Real-Time Operating Systems

Execution Models and Scheduling Foundations · ES/CPSF

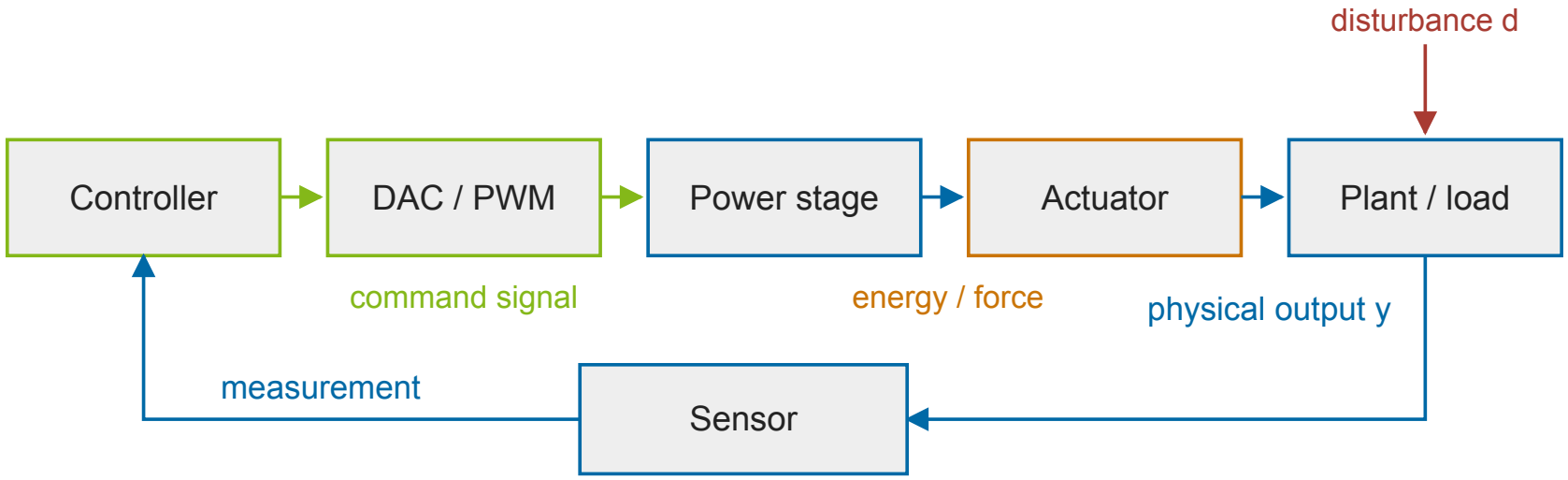
Peter Ulbrich

Course website

Winter Semester 2026/27

The Motor Controller Has a Timing Contract

The plant moves while software is reading the sensor, computing and updating the output.



Communication links may separate these functions across devices.

- **This chapter asks how to run the controller on time.**

Learning Objectives

What must we know before we can promise a timely motor update?

- Obtain the sampling period, software deadline and execution-time bound.
- Explain what changes from a periodic loop to a schedule table and online selection.
- Trace a release, a scheduling decision and a context switch.
- Explain which timing services and isolation an RTOS configuration provides.
- Model recurring work and individual jobs using Burns notation.

- **Use the same motor controller for each implementation and for the task model.**

Chapter Map: Timing Requirements

5.1 Timing Requirements
From plant timing to firmware scheduling

5.2 Scheduling Implementations
Periodic loop, schedule table and online selection

5.3 Execution Models
Switch points and activation lifecycles

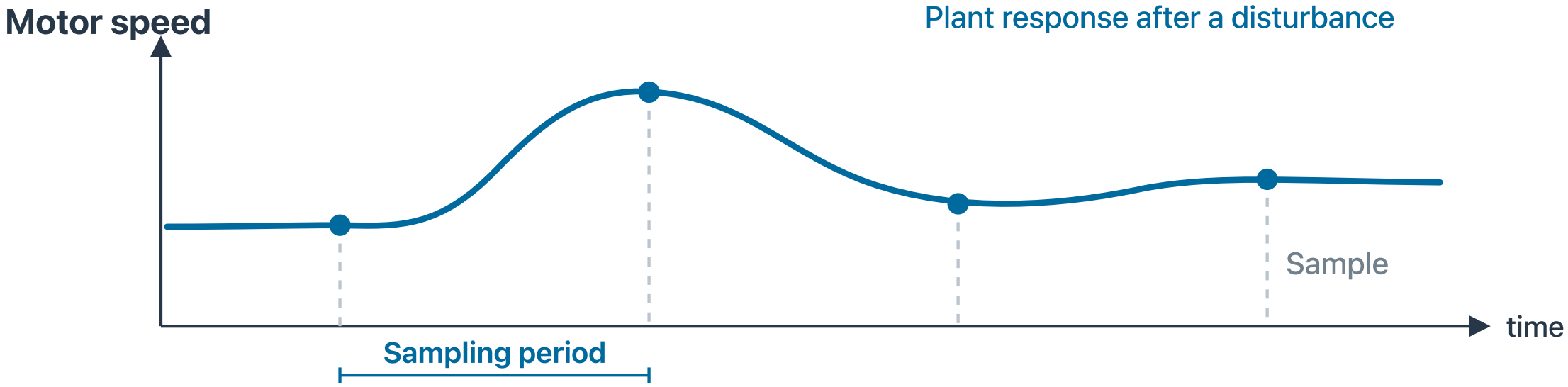
5.4 RTOS Requirements and Architecture
AUTOSAR, kernel designs, protection and OSEK/VDX

5.5 Task Model and Schedulability
Task tuples, jobs and timing guarantees

- **Start with timing requirements that the plant imposes.**

The Plant Sets the Sampling Period

The motor speed changes between observations. The controller operates on sampled values.

