Technische Hochschule Rosenheim
Technical University of Applied Sciences

Prof. Dr. Florian Künzner
Technical University of Applied Sciences Rosenheim, Computer Science

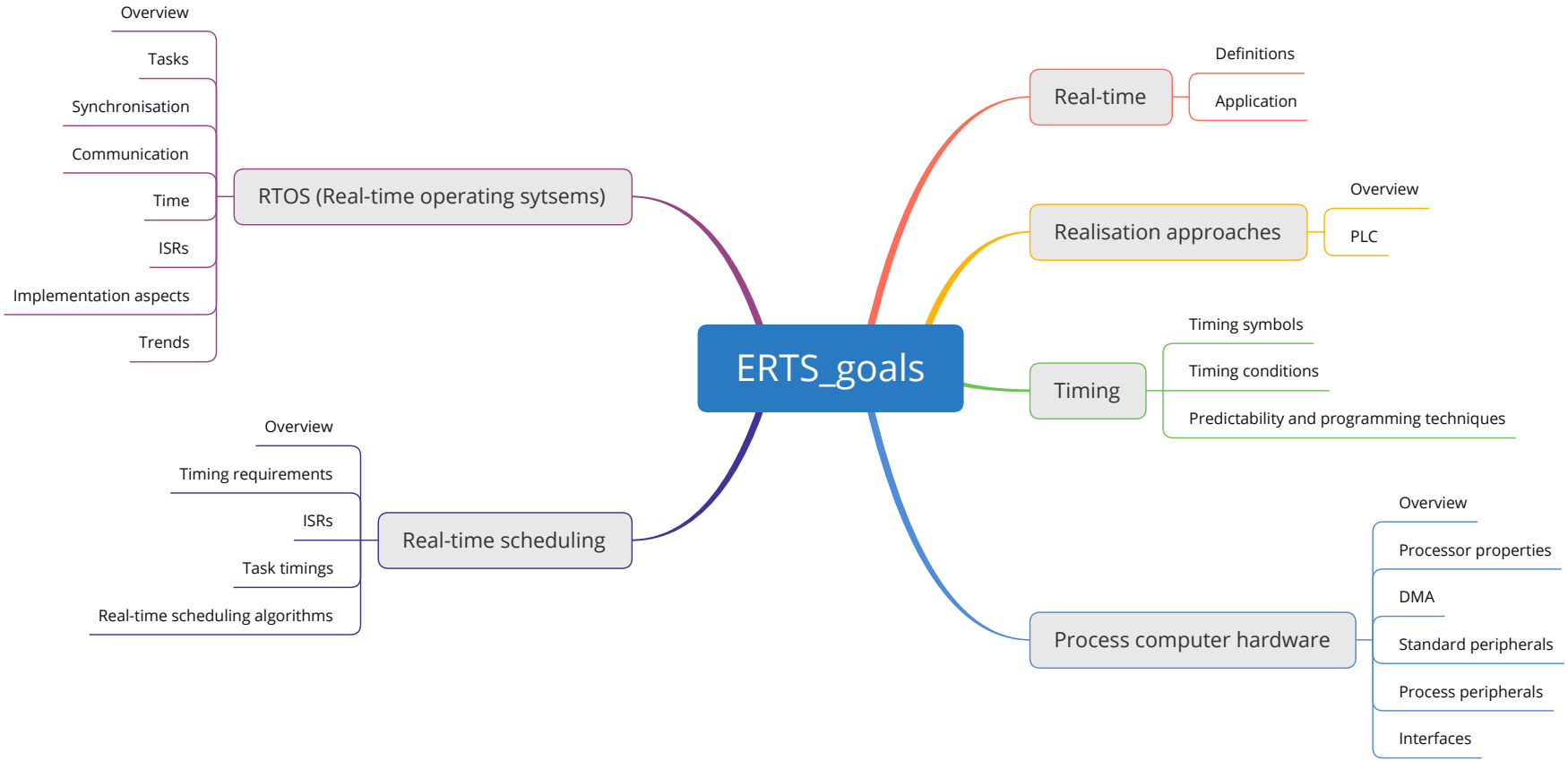
ERTS - Embedded real-time systems

ERTS 8 – RTOS

CAMPUS Rosenheim
Computer Science

Technische Hochschule Rosenheim
Technical University of Applied Sciences

Goal



CAMPUS Rosenheim
Computer Science

Technische Hochschule Rosenheim
Technical University of Applied Sciences

Goal

ERTS::Real-time operating systems

- Intro to RTOS
- Tasks
- Synchronisation
- Communication
- Interrupts
- Timers
- Trends

CAMPUS Rosenheim
Computer Science

Technische Hochschule Rosenheim
Technical University of Applied Sciences

Intro

What is an RTOS?

