

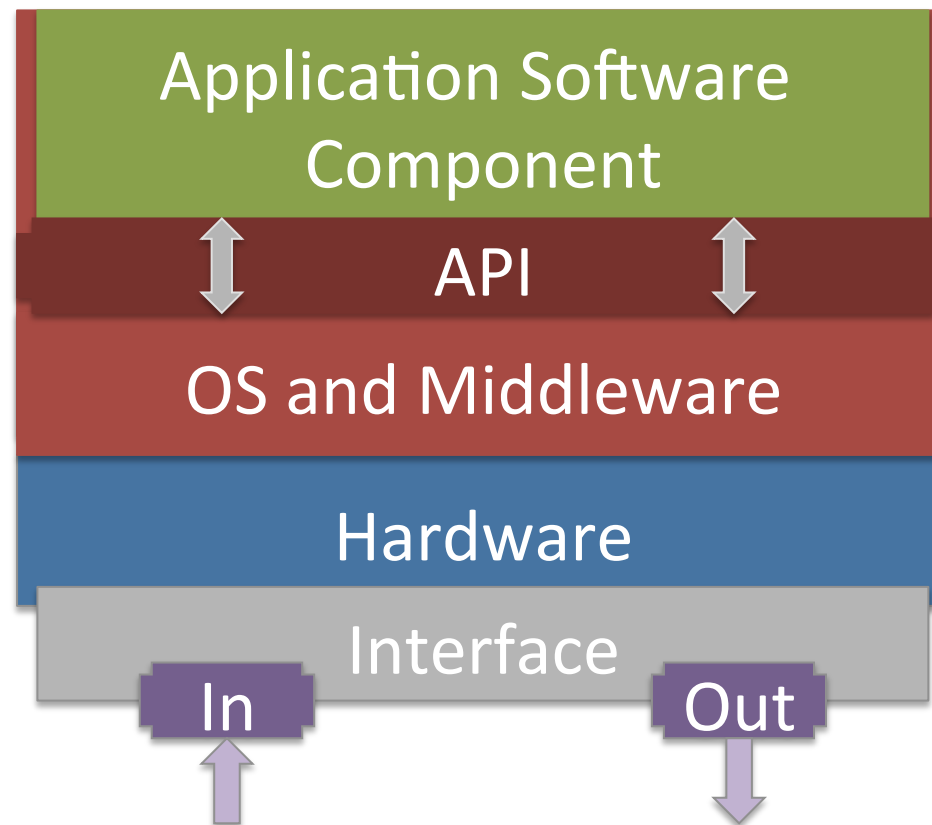
Real-Time Component Software

slide credits: H. Kopetz, P. Puschner

Overview

- OS services
- Task Structure
- Task Interaction
- Input/Output
- Error Detection

Operating System and Middleware



OS Services

- Secure boot of component software
- Reset, start, exec. control of component SW
- Task management
- Task interaction
- Message communication interface
 - Linking Interface (RTS)
 - Config., control (TII), debug (TDI)
- I/O Handling
 - Discretization
 - Agreement

Time predictability, determinism

Assumptions about Software

HRT Software

- Closed world assumption
 - Tasks and task timing parameters known at design time
 - Task communication, precedence known at design time
 - I/O requirements known (values, timing)
- ➡ Pre-runtime preparation/analysis to provide runtime guarantees

SRT Software

- Open world assumption
 - Tasks, task timing and QoS parameters, I/O requirements
- ➡ QoS assessment before runtime; at runtime: best effort

Non real-time Software

Task Management

- The software of a component is structured into a set of tasks (execution of a sequential program) that run in parallel.
- The OS provides the execution environment for each task.
- Temporal and spatial isolation: HRT Software versus other SW
- HRT Tasks are cooperative, not competitive.
- Component = unit of failure
 - No resource-intensive protection between tasks
 - Light-weight OS
- Stateless versus stateful tasks

S-Tasks and C-Tasks

Simple task (S-Task)

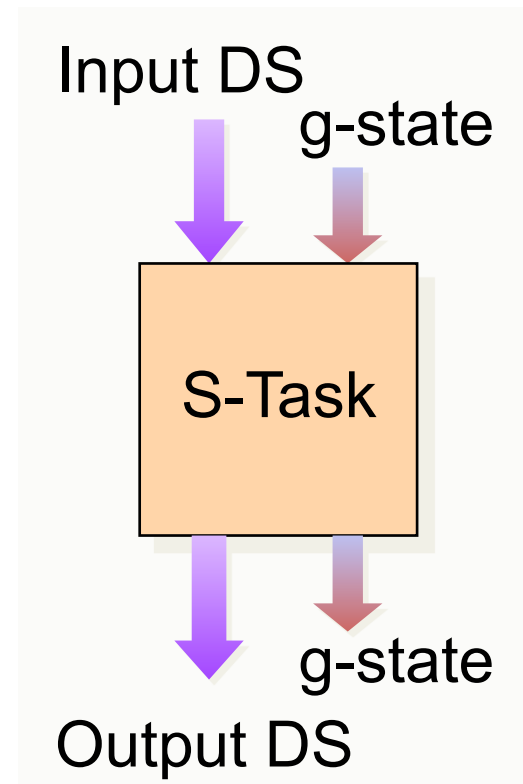
- executes from the beginning to the end without any delay, given the CPU has been allocated.

Complex task (C-Task)

- may contain one or more *WAIT* statements in the task body.

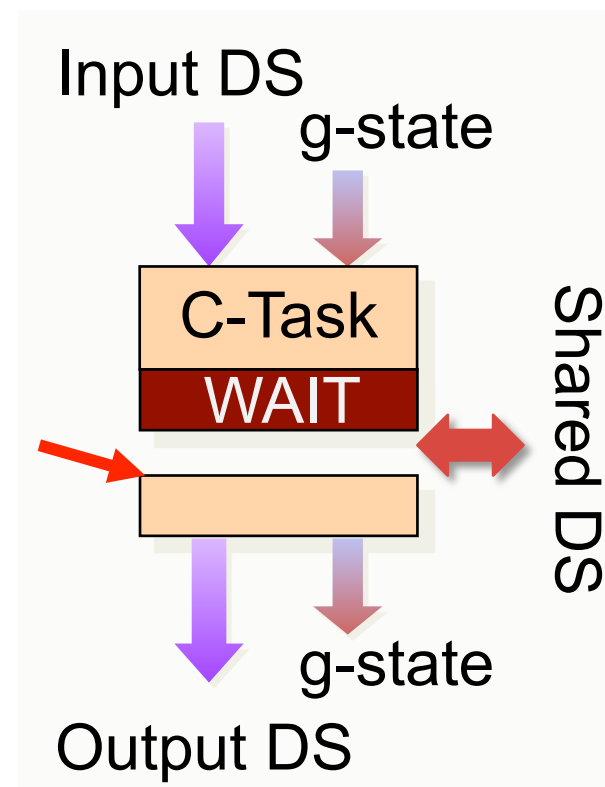
Simple Task (S-Task)

- Can execute from beginning to end without delay, given the CPU has been allocated to it
- No blocking inside (no synchronization, communication)
- Independent progress
- Inputs available in input data structure at the task start
- Outputs ready in the output data structure at the task end
- API: input DS, output DS, g-state



Complex Task (C-Task)

- May contain one or more WAIT operations
- Possible dependencies due to synchronization, communication
- Progress dependent on other tasks in node or environment
- C-task timing is a global issue
- API: input DS, output DS, g-state, shared DS



ARINC Standard WG 48-1999

HRT systems demands (taken from ARINC standard):

*The Avionics Computing Resource (ACR) shall include internal hardware and software management methods as necessary to ensure that **time, space and I/O allocations are deterministic and static.***