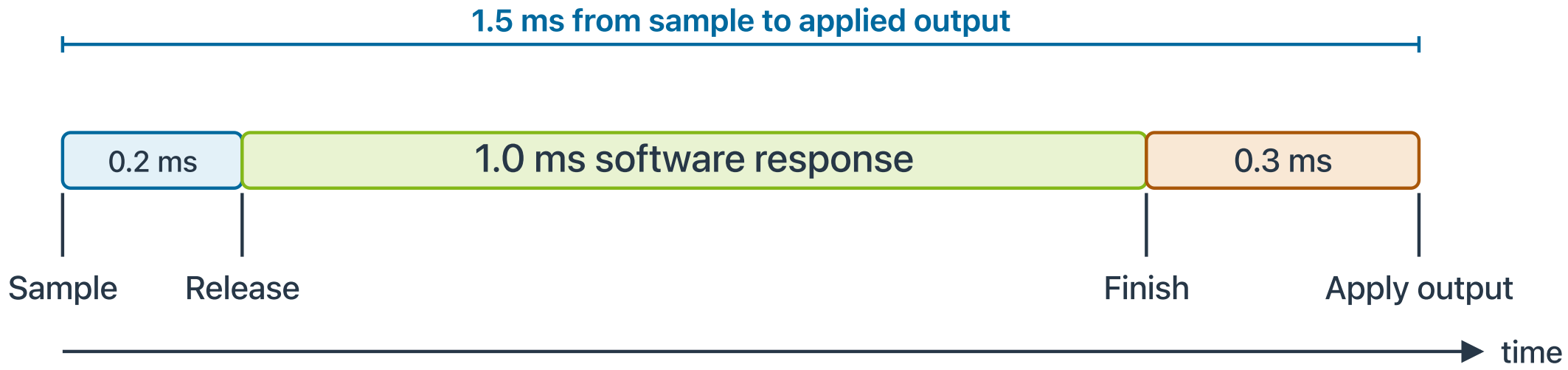


- **The control design selects the sampling period from plant dynamics and the required control quality.**

Teaching example: a controller designed and validated for one observation every 2 ms.

The Control Path Sets the Software Deadline

The application requires each command to take effect at the output within 1.5 ms of sampling.

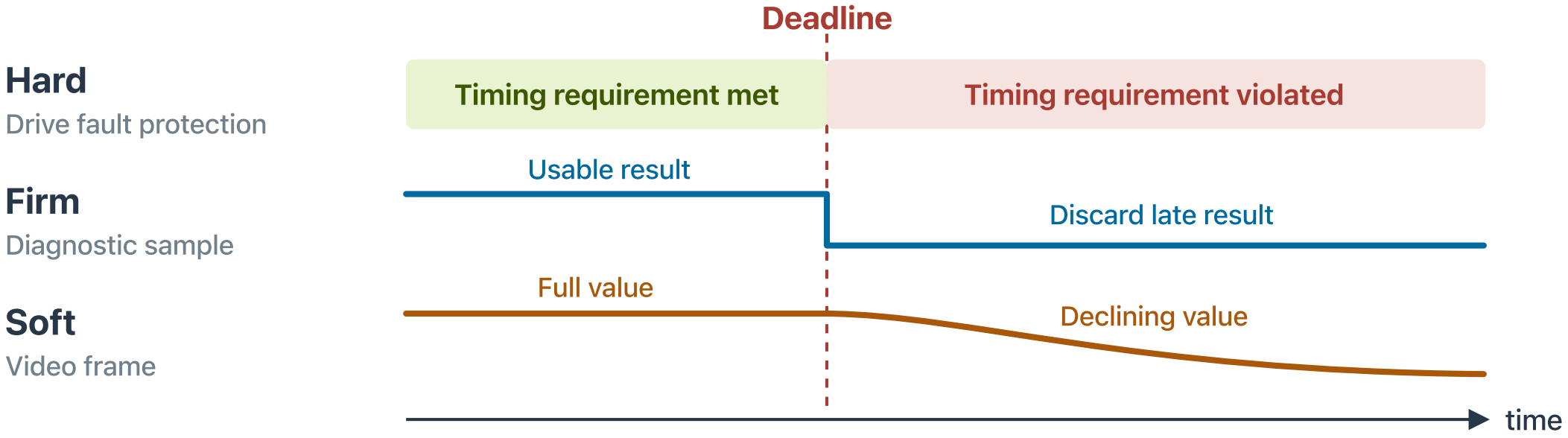


- **Give the control computation a 1.0 ms relative deadline from the time its sample is ready.**

Illustrative allocations for a revised design. The 2 ms sampling period is a separate requirement.

What Happens After a Missed Deadline?

The consequence depends on what the application does with a late result.



- A hard deadline is a required guarantee. With a firm or soft deadline, specify the acceptable loss or degradation.

The Implementation Sets the WCET Bound

The control step reads the published sample, computes the command and publishes it for output. Some inputs take a longer code path.

What must be fixed or bounded?	Motor-control example
Code and inputs	Bounded filter work and a saturation branch
Target hardware	Processor speed and the memory configuration from Chapter 03
Execution-time bound	Assume 0.55 ms of CPU execution per invocation under those conditions

- **The WCET bound is a demand on CPU time. The deadline limits elapsed time, including any wait for the CPU.**

Assumed bound for teaching. Waits and OS overhead are accounted for separately in this example.

Chapter Map: Scheduling Implementations

5.1 Timing Requirements
From plant timing to firmware scheduling

5.2 Scheduling Implementations
Periodic loop, schedule table and online selection

5.3 Execution Models
Switch points and activation lifecycles

5.4 RTOS Requirements and Architecture
AUTOSAR, kernel designs, protection and OSEK/VDX

5.5 Task Model and Schedulability
Task tuples, jobs and timing guarantees

- **Implement the same timing contract with a loop, a table and online selection.**

A Timer-Paced Control Loop

Start with one function on the microcontroller. Every sample triggers one control step.

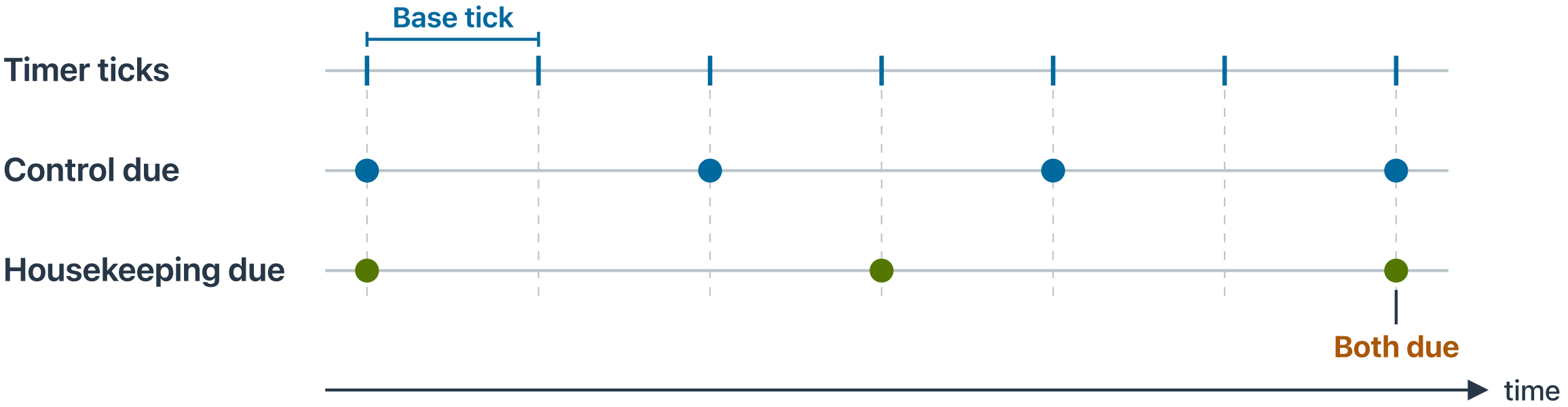
```
next = first_release_time
forever:
    wait_until(next)
    sample = read_published_sample()
    publish_command(control_step(sample))
    next = next + sampling_period
    if now() > next:
        next = apply_overrun_policy(next)
```

- **The timer keeps control releases in phase with sampling, even when the control step takes different amounts of time.**

Pseudocode. The sample is available at the illustrated wake-up; the hardware handles acquisition and the output update.