CAMPUS Rosenheim
Computer Science

Technische Hochschule Rosenheim
Technical University of Applied Sciences

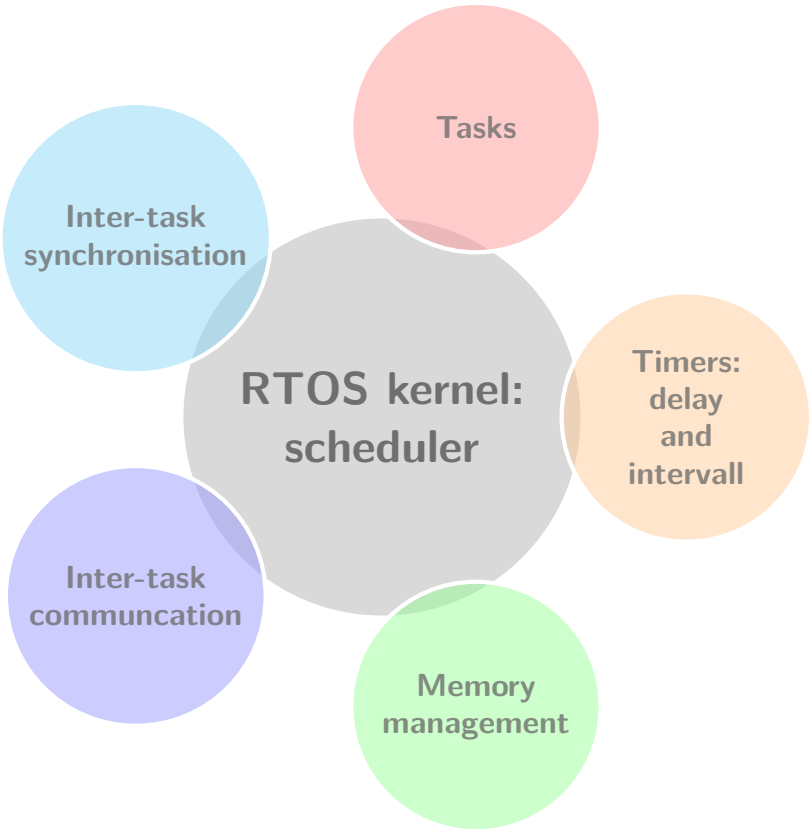
Intro

Difference between bare metal and using an RTOS?

CAMPUS Rosenheim
Computer Science

Technische Hochschule Rosenheim
Technical University of Applied Sciences

Building blocks of an RTOS.



Advantages and disadvantages using an RTOS

Advantages:

- Allows more complex application designs
- Allows scheduling of tasks: priority based, time slicing, ...
- Perform multiple simultaneously running functions
- Better structured software
- Easier maintenance of software

Disadvantages:

- Larger footprint
- Requires additional knowledge
- May introduce some additional jitter or delay (SysTick + scheduler required)

CAMPUS Rosenheim
Computer Science

Technische Hochschule Rosenheim
Technical University of Applied Sciences

Overview of different RTOS

- FreeRTOS
- Azure RTOS ThreadX
- VxWorks
- INTEGRITY RTOS
- Linux: Preempt RT (patch), RTLinux, Xenomai, L4Linux, ...
- ...

Some more: *Comparison of real-time operating systems* or *List of open source real-time operating systems*

CAMPUS Rosenheim
Computer Science

Technische Hochschule Rosenheim
Technical University of Applied Sciences

Tasks

Tasks are the basic execution entities in real-time operating systems.