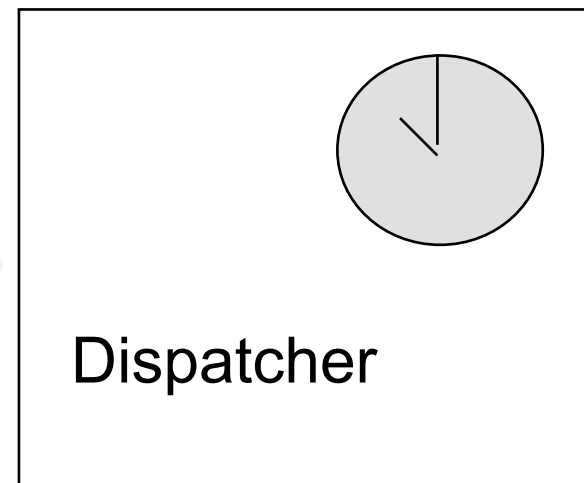
*“Deterministic and static” means in this context, that time, space and I/O allocations are **determined at compilation, assembly or link time**, remain identical at each and every initialization of a program or process, and are not dynamically altered during runtime.*

Time-Triggered Task Control

In strictly TT systems, the dispatcher controls the execution of tasks, by interpreting the Task Descriptor List (TADL).

TADL

Time	Action
10	Start T1
17	Send M5
22	Stop T1
38	Start T3
47	Send M3



The TADL tables are generated and checked by a static scheduler, before runtime.

TT Resource Management

In a TT OS there is hardly any dynamic resource management.

- Static CPU allocation.
- Autonomous memory management. It needs little attention from the operating system.
- Buffer management is minimal. No queues.
- Implicit, pre-planned synchronization fulfills synchronization needs and precedence constraints → S-tasks only
- No explicit synchronization (e.g., mgmt. of semaphore queues).

Operating systems become simple, can be formally analyzed.

Examples: TTOS, OSEK time

TT Task Structure

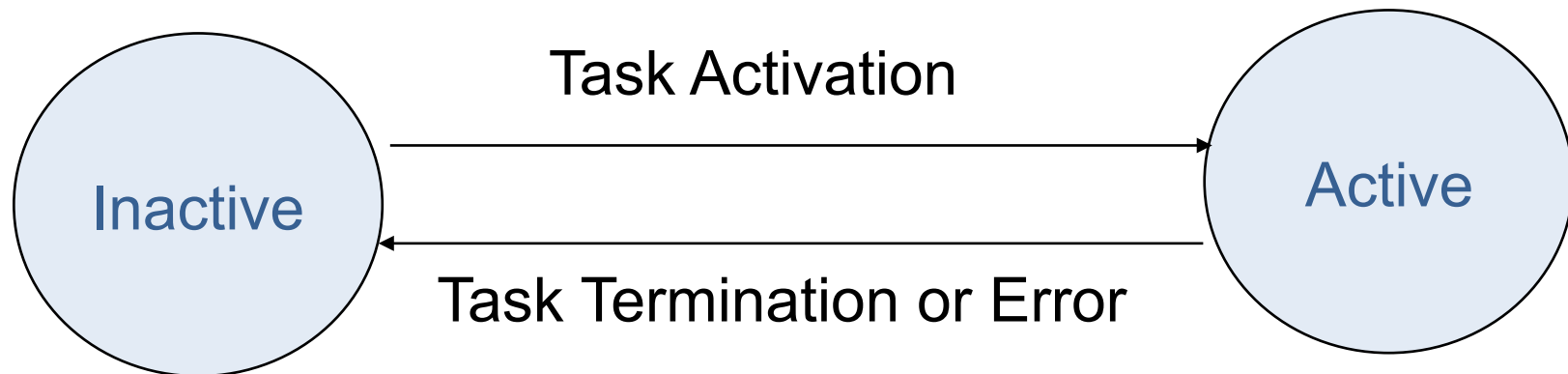
Basically the task structure in a TT system is static. There are some techniques that make the task structure data dependent, but they are limited:

- Mode changes – navigate dynamically between statically validated operating modes.
- Sporadic server tasks: Provide a laxity in the schedule that can be consumed by a sporadic server task.
- Precedence graphs with exclusive or: dynamic selection of one of a number of mutually exclusive alternatives (not very effective!)

This limited data dependency of the task structure has both big advantages and disadvantages.

TT Task States

Non preemptive system



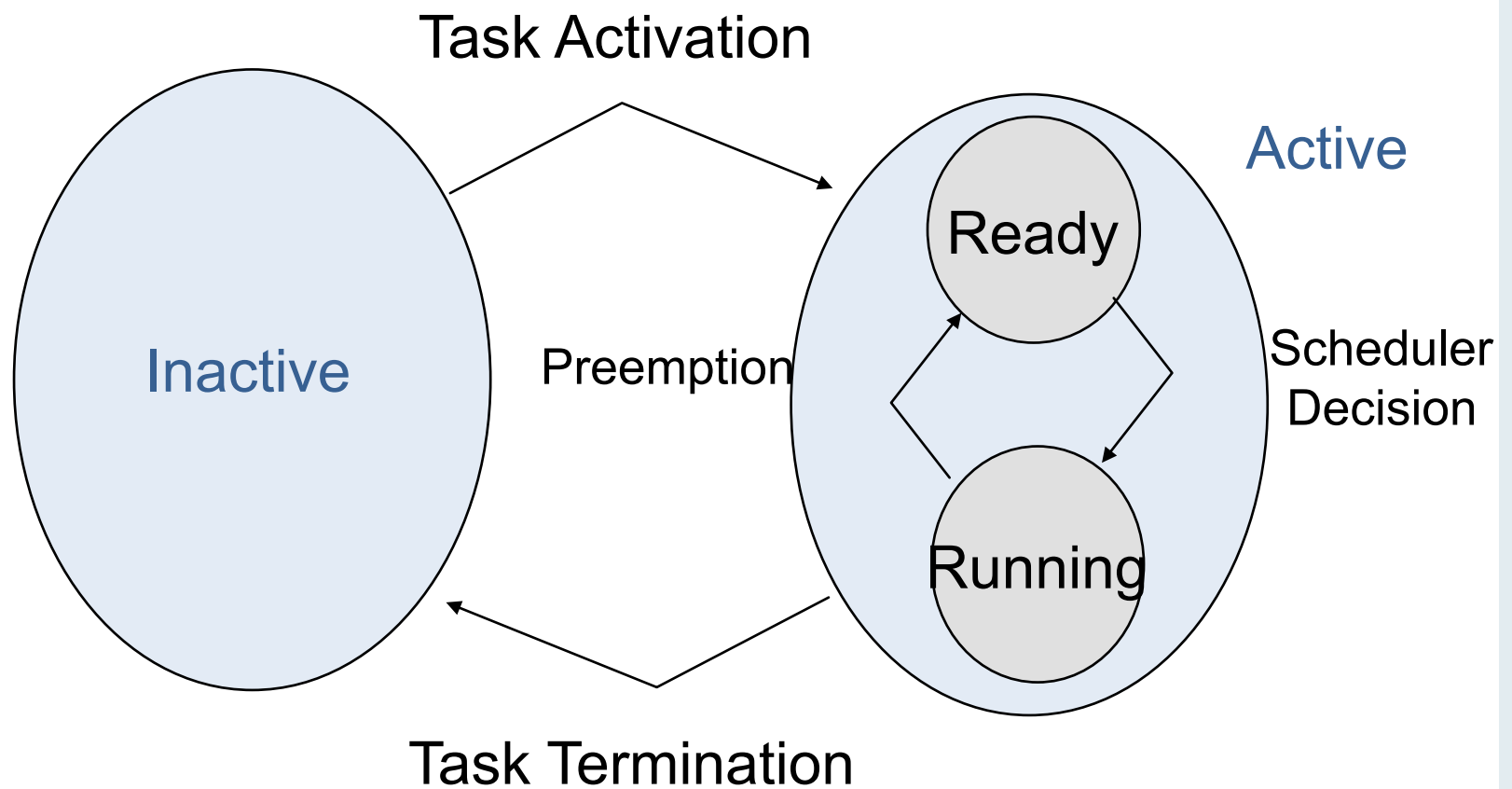
Task Control – ET with S-Tasks

In an ET system, the task control is performed by a dynamic scheduler that decides which task has to be executed next on the basis of the evolving request scenario.

