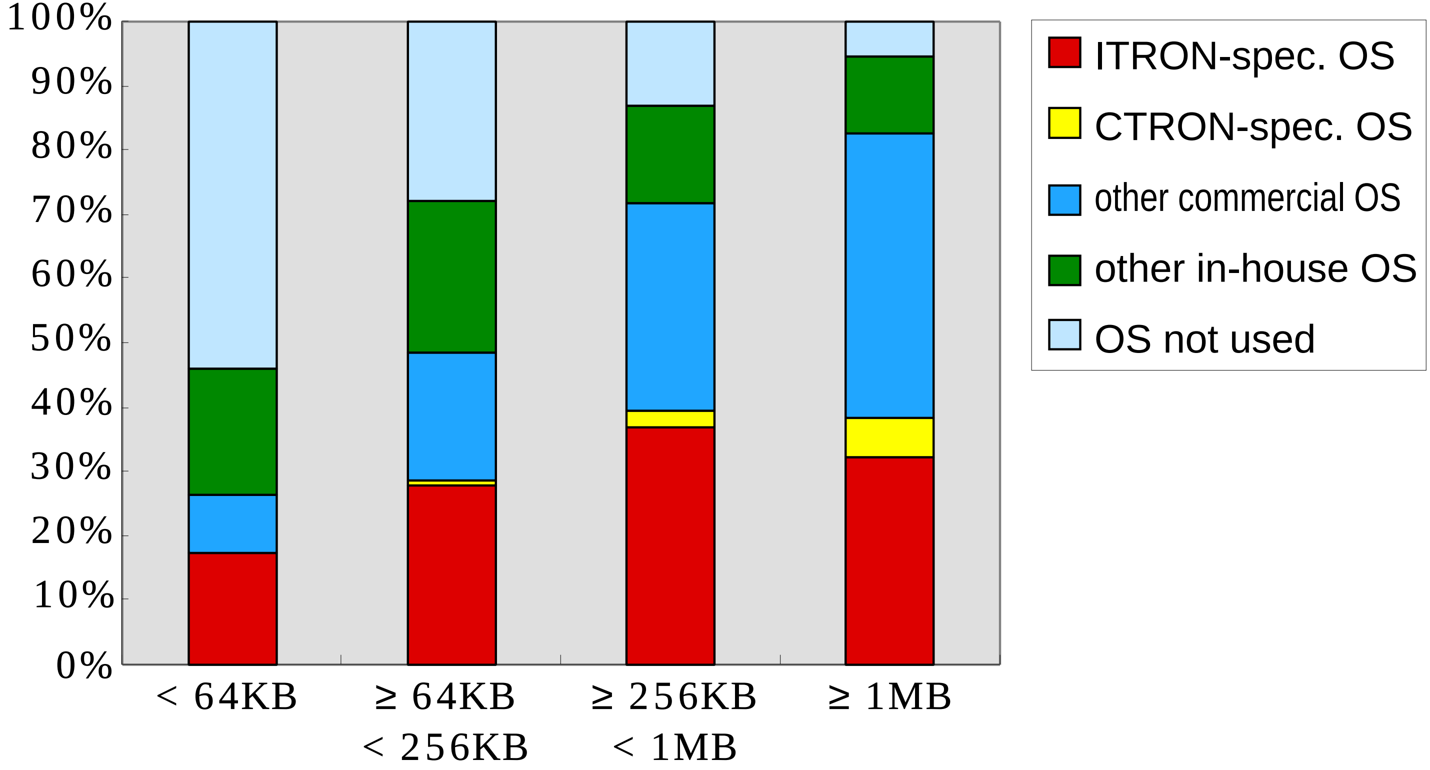
→ When *task Y* is modified, the check points must also be reexamined.