Several Functions Share One Timer

The control function and housekeeping are due at different intervals. Both can be due on the same tick.



- **Counters determine which functions are due. The loop defines their execution order.**

Release marks show which functions may run. The diagram omits their execution times.

Move the Plan into a Table

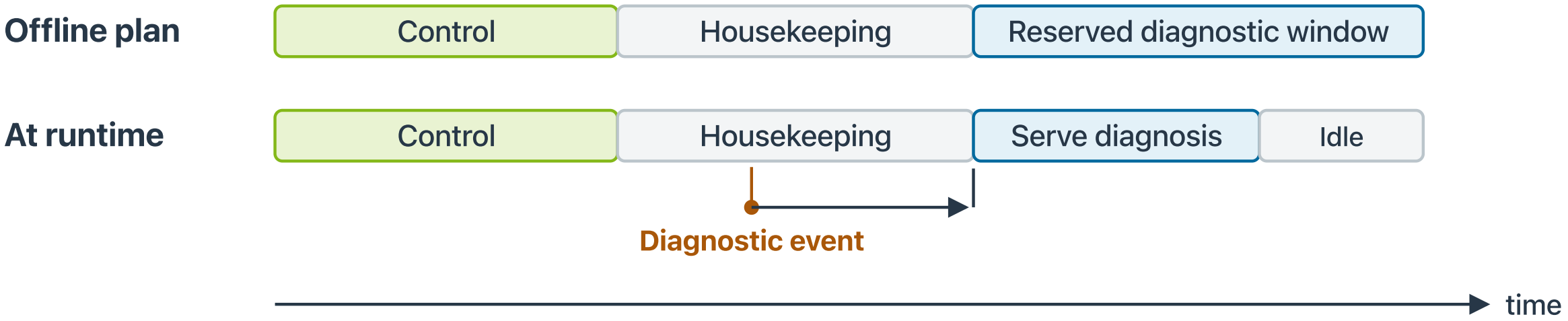
As more functions acquire their own timing constraints, the loop needs more counters and conditions.

Implementation	Where is the execution order specified?
Periodic loop with counters	In the conditions and order of function calls
Table-driven dispatcher	In entries for successive time slots, prepared before execution

- **The dispatcher enacts the table at runtime.**

A Precomputed Dispatch Table

A diagnostic event arrives after the next action has already been fixed in the table.



- **The reservation gives the request a planned service opportunity.**

Schematic example with one pending request and enough time in the reserved window.

When the Ready Work Is Known at Runtime

A diagnostic request can arrive while another function is executing. Each invocation to be scheduled is a **job**.

Approach	Where does the next CPU assignment come from?
Precomputed table	At the planned boundary, dispatch the action assigned to this slot.
Online scheduler	At a scheduling point, select from the work that is ready using the configured rule.

- **With online scheduling, the scheduler makes the concrete choice at runtime.**

What the Firmware Is Already Managing

The microcontroller runs firmware: software stored on the device that controls its hardware and application.

In the implementation so far	What the CPU or system has to do
Periodic loop	Check the time and decide which functions are due.
Schedule table	Track the current slot and start the planned function.
Online scheduling	Record ready work, make a selection and enact it.

- **These are system-software responsibilities. They can live in the application firmware or be provided by a reusable kernel.**

Time spent on bookkeeping, selection or a switch is execution overhead. It is part of the timing budget.

Chapter Map: Execution Models

5.1 Timing Requirements
From plant timing to firmware scheduling

5.2 Scheduling Implementations
Periodic loop, schedule table and online selection

5.3 Execution Models
Switch points and activation lifecycles

5.4 RTOS Requirements and Architecture
AUTOSAR, kernel designs, protection and OSEK/VDX

5.5 Task Model and Schedulability
Task tuples, jobs and timing guarantees

- **Decide when execution can switch and how one invocation is expressed.**

Cooperative and Preemptive Execution

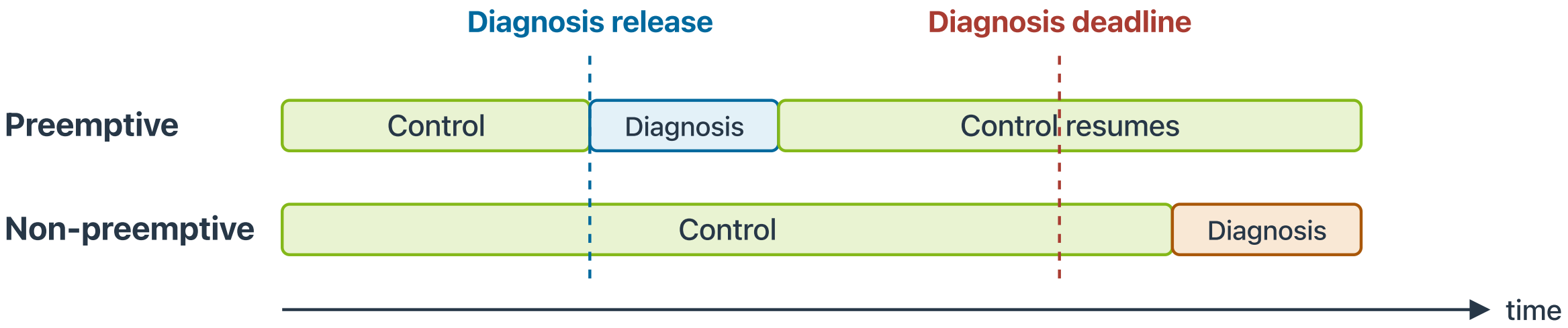
An urgent diagnostic request arrives during a control function.

Execution rule	When can the CPU switch to another function?
Non-preemptive call	At return from the running call.
Cooperative execution	At a defined handover, for example when the running code yields or blocks.
Preemptive execution	At an allowed scheduling point, even if the running code has not offered to hand over.

- **The execution rule determines which switch points are allowed. The scheduling policy selects the work.**

When Can Another Job Take Over?

An urgent diagnostic job becomes ready while the control job is running.



- **Preemption can bring urgent work forward. A cooperative handoff waits for an allowed switch point.**

Same jobs in both traces. Scheduler and context-switch costs are omitted here.