- Advantage:
Actual (and not maximum) load and task execution times form the basis of the scheduling decisions.
- Disadvantage:
In most realistic cases the scheduling problem that has to be solved on-line is NP hard.

ET Task States with S-Tasks

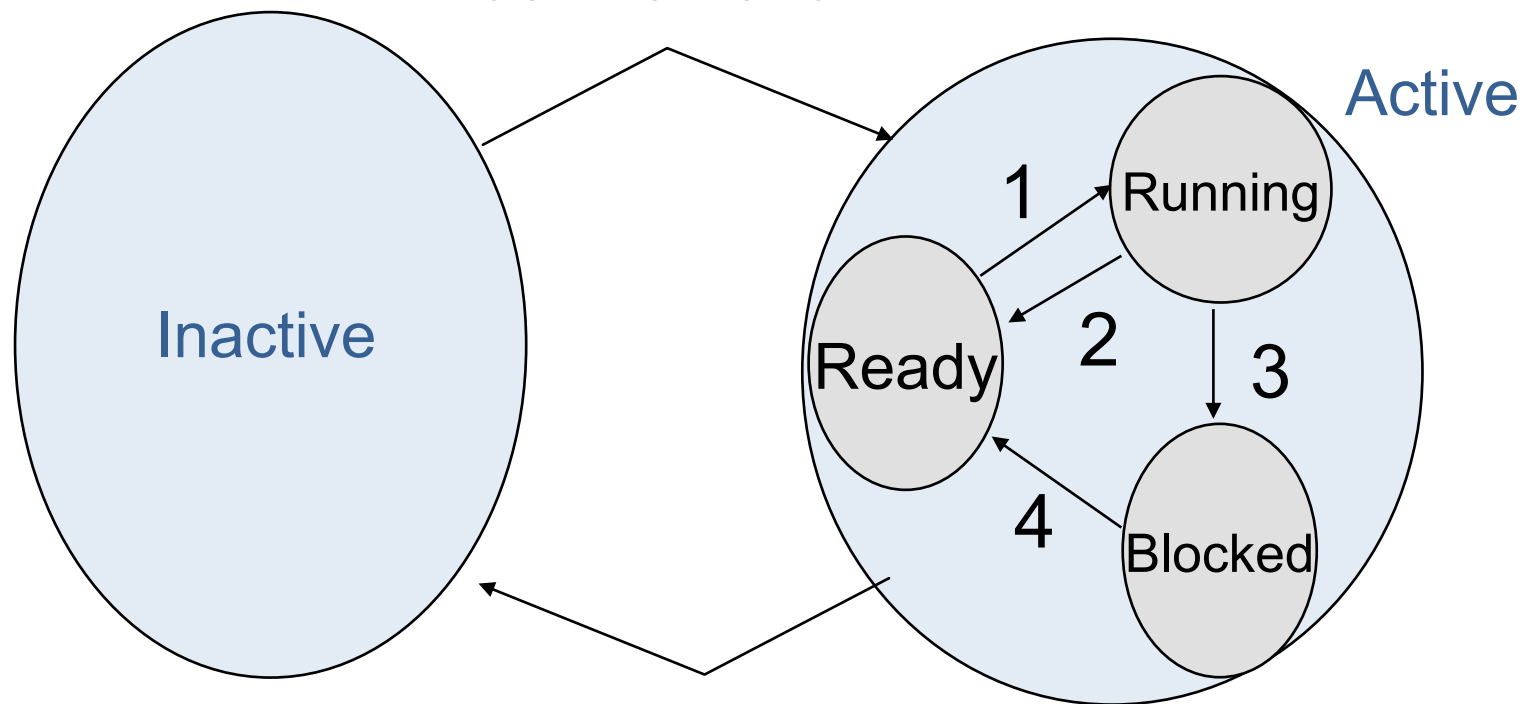
Preemptive system



ET Task States with C-Tasks

Preemptive system

Task Activation



- 1 Scheduler Decision
- 2 Task Preemption

- 3 Task executes WAIT for Event
- 4 Blocking Event occurs

ET Resource Management

In ET OS the dynamic resource management is extensive:

- Dynamic CPU allocation.
- Dynamic memory management.
- Dynamic Buffer allocation and ET management of communication activities
- Explicit synchronization between tasks, including semaphore queue management and deadlock detection.
- Extensive interrupt management.
- Timeout handling of blocked tasks.

A formal timing analysis of ET operating systems is beyond the state of the art (e.g., OSEK).

Task Interaction

Precedence constraints: restrictions on task sequence (e.g., sequence of actions or outputs)

- TT: reflected in TT schedule (TADL)
- ET: WAIT

Exchange of data

- Messages
- Shared data structure \Rightarrow provision of integrity
 - Coordinated task schedules
TT schedule guarantees mutex: deterministic solution, min. overhead
 - Non-blocking write protocol
 - Semaphores

Non-Blocking Write (NBW) Protocol

Assumptions

- “Distributed” System
- Communication via shared memory
- Exactly one writer on dedicated CPU (no conflicts on CPU)
- One or more readers (on one or more CPUs)
- Intervals between write operations are long compared to the duration of a write operation

Non-Blocking Write Protocol

Init: `CCF := 0; /* concurrency control flag */`

Writer:

```
CCF_old := CCF;
CCF := CCF_old + 1;
write to shared struct;
CCF := CCF_old + 2;
```

Reader:

```
start: CCF_begin := CCF;
      if CCF_begin mod 2 = 1
      then goto start;
      read from shared struct;
      CCF_end := CCF;
      if CCF_end  $\neq$  CCF_begin
      then goto start;
```

CCF arithmetics in practice: all CCF operations mod (2 * bigN)

Non-Blocking Write Protocol

Demanded Properties







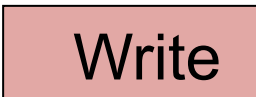



Consistency: Read operations must return consistent results.

Non-Blocking Property: Readers must not block the writer.

Timeliness: The maximum delay of a reader during a read operation must be bounded.

NBW Protocol Correctness

The following szenarios have to be checked for correctness:

1.  
2.  
3.  
4.  
5. 
6. 