Application Status of Real-Time Kernels

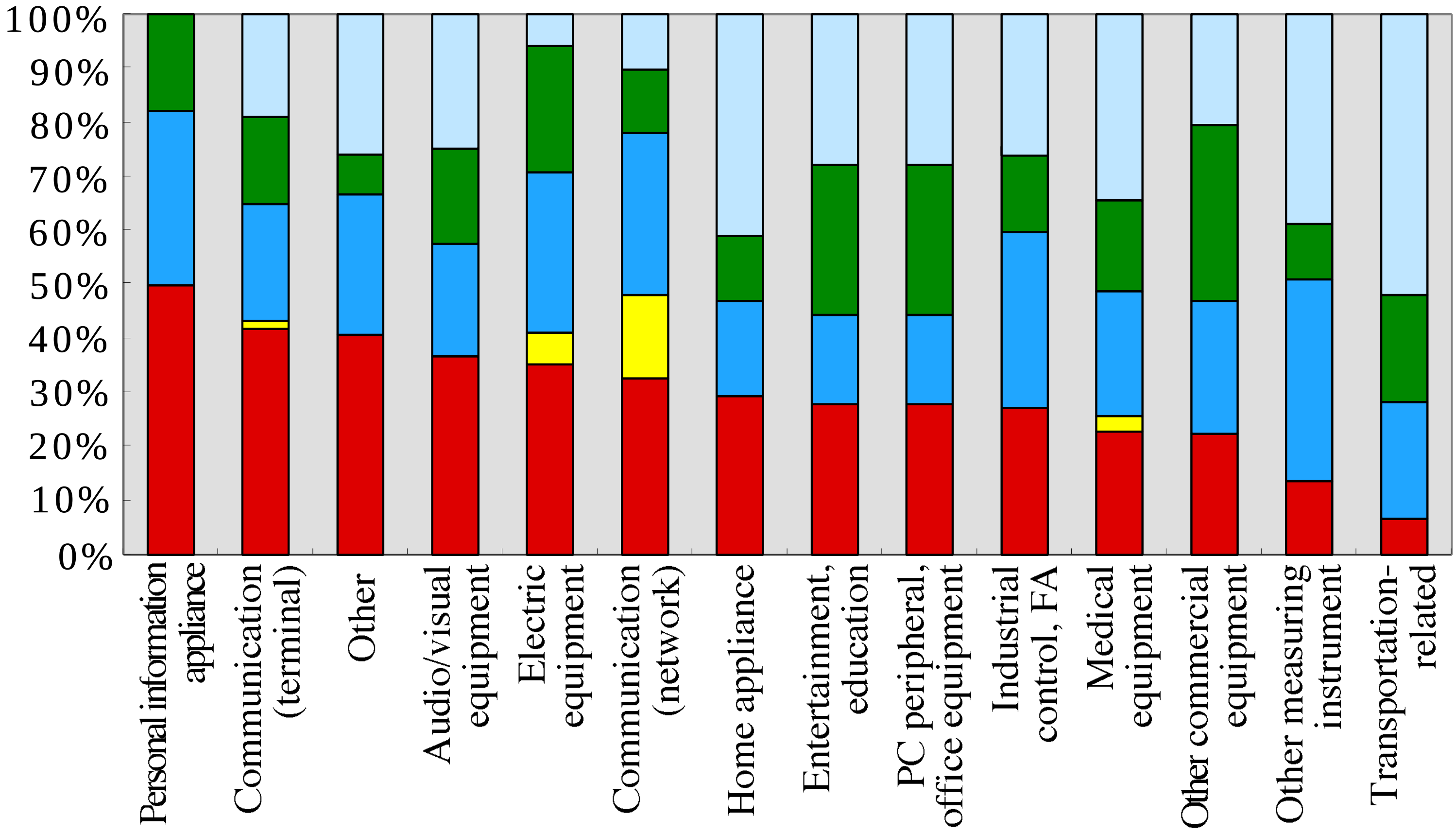


OS Usage vs. CPU Size

(TRON Association Survey, 1997-1998, Japan)



OS Usage vs. ROM Size



OS Usage vs. Application Fields

Application Examples of Real-Time Kernels

Application	FAX machine	CD player
MCU Type	16-bit	8-bit
RAM size	2 KB	512 Bytes
ROM size	32 KB	32 KB
Used Memory RAM	1346 Bytes	384 Bytes
ROM	28.8 KB	17.8 KB
No. of Tasks	6	9
No. of Interrupt Handlers	6	6
No. of Used System Calls	12	7
Kernel Size RAM (ratio)	250 Bytes (19%)	146 Bytes** (38%)
ROM (ratio)	2.5 KB (8.7%)	2.3 KB (13%)

* Both applications adopt μITRON-specification real-time kernels.