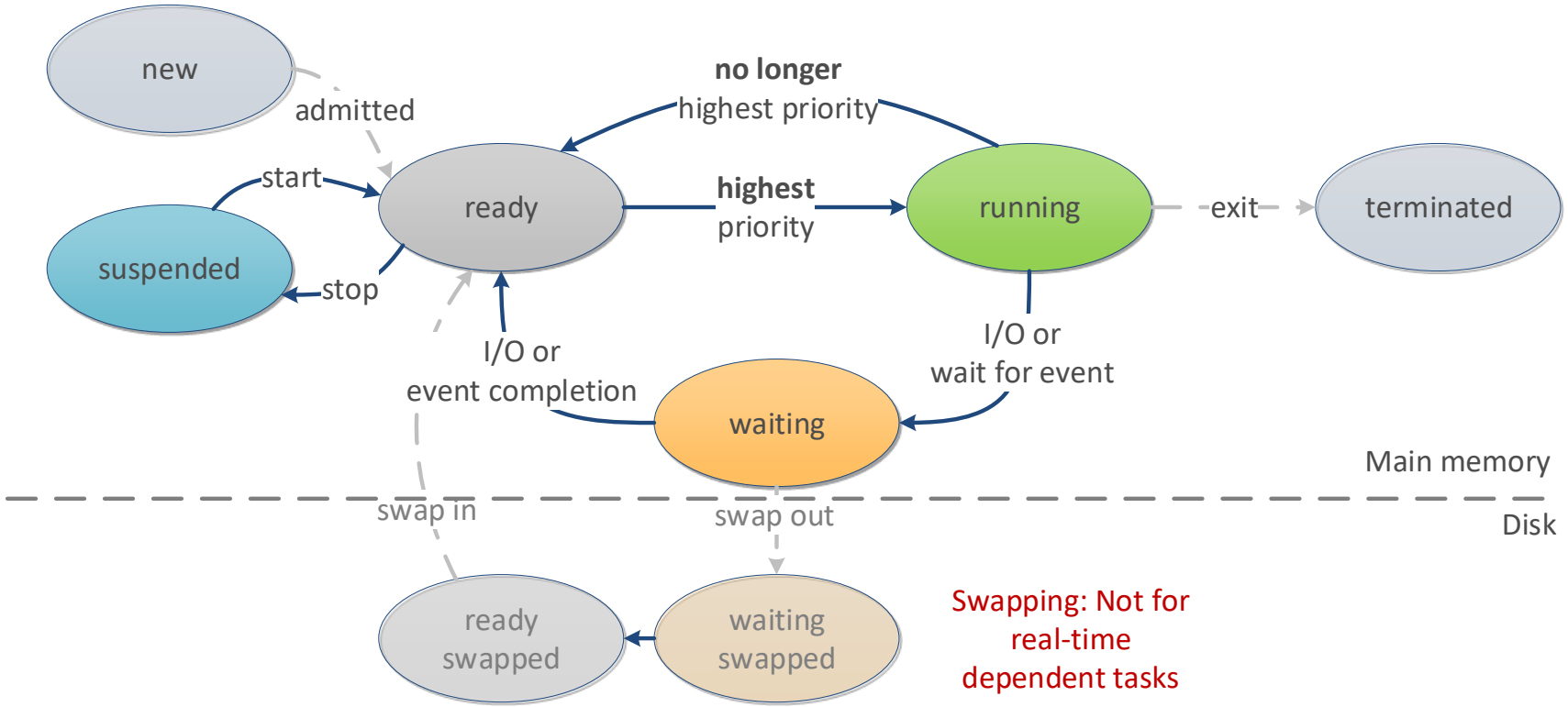
Operating system:

	Heavyweight process	**Lightweight process**
POSIX (original)		
Windows	Process	Thread
Linux	Process: `fork()`	Thread: `pthread_create()`
VxWorks	Real-time process (RTP)	Task
FreeRTOS		Task
Azure RTOS ThreadX		Thread

Real-time programming language:

Ada	Program, partition	Task
PEARL		Task

Task states


new
admitted
no longer
highest priority
start
ready
highest
priority
running
exit
terminated
suspended
stop
I/O or
event completion
I/O or
wait for event
waiting
Main memory
swap in
swap out
Disk
ready
swapped
waiting
swapped
Swapping: Not for
real-time
dependent tasks

Synchronisation

To synchronise tasks and tasks with ISRs, different synchronisation mechanisms exist.

Mechanisms:

- Semaphore
  - Binary semaphore: often called Mutex
  - Counting semaphore
- Event flags

$\Rightarrow$ Some RTOS may offer additional synchronisation mechanisms (e.g. FreeRTOS: direct to task notification).

CAMPUS Rosenheim
Computer Science

Technische Hochschule Rosenheim
Technical University of Applied Sciences

Semaphore

A semaphore is a signaling mechanism to control the access to common resources.

Operations:

Operation	Description
`seminit(s, value)`	Creates and **initialises a semaphore** with a value. The value is a number that specifies the number of processes that can simultaneously enter the critical area.
`P(s)`	**Wait** until the critical area is free (`value--`).
`V(s)`	**Releases** the critical area (`value++`).