Task Interaction

Semaphores: high overheads for small critical regions (typical for real-time applications)

Replica determinism

- Simultaneous access to CCF (NBW) or semaphores may cause race conditions
 - ⇒ unpredictable resolution

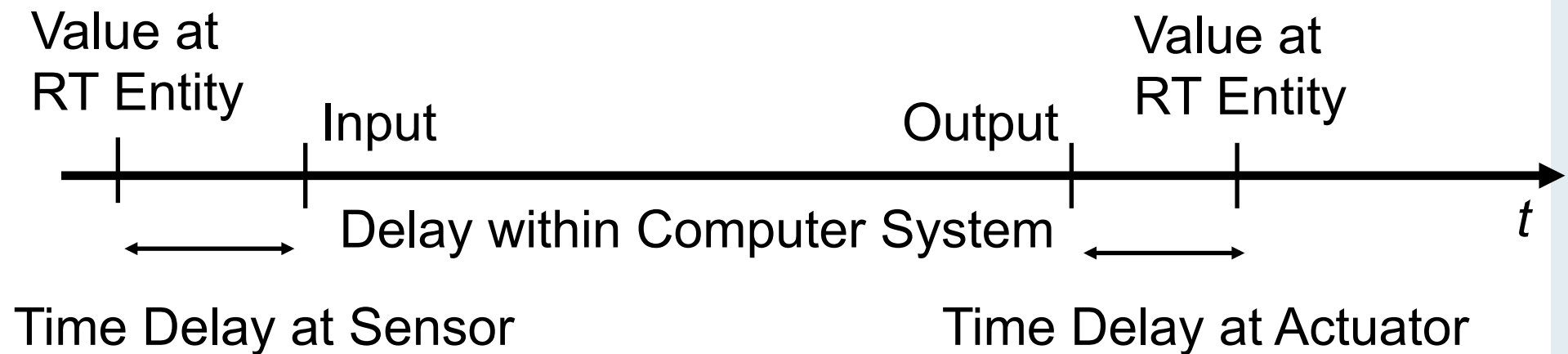
Time Services

Clock Synchronization

Time services:

- Specification of a potentially infinite sequence of events at absolute time-points (off-line and on-line).
- Specification of a future point in time within a specified temporal distance from "now" (timeout service)
- Time stamping of events immediately after their occurrence
- Output of a message (or a control signal) at a precisely defined point in time in the future, either relative to "now" or at an absolute future time point
- Gregorian calendar function to convert TAI (UTC) to calendar time and vice versa

Timing at I/O Interface



Phase-alignment of “sampling – transmission to control node – computation – transmission of set point to the actuator” to reduce dead time of control loops

Dual Role of Time

A significant event in the environment of a real-time computer can be seen from two different perspectives:

- **Time as data:** Point in time of a value change of an RT entity. Precise knowledge of this point in time is important for the analysis of the consequences of the event.

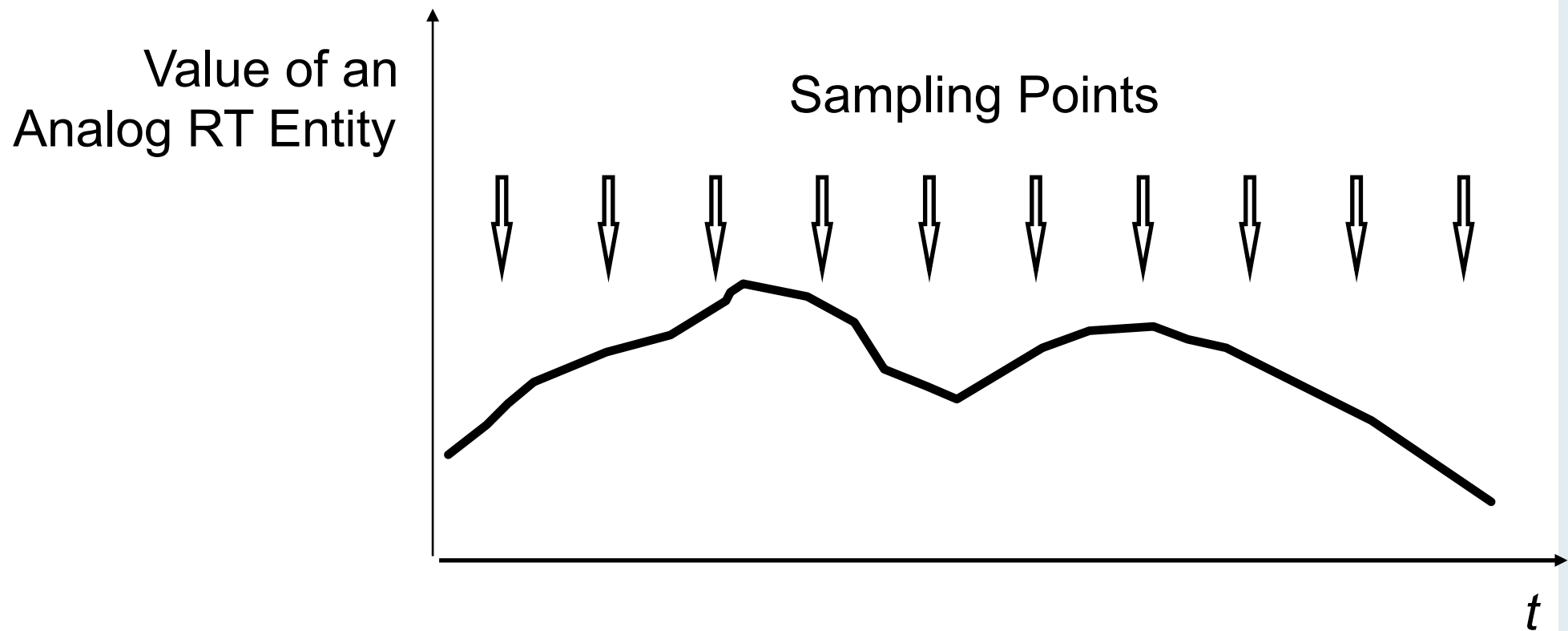
Example: timekeeping in downhill skiing

- **Time as control:** may demand immediate action by the computer system to react as soon as possible to this event.

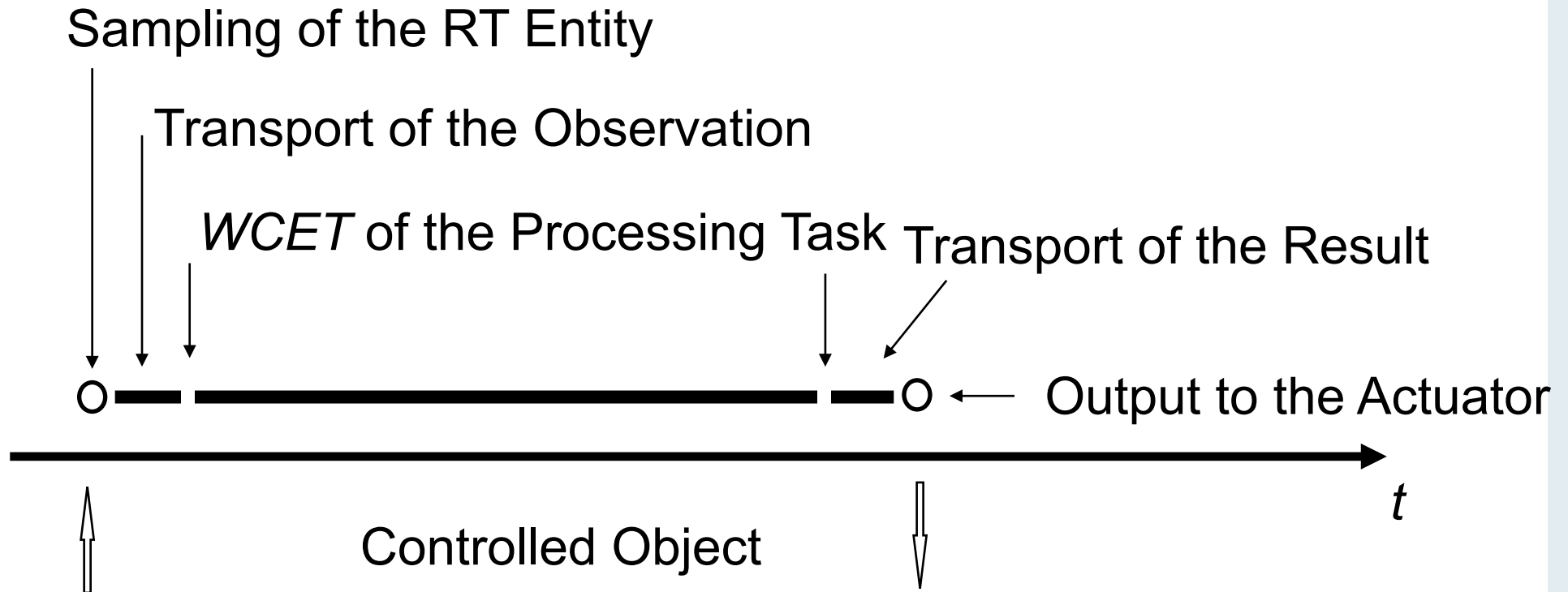
Example: Emergency stop

It is much more demanding to implement *time as control* than to implement *time as data*!