Two Lifecycles for the Same Control Step

The control code needs a sample each time. Where does the waiting happen?

Run-to-completion handler

```
on_control_activation(sample):
    publish_command(control_step(sample))
    return
```

The runtime calls the handler again for the next activation.

Persistent thread

```
control_thread():
    forever:
        sample = wait_for_sample()
        publish_command(control_step(sample))
```

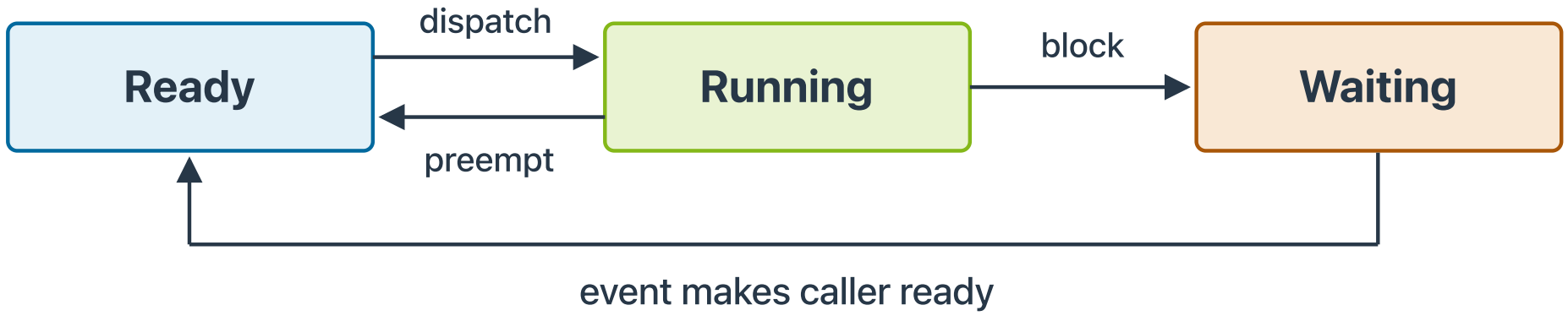
The thread blocks when there is no sample to process.

- **Lifecycle and preemption are separate choices.**

Illustrative pseudocode. The handler executes outside interrupt context.

Ready, Running and Waiting

A control job asks the driver to perform an asynchronous transfer.



- **A blocked caller frees the CPU. A ready caller can compete for it again.**

Four Choices in Scheduling

“Timer-driven”, “online” and “fixed priority” can all describe the same controller.

Choice	Examples
What triggers action?	Timer expiry or device event
When is the CPU allocation decided?	Precomputed table or online selection
When may running work be displaced?	Preemptive, non-preemptive or explicit cooperative switch points
How is eligible work ranked?	Fixed task priority or a job-dependent rule such as earliest absolute deadline

- **Read each term as an answer to a different design question.**

Chapter Map: RTOS Requirements and Architecture

5.1 Timing Requirements
From plant timing to firmware scheduling

5.2 Scheduling Implementations
Periodic loop, schedule table and online selection

5.3 Execution Models
Switch points and activation lifecycles

5.4 RTOS Requirements and Architecture
AUTOSAR, kernel designs, protection and OSEK/VDX

5.5 Task Model and Schedulability
Task tuples, jobs and timing guarantees

- **Connect the RTOS requirements to concrete execution paths and protection.**

What Do We Need from an RTOS?

A reusable kernel takes over timer, event and execution management for the motor controller.

Requirement	What we need from the configured system
Predictable timing	Bounds on interrupt latency and on the kernel paths we use.
Time and scheduling services	A time base, release and wait primitives, and an execution policy that can support the required timing.
Costs that fit the budget	Service and dispatch costs low enough for the application's deadline and load.

- **An RTOS provides mechanisms for meeting timing requirements. The application still needs a timing argument.**

Who Decides, and Who Switches?

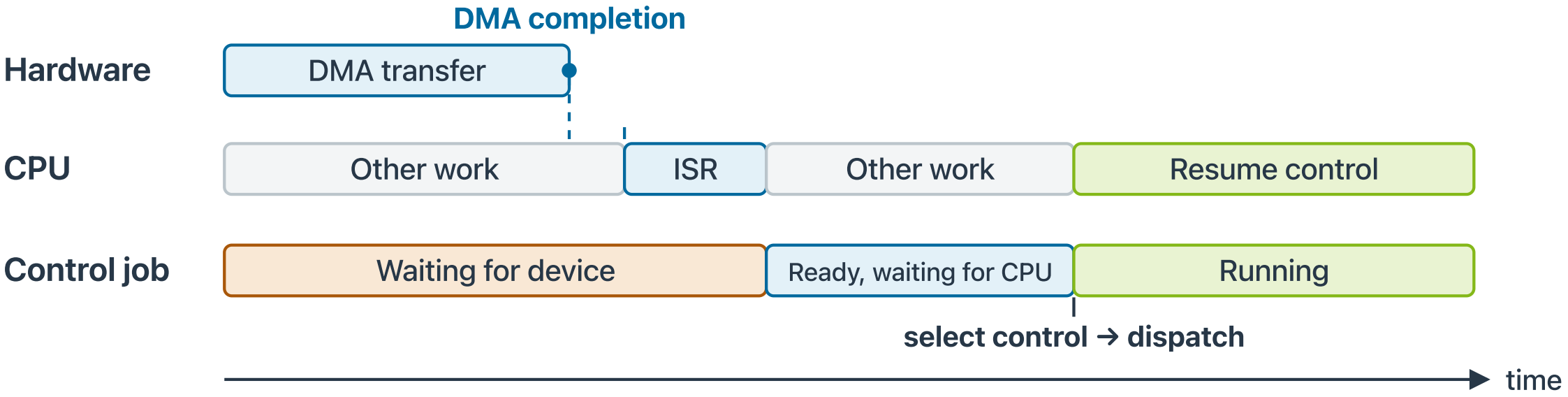
A timer or an interrupt makes another job ready. The current context is still on the CPU.

Responsibility	Decision or action
Release or wake-up	Make the work ready according to the timer or device event.
Scheduler	At a scheduling point, apply the policy to decide which context should run and whether it replaces the current one.
Dispatcher	Enact that decision by saving and restoring context as required and transferring control.