** A stack saving technique is applied.

Real-time kernels are applicable to such small embedded systems.

Requirements on a Real-Time Kernel

General requirements

(a) compactness

(b) low overhead

(c) dependability

(d) predictability

esp. for Small-Scale Embedded Systems

(a) scalability and adaptability

- The real-time kernel code should be tunable to a specific application.

(b) exploiting static information

- Static information should be placed on ROM area to save memory (RAM) consumption.

(c) low cost (of the real-time kernel itself)

Introduction to the μITRON Specifications

- ITRON Project
  standardizing real-time operating systems and related specifications for embedded systems
- A series of the *ITRON real-time kernel specifications* have been published and are widely used.
  ➡ *de-facto industry standard in Japan*
- μITRON specifications are designed for small-scale embedded systems with limited hardware resources.
  recent version: μITRON3.0
  under investigations: μITRON4.0
- The ITRON specifications are *open* in that anyone is free to implement and sell products based on them.

Design Principles of the ITRON Specifications

- design concept: *loose standardization*

  maximum performance cannot be obtained with strict standardization

- design principles
  - allow for adaptation to hardware; avoiding excessive hardware virtualization
  - allow for adaptation to the application
  - emphasize software engineer training ease
  - organize specification series and divide into levels
  - provide a wealth of functions

Functions Supported in μITRON3.0 Specification

- task management
- task-dependent synchronization
- basic synchronization and communication
  (semaphore, eventflag, mailbox)
- extended synchronization and communication
  (message buffer, rendezvous)
- interrupt management
- memory pool management
- time management
- system management

The specification can be downloaded from the ITRON Home Page.

http://tron.um.u-tokyo.ac.jp/TRON/ITRON

Implementation Status

! *We do not know how many real-time kernels are implemented based on the ITRON specifications.*

- about 45 registered implementations for about 35 processors
- several non-registered commercial implementations

implemented for almost all major processors

8-bit to 32-bit MCUs/MPUs

- many in-house implementations
- some freely distributed implementations

Application Status

- widely used for various application areas
- most popular RTOS specification in Japan

Implementation Examples

- Two μITRON-specification kernels for an MCU

OS type	Single-chip	General-purpose
No. of system calls	Task part: 29 Non-task part: 15	Task part: 36 Non-task part: 27
Scheduling	Fixed priority 1 task per priority	Variable priority
System call interface	Subroutine call	Software interrupt
Exception management	None	Exit exception, CPU exception
Wakeup request count	Max. 15	Max. 255
Semaphore count	Max. 255	Max. 65,535
System timer	32-bit	48-bit
Program size	0.6 – 4.4 KB	1.9 – 5.3 KB
Typical RAM use*	200 Bytes	640 Bytes
Task switching time**	17μS	32.5μS
Max. interrupt masking time**	9μS	9.5μS

* OS work area and various stack areas in the following configuration
tasks: 10, semaphores: 2, eventflags: 2, mailboxes: 2, external interrupts: 2 levels

** Clock 16 MHz, using on-chip memory

OSEK/VDX OS Specification

- OSEK/VDX Project

  standardizing an open-ended architecture for distributed control units in automobiles

- A real-time kernel API, and software interfaces and protocols for communication and network management are jointly specified.

- OSEK/VDX OS Specification
  - very compact real-time kernel specification targeted for *automotive* and *distributed* applications
  - version 2.0 released in Oct. 1997

http://www-iiit.etec.uni-karlsruhe.de/~osek/main.html

Designing with a Real-Time Kernel

- two important design issues with a real-time kernel
  - *How the system is decomposed into tasks?*
  - *How priorities are assigned to the tasks?*

theoretically ... (various assumptions omitted)

- *Deadline monotonic priority assignment is the optimal static priority assignment method, in order not to miss any deadlines.*

static ··· The priority of a task is assigned *statically*.

deadline monotonic

··· Task with *shorter deadline* should be assigned a *higher priority*.