Sampling



Timing in a Sampled System



Sampling States vs. Events

Sampling refers to the periodic interrogation of the state of a RT entity by a computer.

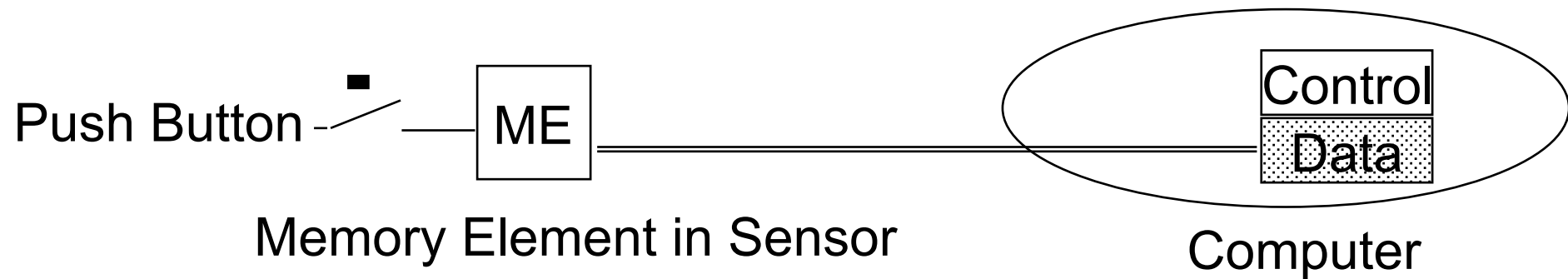
The duration between two sampling points is called the sampling interval.

The length of the sampling interval is determined by the dynamics of the real-time entity.

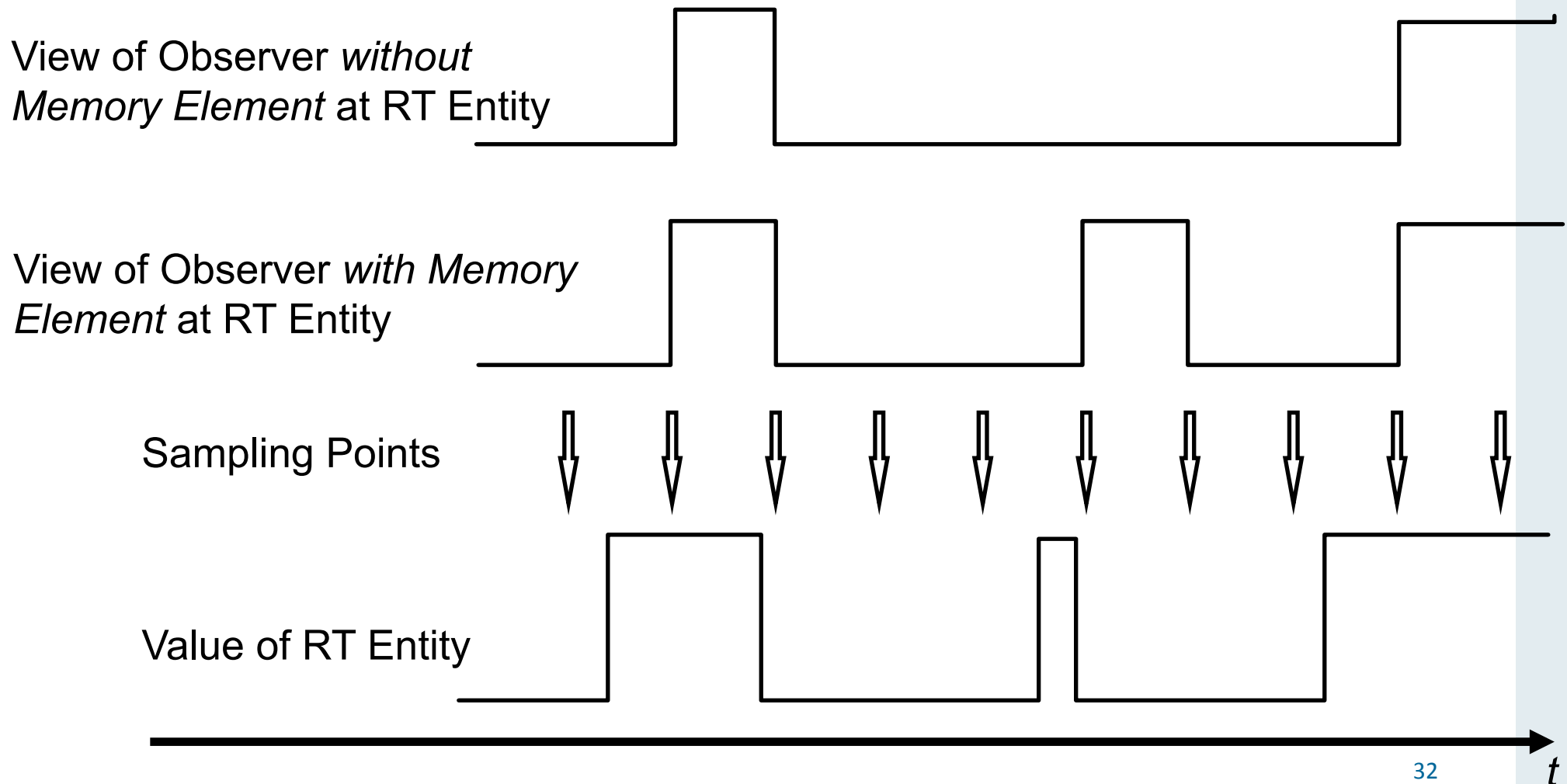
States can be observed by sampling.

Events cannot be sampled. They have to be stored in an intermediate memory element (ME).