- **The scheduler decides. The dispatcher implements the switch that makes preemption possible.**

From DMA Completion to the Next CPU Assignment

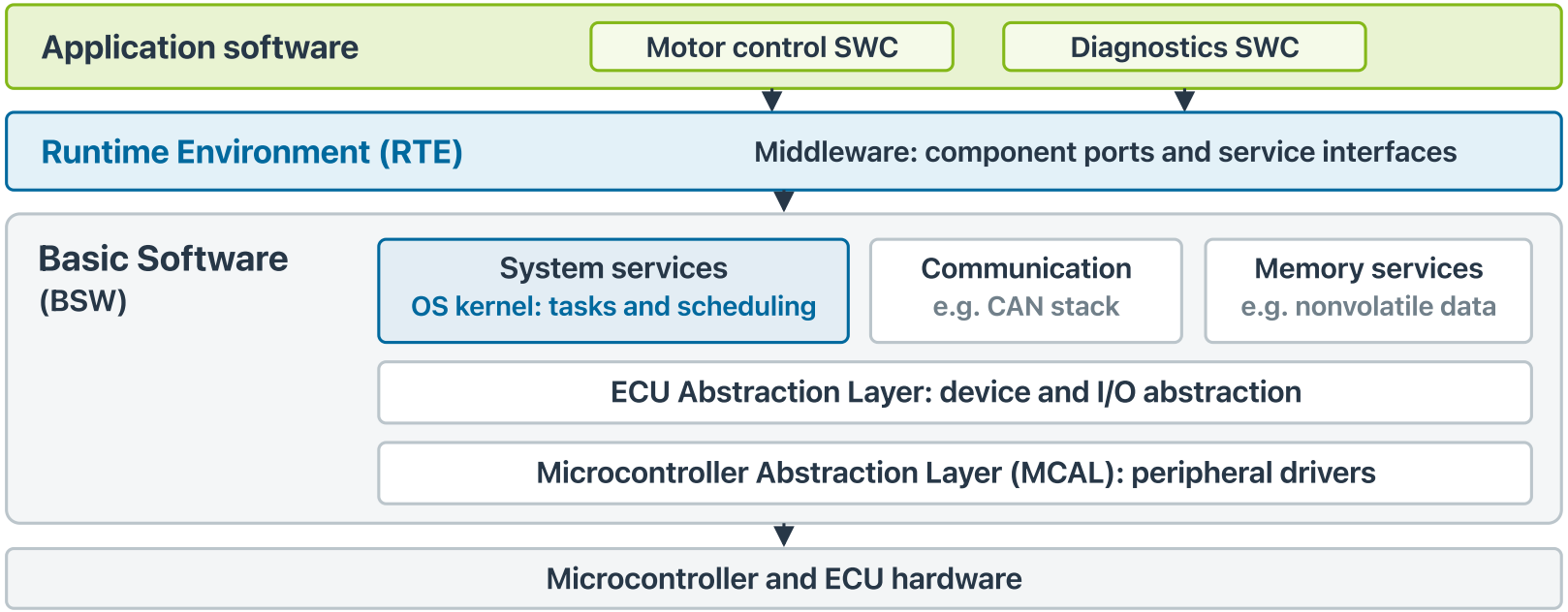
The device finishes a transfer. Another job is still running.



- **The ISR makes the caller ready. The scheduler selects it at a scheduling point. The dispatcher resumes it.**

AUTOSAR Classic Software Layers

Where do motor control, communication stacks and the real-time kernel fit on one electronic control unit (ECU)?

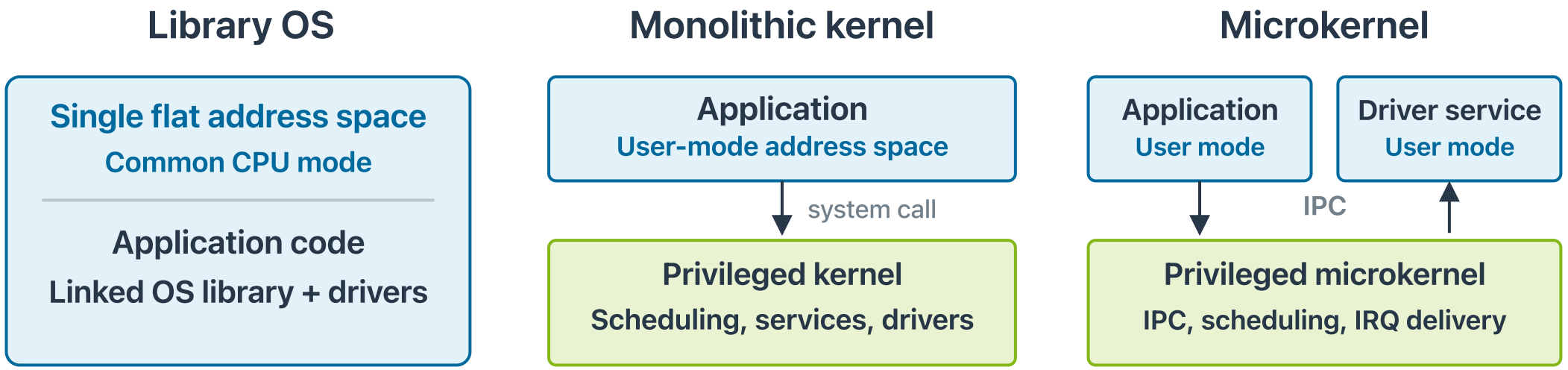


- **The RTE connects components and BSW services. The OS provides execution services within Basic Software.**

Selected layers and examples. Teaching diagram based on AUTOSAR Classic Platform.

Where Do OS Services Run?

Compare three ways to implement the OS services beneath the application interfaces.



Boxes show protection domains. The microkernel example places the driver in its own user address space.

- **Moving a service out of the kernel changes which code executes with privilege and how clients reach it.**

Three example configurations. Microkernel example: seL4 documentation.

Calls, Protection and Fault Scope

A control function requests a device operation. Which boundary does the request cross?

Example	Interface and boundary	If the driver writes to the wrong address
Linked OS library	Direct function call within the same flat domain	The driver can corrupt reachable controller and OS state.
Monolithic kernel	System call into a privileged kernel with in-kernel drivers	A privileged driver fault can affect kernel state and other clients.
Microkernel	Kernel-mediated interprocess communication (IPC) with a user-mode driver	The driver's permissions constrain the write. Clients can still lose the service.

- **The privileged code and granted access rights determine the fault scope. A boundary also changes the execution path.**

Lightweight and Heavyweight Execution Contexts

Control and diagnosis need separate execution contexts. Should they share the same memory and access rights?

Example	What changes on a switch?	What does the boundary protect?
Lightweight: threads in one process	Save and restore registers, program counter and stack pointer.	Threads have their own stacks but share the address space and its permissions.
Heavyweight: separate processes	Switch execution context and, where applicable, address-space or protection state.	Each process has its own access domain. Sharing requires an explicit mapping or IPC.

- **An execution context holds where the code resumes. A protection domain controls which memory and devices it can reach.**

"Lightweight" and "heavyweight" describe the two illustrative cases here.

Memory Management and Stack Protection

A diagnostic thread needs a stack, private state and a controlled way to share samples.