Basic usage:

```
1 seminit(s, 1);
2
3 P(s);
4 //critical area..,
5 V(s);
```

Implementation:

Single-processor system:

- Disable interrupts (not preferred)

Multi-processor system:

- Test-and-set
- Compare-and-swap
- Spinlock (busy wait, for short waiting periods)

⇒ **Can support: Polling, FIFO queueing, priority queuing, and priority inheritance.**

Semaphore: Types

Types	Initialisation	Description
Mutex	`seminit(s, 1)`	A **mutex** semaphore is used for **mutual exclusion**. Typically initialised with 1.
Binary	`seminit(s, 0)`	A **binary** semaphore is used when there is only **one shared resource**. Initialisation with 0/1 possible.
Counting	`seminit(s, N)`	A **counting semaphore** is used to handle more than one shared resource. Typically initialised with the number N of shared resources. Initialisation with 0/N possible.

CAMPUS Rosenheim
Computer Science

Technische Hochschule Rosenheim
Technical University of Applied Sciences

Event flags

An event bit (or flag) is a one bit information to synchronise tasks or tasks with an ISR.

Often, the event bits are combined into an event group.

Usually these features are available:

- `EVENT_SET` Sets an event bit
- `EVENT_CLEAR` Clears an event bit
- `EVENT_WAIT` Waits until an event occurs; sometimes also OR, AND, NOT combinations possible

Additional possible:

- `EVENT_WAIT_ANY` Waits for any of the specified event bits (OR)
- `EVENT_WAIT_ALL` Waits for all of the specified event bits (AND)

Comparison of semaphores and events

Event flags	Counting semaphores
State (bit information)	Counter
On `EVENT_WAIT`, no change	`P()` consumes an entity from counter
`EVENT_WAIT` of n tasks is satisfied by one `EVENT_SET`	`P()` of n tasks requires n `V()` operations
`EVENT_SET` and `EVENT_CLEAR` without an effect, if no task is waiting	`V()` (without an corresponding `P()`) increases the counter
Typically: AND, OR available in `EVENT_WAIT`	Typically: AND, OR **not** available in `P()`

Sometimes:

- Combination of semaphore and event flags possible
- Linux supports semaphore arrays which allow AND, OR

Communication overview

Different RTOS provide different communication mechanisms*:

Communication mechanism	Inter process	Over network	Stream (byte) oriented	Message oriented
Message queue	X			X
Stream buffers	X		X	
Sockets (TCP/IP)	x	X	X	
Pipes	X		X	x
Shared memory	X		X	x

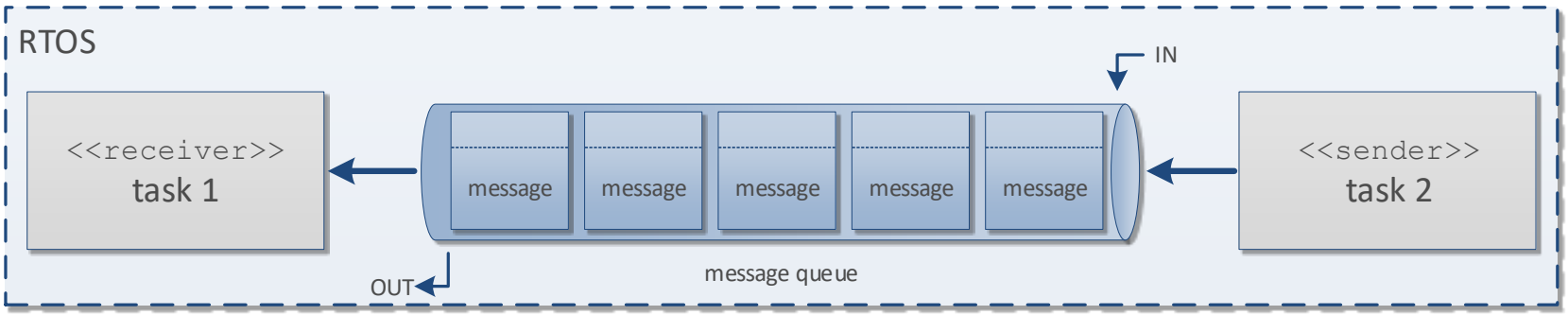
*The concrete naming and the implemented feature set depends strongly on the RTOS and can differ significantly.