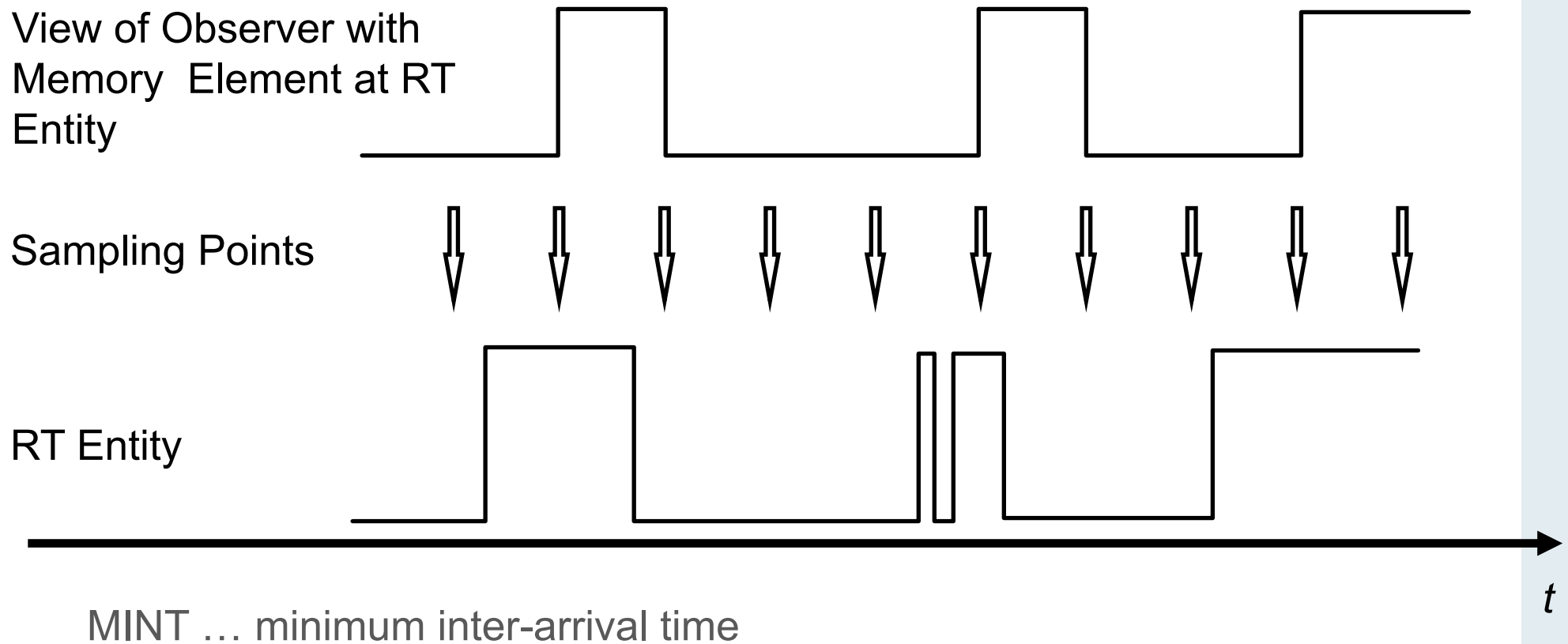
Sampling – Position of Memory Element



Sampling – Role of the Memory Element



Sampling – Importance of MINT



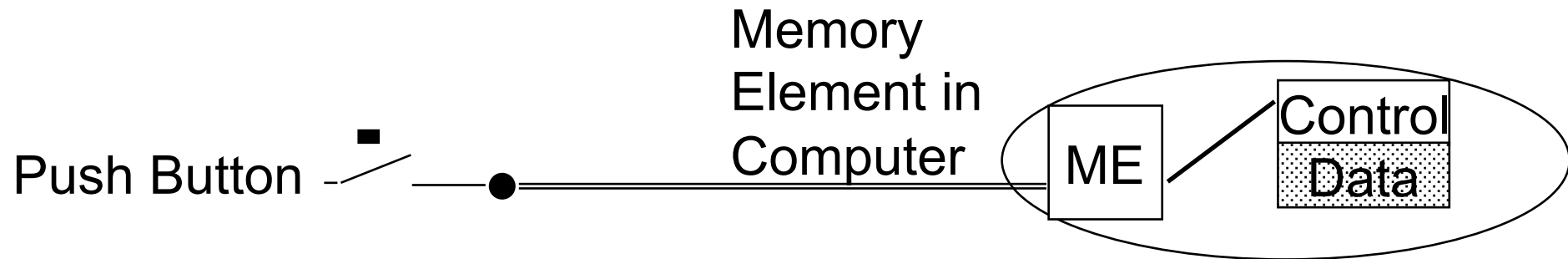
Interrupt

An interrupt is a hardware mechanism that periodically monitors (after the completion of each instruction – or CPU clock cycle) the state of a specified signal line (interrupt line).

If the line is active and the interrupt is not disabled, control is transferred after completion of the currently executing instruction (from the current task) to an instruction (task) associated with the servicing of the specified interrupt.

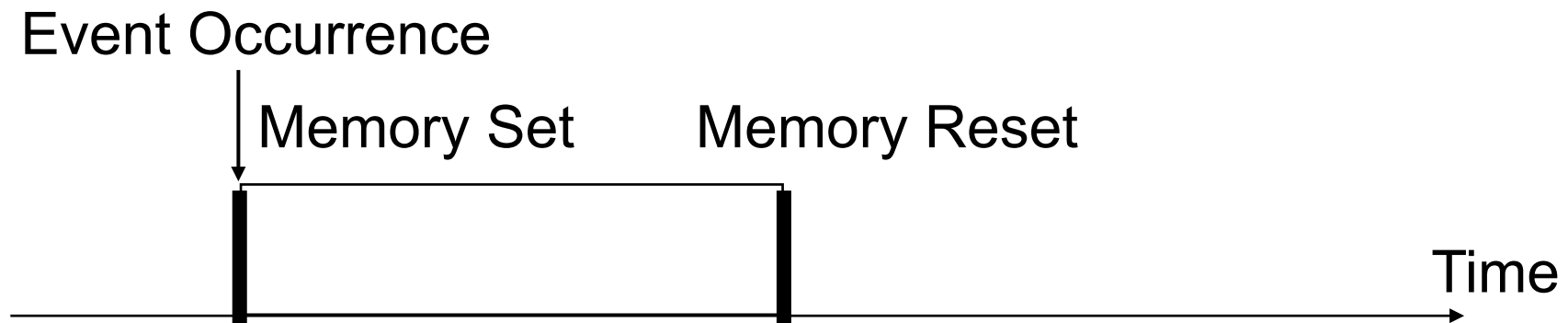
As soon as an interrupt is recognized, the state of the local “interrupt” memory is reset.

Interrupt



External event forces computer into interrupt service state.