Mechanism	What it does	Example
Memory management	Allocates storage and configures access rights. An MPU checks regions; an MMU can also translate addresses.	Diagnosis can read a shared sample buffer and write its own state.
Stack guard	Denies access to a guard region or page so a crossing access faults.	A stack write into the guard triggers a protection fault.
Stack monitoring	Checks stack usage or a sentinel at defined points.	A context-switch check can detect an overrun after it has occurred.

- **Distinguish enforcing an access boundary from checking whether a stack stayed within its budget.**

MPU: Memory Protection Unit. MMU: Memory Management Unit. See AUTOSAR OS R24-11, §§7.5 and 7.7.

Timing Protection in AUTOSAR OS

A memory-isolated diagnostic component can still consume CPU time, hold a lock or trigger too often.

Configured limit	What the OS monitors	Motor example
Execution Budget	CPU time used by a task or Category 2 ISR	Diagnosis exceeds its allowed execution.
Lock Budget	Time holding a resource or keeping interrupts suspended	Diagnosis delays the next control release or dispatch.
Time Frame	Minimum spacing of monitored task activations/releases or Category 2 ISR arrivals	A diagnostic source triggers too frequently.

- **Timing protection enforces assumptions at runtime. Schedulability analysis checks whether those assumptions suffice for the deadlines.**

Availability and handling depend on configuration. AUTOSAR OS R24-11, §§7.7.2, 7.8 and 7.11.

The Same Completion Takes Different Paths

The controller waits for a DMA completion. Compare what the configured architecture has to do.

Illustrative design	Completion path	What enters the timing argument?
Linked services	Interrupt handler publishes completion to the linked control code.	Interrupt delay, handler work and the next allowed dispatch.
Monolithic kernel	Kernel driver signals a waiting application context.	Kernel wake-up path and any context or protection-domain switch.
Microkernel with driver service	IRQ notification reaches the service, which signals the controller via IPC.	Driver-service scheduling, message delivery and dispatch across domains.

- **Compare concrete paths and their bounds. Architecture labels alone give no runtime cost.**

Account for the Time Used by the Runtime

The control computation can consume up to 0.55 ms of its 1.0 ms software deadline.

Where time is spent	What the timing argument must cover
Before the software release	Interrupt delivery and notification if they lie in this part of the end-to-end chain
From release to finish	Time when other work runs, kernel paths, and dispatch costs attributable to the interval
Inside the control computation	Execution-time bound for the code, with memory and cache assumptions stated

- **A bound is useful only if its interval and assumptions match the budget.**

In this example, the CPU computation leaves 0.45 ms for everything else between release and finish in its worst execution case.

OSEK/VDX Alarm and Task Activation

How can the drive configure a periodic control activation at build time?

Configured object	Selected properties	Runtime effect
Counter	Driven by the system time base	Provides ticks for the alarm
Cyclic alarm	Action: activate `Control`	Requests a task activation on expiry
Basic task `Control`	Fixed priority, `SCHEDULE = FULL`	Eligible for preemptive dispatch, terminates after the control step

- **The configuration specifies what becomes ready; the scheduler still chooses what runs.**

OSEK/VDX OS 2.2.3, §§4.2, 4.5–4.6 and 9. Selected settings for an illustrative configuration.

OSEK/VDX Basic and Extended Tasks

The periodic control step can finish after each activation. A diagnostic task may wait for a device event.

	Basic task	Extended task
Activation	Becomes ready	Becomes ready
Event wait	No Waiting state	`WaitEvent` may enter Waiting
End of activation	`TerminateTask`	`TerminateTask`
Preemption	Depends on scheduling configuration	Depends on scheduling configuration

- **Task type controls event waiting; scheduling configuration controls task preemption.**

A Shared Buffer Can Delay a Job

The controller and the diagnostic thread access the same published sample.

1. The diagnostic thread holds the mutex protecting the sample.
2. The controller becomes ready and is dispatched.
3. Its attempt to acquire the mutex waits until the resource is available.

- **A ready job can start on time and then block on a resource.**

Chapter Map: Task Model and Schedulability

5.1 Timing Requirements
From plant timing to firmware scheduling

5.2 Scheduling Implementations
Periodic loop, schedule table and online selection

5.3 Execution Models
Switch points and activation lifecycles

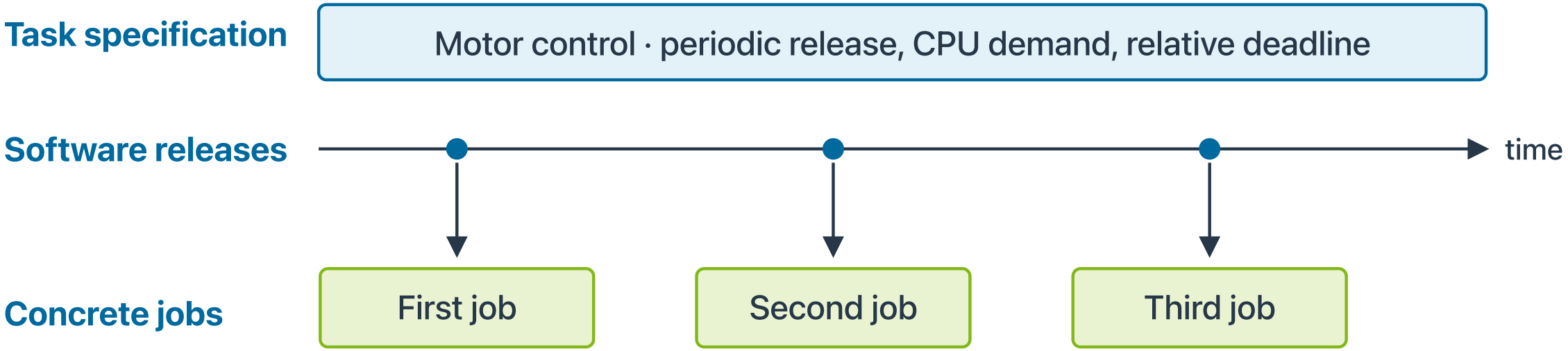
5.4 RTOS Requirements and Architecture
AUTOSAR, kernel designs, protection and OSEK/VDX

5.5 Task Model and Schedulability
Task tuples, jobs and timing guarantees

- **Describe the workload and ask what guarantees can be proved.**

One Task, Successive Jobs

The design defines one recurring control task. Each timer release requests one concrete invocation.



- A task is the recurring specification. A job is the instance the scheduler plans.