CAMPUS Rosenheim
Computer Science

Technische Hochschule Rosenheim
Technical University of Applied Sciences

Communication

Message queues: a place for one or multiple messages



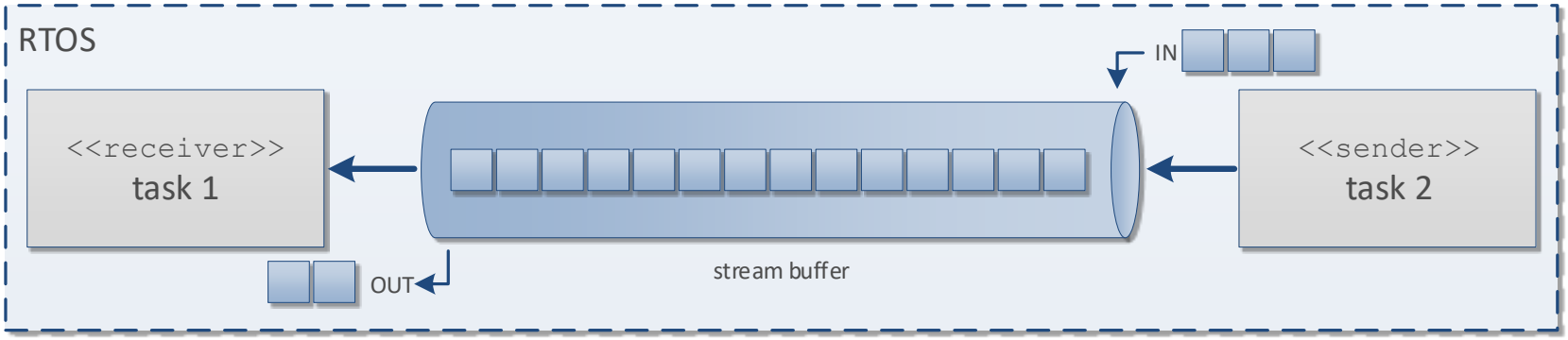
- Task to task communication
- Task to ISR communication

CAMPUS Rosenheim
Computer Science

Technische Hochschule Rosenheim
Technical University of Applied Sciences

Communication

Stream buffers: a stream of bytes



- Task to task communication
- Task to ISR communication

Communication patterns

Common communcation patterns:

- 1:1 communication (one queue, or multiple queues)
- n:1 communication (one queue, or multiple queues)
- 1:n communication (one queue, or multiple queues)
- m:n communication

Other variants:

- Messages buffered or unbuffered
- Blocking or non-blocking send/read
- Broadcast (send to n mailboxes in parallel)
- Communication via sockets (network, TCP/IP)

CAMPUS Rosenheim
Computer Science

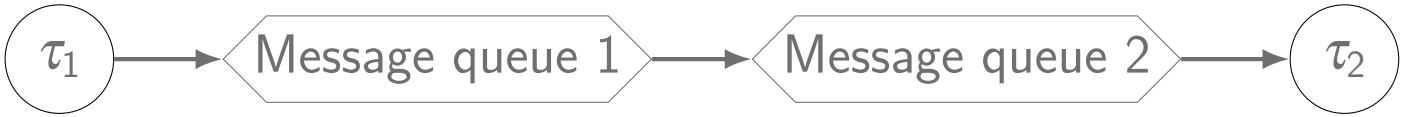
Technische Hochschule Rosenheim
Technical University of Applied Sciences

1:1 communication

1:1 communication with one message queue:

A fixed communication between two tasks.

1:1 communication with multiple message queue in a sequence:



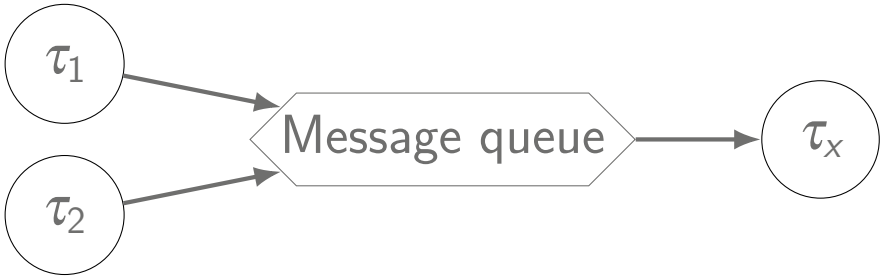
A more flexible communication pattern between two tasks.

CAMPUS Rosenheim
Computer Science

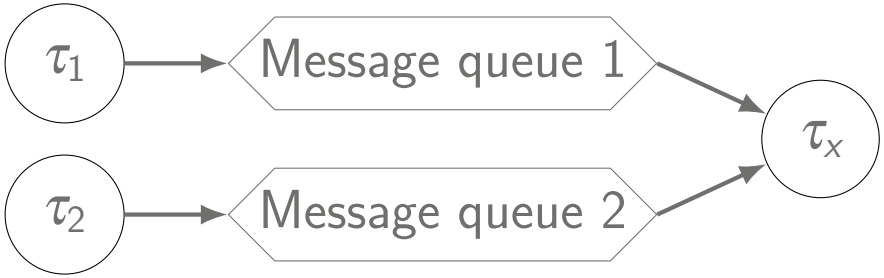
Technische Hochschule Rosenheim
Technical University of Applied Sciences