Memory Element for an Event



Sampling: after a fixed period by sensor

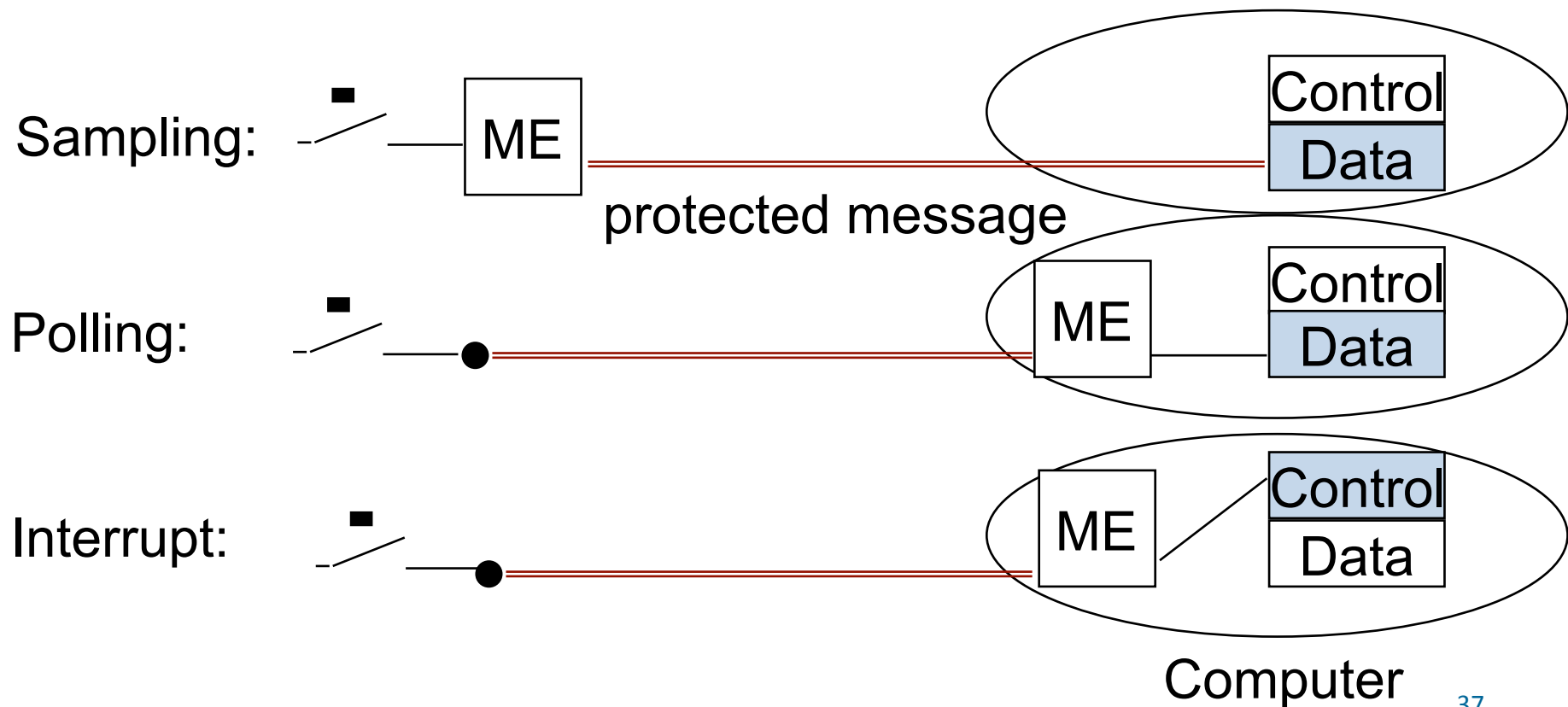
Polling: by CPU

Interrupt: by CPU

In the interval between the event occurrence and the resetting of the memory, no further events are recognized

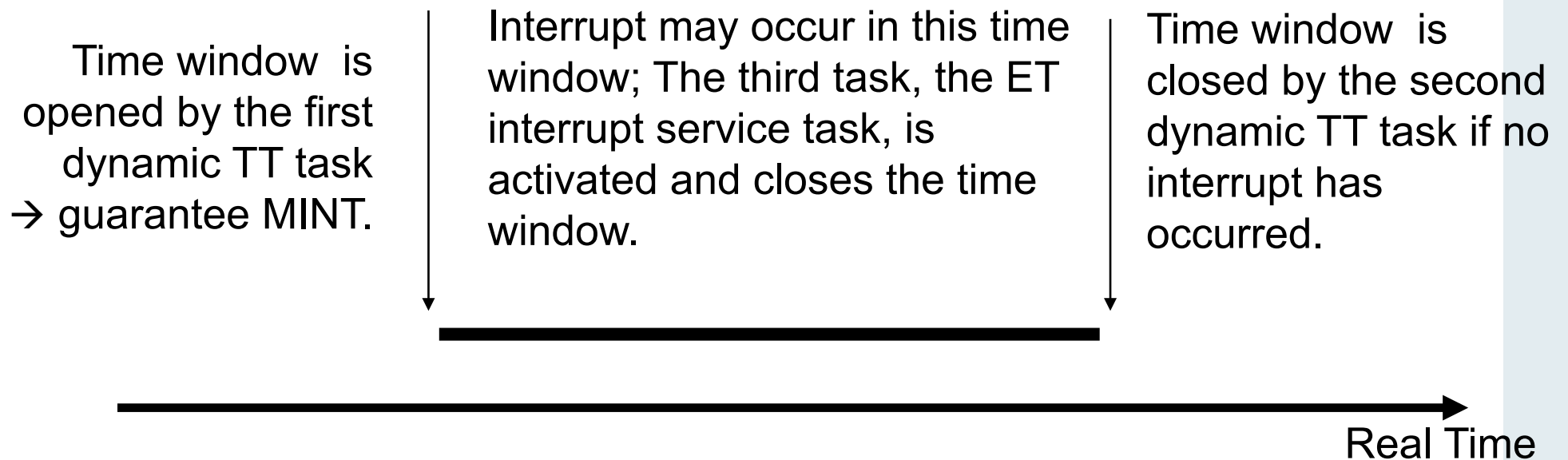
Sampling, Polling, Interrupt – Failures

What happens in the case of failure on transmission line?



Interrupt Handling

Three tasks to handle an interrupt:



Need for Agreement Protocols

If an RT entity is observed by two (or more) nodes of the distributed system, the following may happen:

- The same event can be time-stamped differently by two nodes – *fundamental limit of time measurement*.
 - When reading an analog sensor, a dense quantity is mapped onto a digital values – *discretization error*. Even sensors of highest quality may yield readings differing by a single-bit quantity.
- ➡ Whenever a dense quantity is mapped onto a discrete representation, *agreement protocols* are needed to get an agreed view on multiple redundant sensor readings

Agreement Protocol

An agreement protocol provides a *consensus* on the *value* of an observation and on the *time* when the observation occurred among a number of fault-free members of an ensemble:

- The first phase of an agreement protocol concerns the exchange of the local observations to get a globally consistent view to each of the partners
- In the second phase each partner executes the same algorithm on this global data (e.g., averaging) to come to the same conclusion – the agreed value and time

Agreement always needs an extra round of communication and thus weakens the responsiveness of a real-time system.

Agreement of Resource Controllers

As long as a number of resource controllers that observe a set of real-time entities has not agreed on the observations, active redundancy is not possible (and therefore no need for replica determinism):

- The world interface between a sensor and the associated resource controller can be serviced without concern for replica determinism (local interrupts!!)
- It should be an explicit design goal to eliminate the h-state from the resource controller (stateless protocols) – as far as possible.
- The message interface of the resource controller to the rest of a cluster should provide agreed values only!

Byzantine Agreement

Byzantine agreement protocols have the following requirements to tolerate the Byzantine failures of "f" nodes :

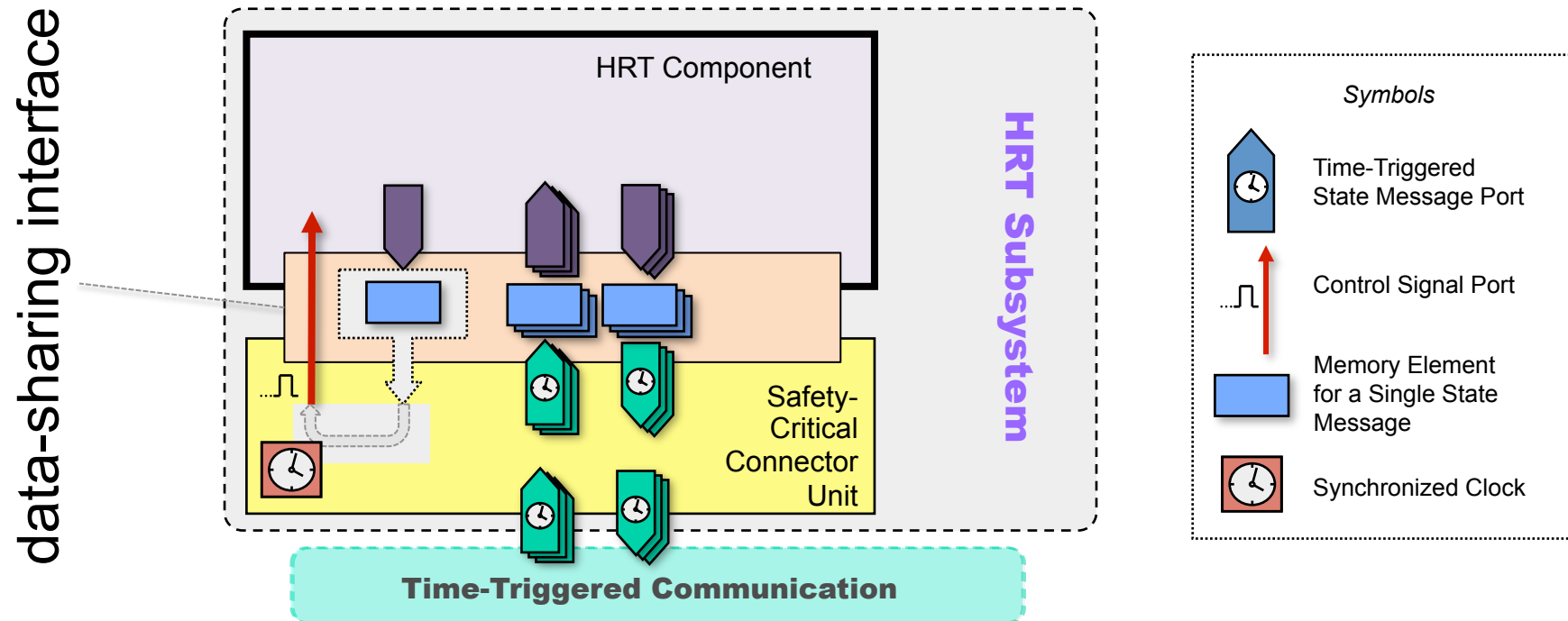
- There must be at least $3f+1$ nodes in a FTU.
- Each node must be connected to all other nodes of the FTU by $f+1$ disjoint communication paths.
- In order to detect the malicious nodes, $f+1$ rounds of communication must be executed among the nodes.
- The nodes must be synchronized to within a known precision of each other.