The Periodic Task Tuple

The three timing quantities from the motor example now become model parameters.

$$\tau_i = (T_i, C_i, D_i, \phi_i)$$

Symbol	Meaning	Control example, $i = 1$
τ_i	Task with index i	Control computation
T_i	Period between nominal releases	$2\,\mathrm{ms}$
C_i	Upper bound on execution time per job	$0.55\,\mathrm{ms}$
D_i	Relative deadline from release	$1.0\,\mathrm{ms}$
ϕ_i	Nominal first release on the chosen time axis	$0\,\mathrm{ms}$ on a release-aligned axis

- **The tuple describes the task as a whole; each release gives an instance its own time coordinates.**

ϕ_i is a course convention for the phase.

From the Task Tuple to an Individual Job

$J_{i,k}$ is job k of task τ_i. The job index $k = 1, 2, \ldots$ counts releases of that task.

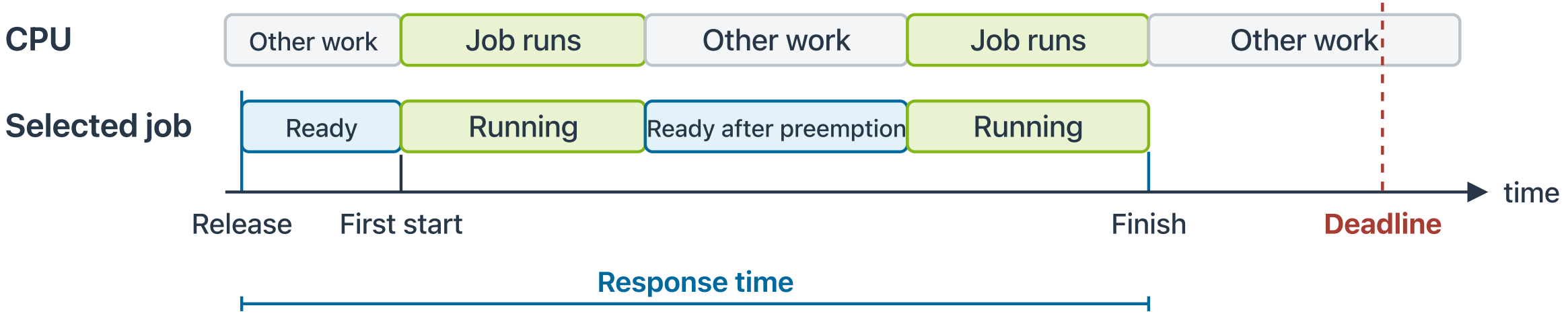
Job property	In the periodic model without release jitter
Release $r_{i,k}$	$\phi_i + (k-1)T_i$
CPU demand bound	Each job may require up to C_i of CPU time.
Absolute deadline $d_{i,k}$	$r_{i,k} + D_i$
Actual finish $f_{i,k}$	Determined by the execution and schedule.

- **The release and the relative deadline give the job a specific deadline on the time axis.**

For the control task, the second job becomes ready at 2 ms and must finish by 3 ms.

Execution Time and Response Time

Job k of task i can wait before its first dispatch and again after preemption.

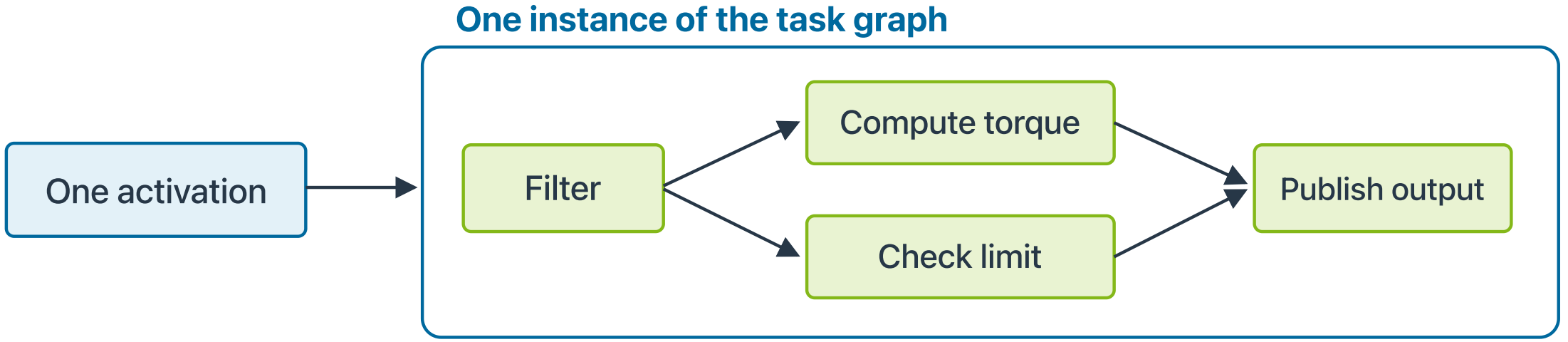


Execution time adds the two running intervals. **Response time** is $f_{i,k} - r_{i,k}$.

- **The job meets its deadline if $f_{i,k} \leq d_{i,k}$. The task's worst-case response time is R_i.**

When One Activation Has Several Dependent Steps

A sample may require filtering before torque calculation and limit checking can use the result.

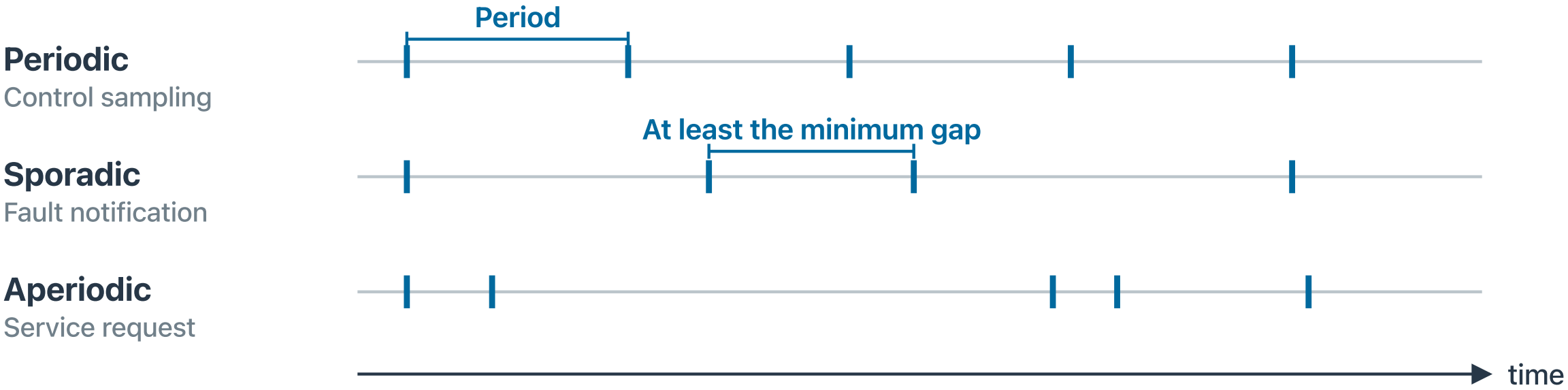


- **A directed acyclic graph (DAG) models these dependencies explicitly. Its successors become eligible after their prerequisites.**

Extension of the model. The first scheduling analyses use one independent sequential job per release.