n:1 communication

n:1 communication with one message queue



n:1 communication with multiple queues

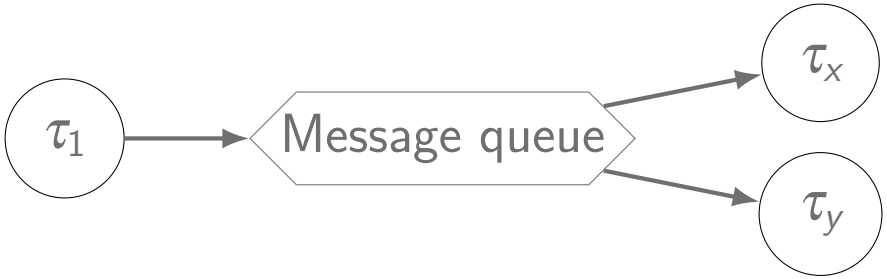


CAMPUS Rosenheim
Computer Science

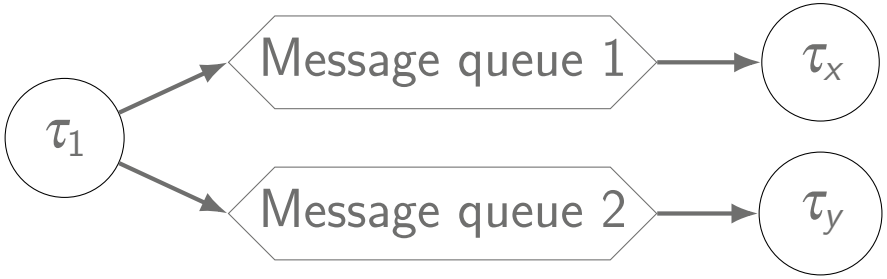
Technische Hochschule Rosenheim
Technical University of Applied Sciences

1:n communication

1:n communication with one message queue



1:n communication with multiple message queues

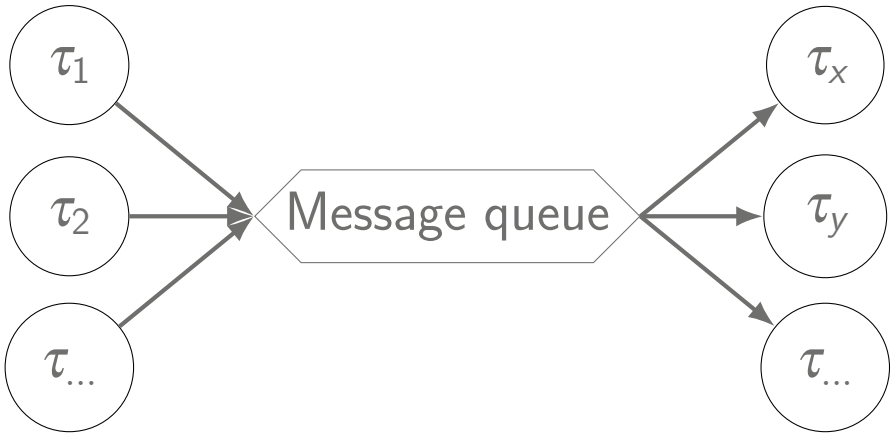


CAMPUS Rosenheim
Computer Science

Technische Hochschule Rosenheim
Technical University of Applied Sciences

m:n communication

m:n communication with one message queue



Interrupts

Interrupts do have:

- No state information (like a TCB (task control block)):
- No stack
- No virtual PC
- No own register set

$\Rightarrow$ Therefore, no blocking system calls are allowed (like `P()`)

But:

- Non-blocking system calls are possible (like `V()`)
- Secondary reaction: resume a task

Implementation:

- With assembler, or **with a higher level language: e.g. C/C++**
- Registration: directly within the interrupt vector table, or through an RTOS kernel wrapper function

CAMPUS Rosenheim
Computer Science

Technische Hochschule Rosenheim
Technical University of Applied Sciences

Interrupts
Variant 1: Mini ISR

Mini ISR:

```
1 void ISR() {
2   //e.g. transfer 1 word from/to I/O register
3   //      to/from an internal buffer
4   buffer[i] = ...
5   ++i;
6 }
```

Details:

- No interaction with the RTOS kernel (no system calls)
- Interruptions through such ISRs can usually always be allowed
- In practise usually not sufficient (because no notification is done)

CAMPUS Rosenheim
Computer Science

Technische Hochschule Rosenheim
Technical University of Applied Sciences

Interrupts
Variant 2: ISR with system calls (non-blocking)