↑

RMA (Rate Monotonic Analysis) theories

How Deadlines Look Like?

Example 1. (input – output relation)

- An LED must be lighted on within 500 msec after a switch is pushed.

deadline = 500 msec

Example 2. (input – output relation)

- A robot arm must be stopped within 200 msec after a collision is detected.

deadline = 200 msec – [*mechanical time*]

Example 3. (input – input relation)

- A data must be taken out of a buffer within 10 msec after the system receives the data. (Otherwise, the data may be overwritten by the next data.)

deadline = 10 msec

Decomposing into Tasks

Basic Guidelines

- Programs (*or* routines) started with different events should be included in different tasks.

  Each task is started with a kind of event.

- Programs with different deadlines should be included in different tasks.

  ↓

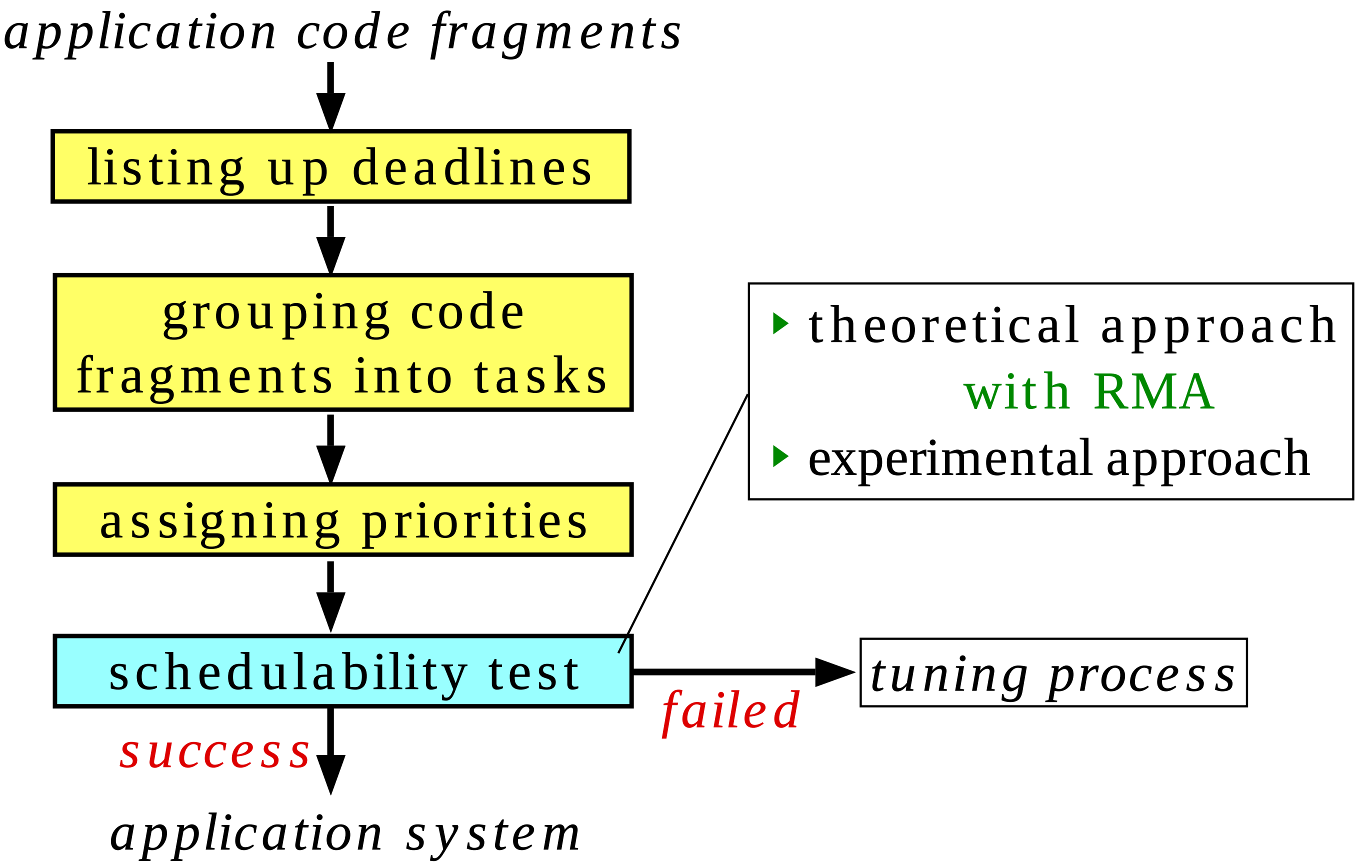
  Deadline monotonic priority assignment becomes possible.