Error Detection Mechanisms

An RTOS must provide error detection in the temporal domain and in the value domain

- Consistency checks
- Monitoring task execution times
- Monitoring interrupts (MINT)
- Double execution of tasks (time redundancy)
- Watchdogs – observable heart-beat signal
- CRC checks

Separating Control Flow

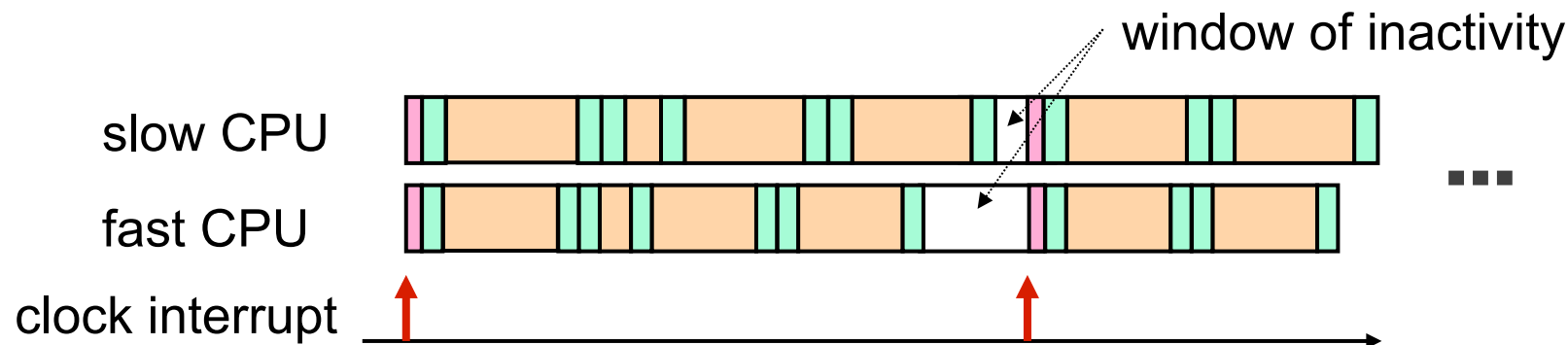


Synchronization with Real-Time Clock

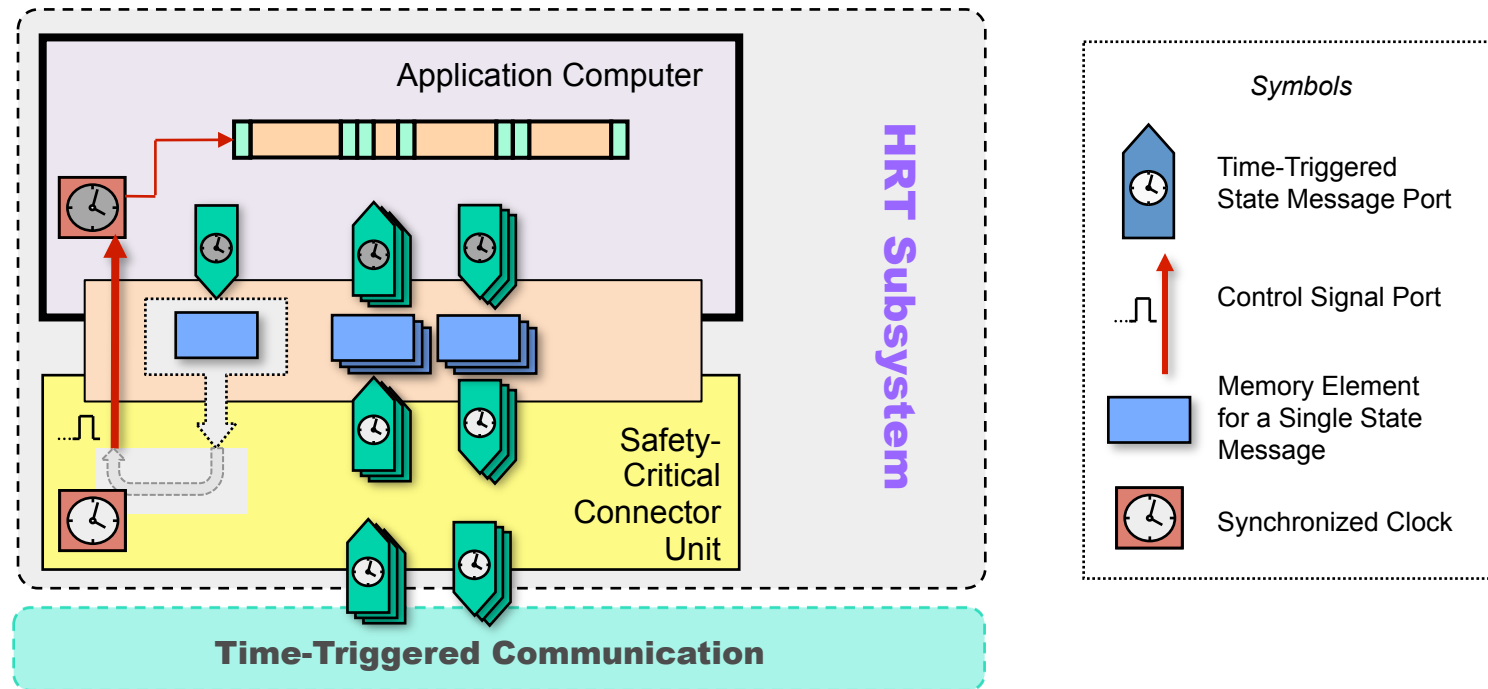
Master clock synchronization

- Programmed clock interrupt from connector unit

Planned window of inactivity before expected sync. time allows slow CPUs to complete the same workload as fast CPUs (needs bound on clock skew)



A Time-Predictable Component



Time-Predictable Component (2)



synchronized representation of global time



instruction counter “clock”,
synchronized to the local representation of global time



Static schedule (instruction-counter interrupt for preemptions)



Data transfer triggered by progression of instruction counter clock



Data transfer triggered by progression of global-time representation



Programmable clock interrupt to synchronize the instruction-counter clock with the global-time representation

Points to Remember

- Separation HRT SW vs. other SW
- Pre-planned TT operation keeps SW simple and verifiable
- Time services and the role of time
- Sampling I/O
 - Application-dependent timing parameters
 - Protection against failures
- Agreement at system borders