ISR with system calls (non-blocking):

```
1 void ISR() {
2   ENTER_KERNEL(); //save registers, switch into system address space
3
4   //prepare message
5   Message m = ...
6   //e.g. MESSAGE_SEND  m to task t
7   MESSAGE_SEND(m, t);
8 }
```

Details:

- Direct `ENTER_KERNEL();` mechanisms required
- Such an ISR can not run during other kernel activities (e.g. `MESSAGE_SEND();`)
- Sometimes, ISRs are executed in the system address space (e.g. ARM Cortex M)

CAMPUS Rosenheim
Computer Science

Technische Hochschule Rosenheim
Technical University of Applied Sciences

Interrupts
Variant 3: combination of 1 and 2

Mini ISR with system calls (non-blocking):

```
1 void ISR() {
2   //e.g. transfer 1 word from/to I/O register to/from an internal buffer
3   buffer[i] = ...
4   if (buffer[i] == EOF){
5     ENTER_KERNEL(); //save registers, switch into system address space
6     V(TRANSFER_READY);
7   }
8   ++i;
9 }
```

Details:

- Often used pattern
- Problem: When should such an ISR be allowed?

CAMPUS Rosenheim
Computer Science

Technische Hochschule Rosenheim
Technical University of Applied Sciences

Interrupts

Variant 4: ISR with system call buffer

Mini ISR with system calls (non-blocking):

```
1  void ISR() {
2    //e.g. transfer 1 word from/to I/O register to/from an internal buffer
3    buffer[i] = ...
4    if (buffer[i] == EOF){
5      if(ENTER_KERNEL()) { //save registers, switch into system address space
6        V(TRANSFER_READY);
7      } else {
8        //Buffer: V(TRANSFER_READY);
9        //and execute it later through the RTOS
10     }
11   }
12   ++i;
13 }
```

Details:

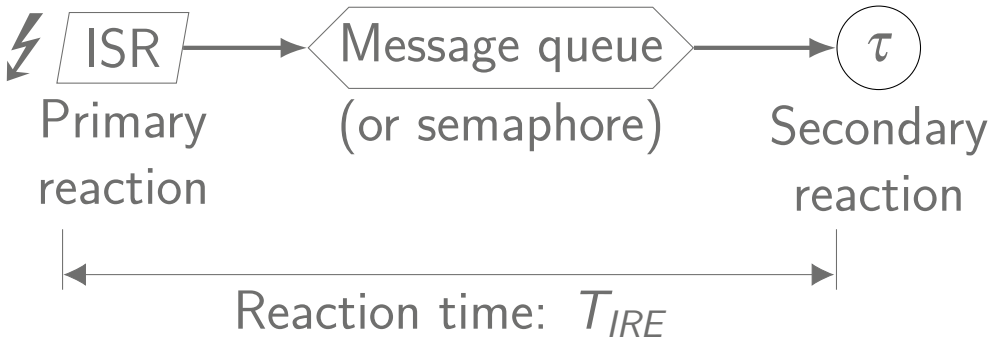
- Also known as the deferred interrupt handling (or deferred procedure call in Windows)
- Minimises the execution time of ISRs (can improve response time for otherwise long running ISRs)

CAMPUS Rosenheim
Computer Science

Technische Hochschule Rosenheim
Technical University of Applied Sciences

Interrupts

Primary/secondary reaction pattern



Process dispatch latency time (PLDT)

```
1 void ISR() {
2   Message m = ...
3   MESSAGE_SEND(QUEUE_ID, m);
4 }
```

```
1 void task() {
2   Message m;
3   //blocks until a message is available
4   MESSAGE_RECEIVE(QUEUE_ID, &m);
5
6   //work with message...
7   //can take a while
8 }
```

CAMPUS Rosenheim
Computer Science

Technische Hochschule Rosenheim
Technical University of Applied Sciences

Timers

Typical time services in RTOS:

- Start a timer (timer task or function callback)
  - in x relative time unites (e.g in 5 seconds)
  - at absolute time x (e.g at 12:23:01 o'clock)
  - periodic with cycle time x (e.g. 5 seconds with auto reload)
- Resume a timer (time specification as for start)
- Stop a timer
- Watchdog: Reporting a timeout (e.g. when waiting for I/O, semaphore, event, or message)

Implementation: HW vs SW timer

Hardware timer:

- Provided by the HW (timer or/and real-time clock)
- Usually, very accurate timing (low jitter)
- Requires the implementation of an ISR and a task notification mechanism

Software timer:

- Provided by the RTOS
- May be not as precise as HW timers (some jitter)

Trends

Multiprocessor or multi core systems are used for real-time systems

- *n* tasks run on *m* equally processors (CPU cores)
- Real parallel execution of tasks
- If correctly synchronised, everything works such as on a single core CPU
- Scheduling guarantees (and response times) are much harder to proof