Another Guideline (from another motivation)

- decomposition for modular design

  eg.) Different groups of I/O devices should be handled with different tasks.

Basic Design Flow



Another Type of Timing Constraints

Example.

- A series of data must be sent to an output port every 100 msec. (Permissible error of the period is 1%.)

  ↓ *translate*

  The program to send a data to the port should be started at 99 msec after the previous data is sent, and its deadline is 2 msec.

- How if some pre-processing is necessary for preparing the next data and its execution time is longer than 2 msec?

  ↓

  The deadline of the pre-processing is longer.

  Decompose into separate tasks !

Realizing Mutual Exclusion

mutual exclusion

··· When a task is accessing a shared resource, the other tasks must not access it.

- disabling task dispatchings (or interrupt services) while a task is accessing a shared resource
- using a semaphore

  A semaphore (or an equivalent) is supported with most real-time kernels.

  disadvantages

  - *priority inversion problem*
  - *Contending tasks may be blocked.*

- stack resource policy

  sometimes called as priority ceiling protocol

  - Before a task accesses a shared resource, the priority of the task is temporarily raised to the same or a higher level than any other task that can access the same shared resource.
  - After the access, the priority of the task is recovered to its original level.

advantages

- *Tasks are never blocked for mutual exclusion.*

limitations

- A task must not be blocked while it is accessing a shared resource.
- Which task accesses which shared resource must be known beforehand.

APIs with μITRON Kernel

- creating a task (statically)

  `cre_tsk / CRE_TSK`

  The initial priority of the task is passed as a parameter.

- starting / terminating a task

  `sta_tsk / ext_tsk`

- changing the priority of a task

  `chg_pri`

- obtaining / releasing a semaphore

  `wai_sem / sig_sem`

- Many other APIs (about 100) are defined in the μITRON specifications.

↓

Only used code is linked to the application.