Periodic, Sporadic and Aperiodic Work

The controller must describe how often each kind of work can arrive.



- **The arrival model determines which workloads the timing claim covers.**

Sporadic Tasks and Aperiodic Jobs

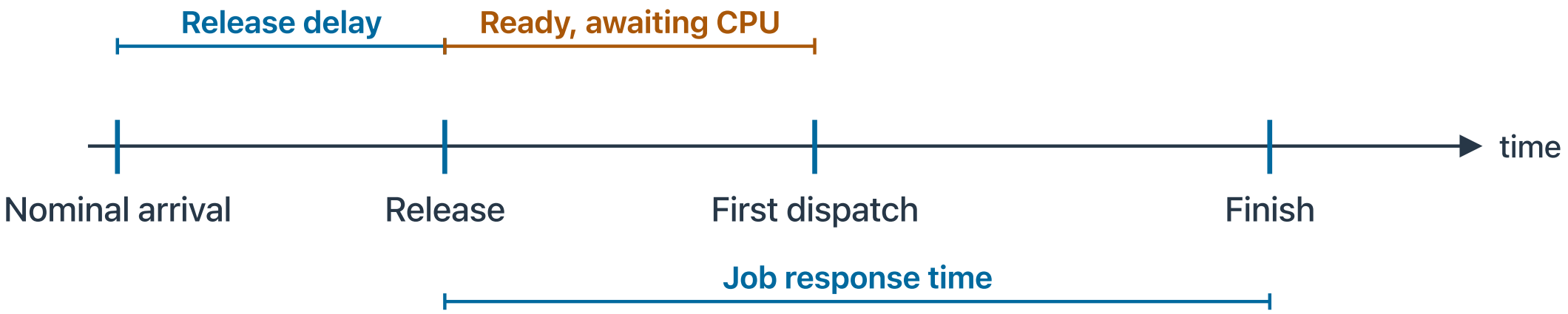
The control loop has a period. For other work, the arrival guarantee changes.

Workload	Model in this course
Periodic	$\tau_i = (T_i, C_i, D_i, \phi_i)$. The period gives nominal release times.
Sporadic	$\tau_i = (T_i, C_i, D_i)$. Here T_i gives a minimum inter-arrival time.
Aperiodic	The workload has no guaranteed minimum inter-arrival time in this model. For a given set of jobs, specify individual releases, demands and deadlines.

- **The same T_i symbol describes a different arrival guarantee in the periodic and sporadic models.**

Arrival, Release and First Dispatch

A physical event can occur before the software is ready to process it.



- **J_i bounds release jitter. Waiting for the CPU after release contributes to response time.**

EZS Cheat Sheet · Tasks and Jobs

i identifies a task. $k = 1, 2, \ldots$ counts its successive jobs.

Task notation	Meaning	Job notation	Meaning
τ_i	Task i	$J_{i,k}$	Job k of task i
T_i	Period or minimum inter-arrival time	$r_{i,k}$	Release time
C_i	Execution-time bound per job	$d_{i,k} = r_{i,k} + D_i$	Absolute deadline
D_i	Relative deadline	$f_{i,k}$	Finish time
ϕ_i	Phase of a periodic task	$f_{i,k} - r_{i,k}$	Actual response time
J_i	Release jitter bound	$f_{i,k} \leq d_{i,k}$	Deadline met
R_i	Worst-case response time		

Periodic $\tau_i = (T_i, C_i, D_i, \phi_i)$ · **Sporadic** $\tau_i = (T_i, C_i, D_i)$

Burns Standard Notation (Davis, 2013). Course conventions ϕ_i, $J_{i,k}$, $r_{i,k}$ and $f_{i,k}$.

The First Scheduling Model

The complete drive includes device waits, shared buffers and an actuator. Which part can we analyse first?

Assumption for the initial analysis	Implication
One CPU	At most one job executes at a time.
Independent CPU jobs	There are no precedence or shared-lock constraints in this model.
No self-suspension	Once released, a job stays eligible until it finishes.
Specified workload and costs	Arrival assumptions, execution bounds and deadlines delimit the claim.

- **Analyse the CPU workload under explicit assumptions.**

From One Schedule to a Guarantee

A single trace shows that both jobs met their deadlines in one situation.

Question	Scope
Feasibility	Does a schedule satisfying all constraints exist for this job set?
Schedulability under a policy	Will the specified policy meet the deadlines for every workload allowed by the model?
Evidence	Use a test whose assumptions match the claim.

- **A trace can demonstrate a miss. A guarantee must cover all admitted cases.**

Reading a Schedulability Test

A test condition is evaluated for a specified workload model and policy.

Test condition	If it holds	If it fails
Necessary	The result may remain open.	The guarantee fails.
Sufficient	The guarantee holds.	The result may remain open.
Exact	The guarantee holds.	The guarantee fails.

- **Check the direction of implication before using a test result.**

Self Check

The device completes a transfer while another job runs. A diagnostic thread is also ready.

1. Which transition is caused by the ISR, and when does the controller use the CPU again?
2. Which parts of the trace count toward execution time and which toward response time?
3. Which workload and policy assumptions would you need to justify a hard deadline?

- **Identify the release or wake-up, dispatch and finish before making a timing claim.**

Scheduling Problems Ahead

The first analysis considers independent jobs on one CPU.

Next topic	Question
Aperiodic jobs	Can the given jobs meet their deadlines?
Periodic tasks and EDF	How do recurring arrivals affect deadline guarantees?
Fixed-priority scheduling	How much delay can higher-priority work impose?
Shared resources and blocking	How much delay can a resource holder impose?

- **The task model carries across these questions as we add new assumptions.**

References

Reading in Marwedel, 4th edition

- Embedded and real-time operating systems → §§4.1.1–4.1.2
- Scheduling problem and assumptions → §§6.1.1–6.1.2
- Follow-up reading → §§6.2.1 and 6.2.3 (one CPU), §4.2 (resource protocols)

Bibliography

- P. Marwedel. *Embedded System Design*, 4th ed., 2021. Springer
- R. I. Davis. *Burns Standard Notation for real time scheduling*, 2013. Author's copy
- P. Ulbrich. *Echtzeitsysteme (EZS)*, FAU Erlangen-Nürnberg, winter semester 2019/20. Lecture slides
- OSEK/VDX. *Operating System Specification 2.2.3*, 2005. Specification
- AUTOSAR. *Classic Platform* and *Specification of Operating System*, R24-11. Architecture · OS specification
- seL4 Foundation. The seL4 microkernel · Interprocess communication