Some details:

- Homogeneous vs heterogeneous multicore systems
- Symmetric vs asymmetric multiprocessing

CAMPUS Rosenheim
Computer Science

Technische Hochschule Rosenheim
Technical University of Applied Sciences

Homogeneous vs heterogeneous multicore systems

Homogeneous multicore systems

- Multiple CPU cores
- Each CPU core has the same architecture
- Example: ARM Cortex A53 (quad core)

Heterogeneous multicore systems

- Multiple CPU cores
- Two or more CPU cores that differ in architecture
- Example: BeagleBone black (1 Cortex A8 (AM3358), 2x Cortex M3) or NXP i.MX7D
- Example: NXP i.MX7D (Cortex A7 + M4)

Advantages of heterogeneous systems:

- Performance optimisation
- Reduction of power consumption
- Improved system reliability and security
- Processing real-time tasks and non real-time tasks (such as GUI) on the same system

[source: nxp.com]

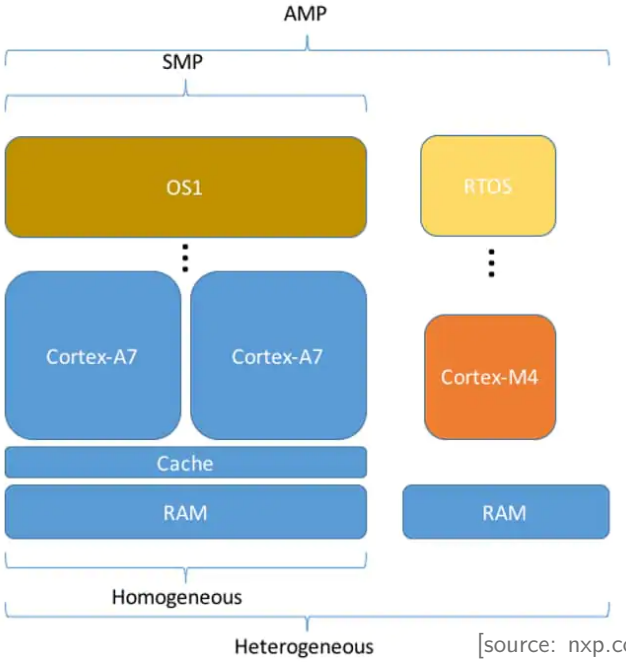
CAMPUS Rosenheim
Computer Science

Technische Hochschule Rosenheim
Technical University of Applied Sciences

Symmetric vs asymmetric multiprocessing

Symmetric multiprocessing (SMP)

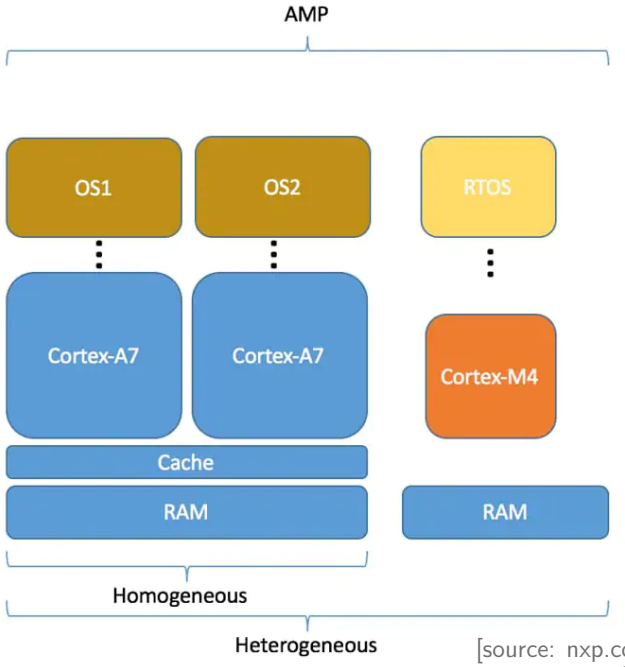
- One kernel, multiple cores (usually identical cores)
- Enables load balancing and real parallel processing of tasks



[source: nxp.com]

Asymmetric multiprocessing (AMP)

- Contains multiple cores (homogeneous or heterogeneous cores)
- Usually, more than one OS is running



[source: nxp.com]

CAMPUS Rosenheim
Computer Science

Technische Hochschule Rosenheim
Technical University of Applied Sciences

Summary and outlook

Summary

- Intro to RTOS
- Tasks
- Synchronisation
- Communication
- Interrupts
- Timers
- Trends

Outlook

- FreeRTOS