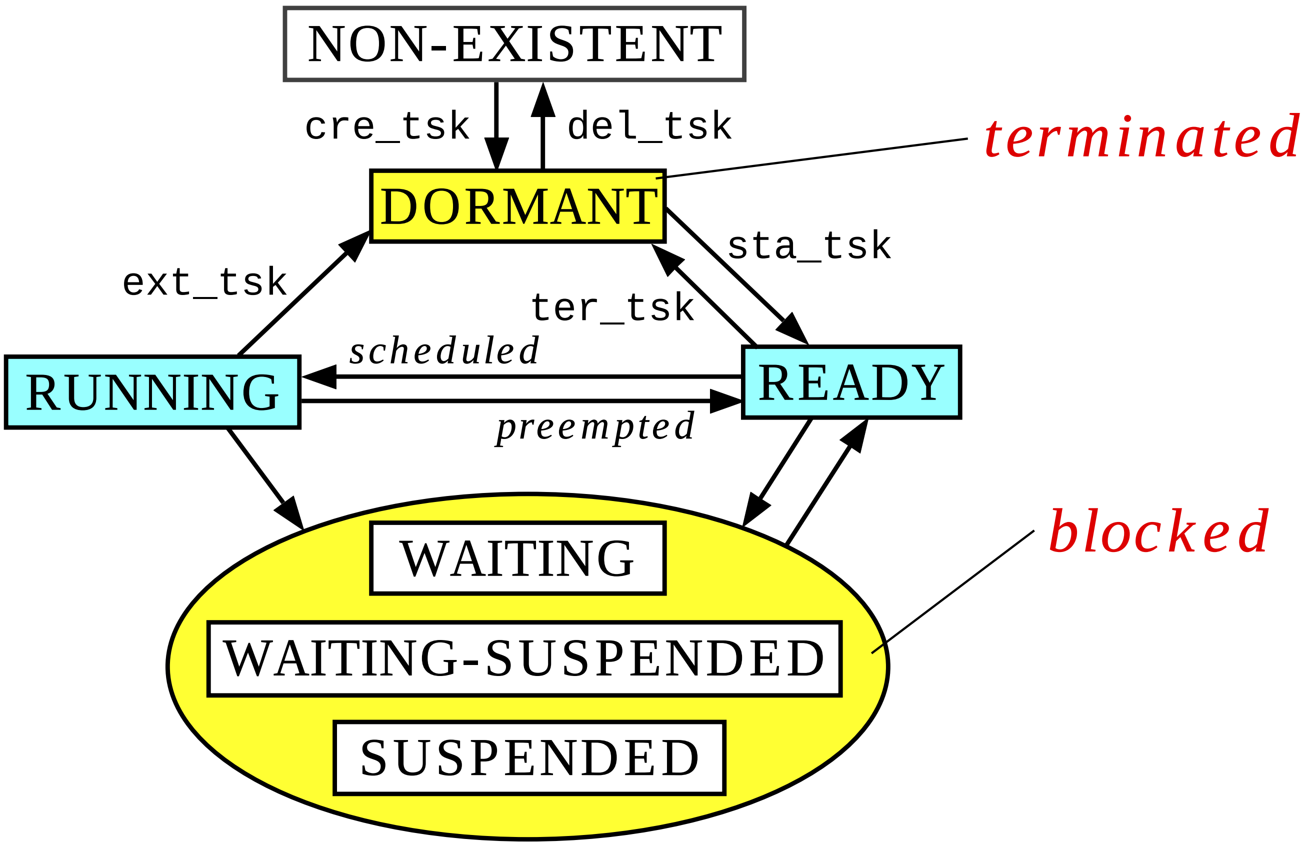
Blocked vs. Terminated

- A blocked task will resume execution from the blocked point.
  - *The task context must be saved during the task is blocked.*
- A terminated task will be started from the beginning.
  - *The task context need not be saved when it is terminated.*
- All the tasks that are never blocked can share a stack (if the real-time kernel supports it).

  Apply to the other synchronization pattern !

! Notice that the name of task states are different for each real-time kernel.

Task States with μITRON Kernel



Relaxing the Basic Guidelines

- Naive application of the basic guidelines may result in too large number of tasks.
  → *not good for saving memory space*

↓

- The basic guidelines can be relaxed.
  - A task with longer deadline can be included in a task with shorter deadline.
  - An event-driven task can be included in a periodic task if the period is enough shorter than the deadline of the event-driven task.
  - *many others ...*

Summary

- A real-time kernel is an effective even for small-scale embedded system.
- trade-off between the elegant programming style and the efficiency (esp. memory consumption)
- A basic approach in designing with a real-time kernel is introduced.
  - How to decompose the system into tasks?
  - How priorities are assigned to the tasks?

An effort in the ITRON Project

- establishing "application design guidelines" for real-time systems

  important for the circulation of software components

For Further Information

- ITRON Home Page

  http://tron.um.u-tokyo.ac.jp/TRON/ITRON/

- RMA (Rate Monotonic Analysis)

  [6] A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems

- Intermediate Courses on RTOS Use

  311–321 Multitasking Design and Implementation Issues in Embedded Systems

  331–341 RTOS Design: How Your Application is Affected

- presentation material of this class (*will be available within a week*)

  http://www.ertl.ics.tut.ac.jp/~hiro/escs98-ohp.pdf