Technische Hochschule Rosenheim
Technical University of Applied Sciences

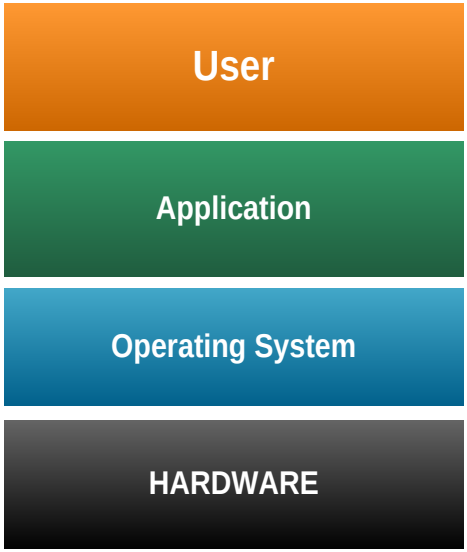
# Embedded Systems
# Kapitel 09: Embedded Betriebssysteme

Prof. Dr. Wolfgang Mühlbauer
Fakultät für Informatik
wolfgang.muehlbauer@th-rosenheim.de

Sommersemester 2020

# Definitionen

- ***Betriebssystem / Operating System (OS)***
  - Schnittstelle zwischen Anwendung und Hardware
  - Typische Komponenten:
    - Scheduler
    - Speicherverwaltung
    - Dateisystem
    - Kommunikation übers Netzwerk und zwischen Prozessen
    - GUI




- ***Embedded OS***
  - Fokus: Hohe Zuverlässigkeit, Einsparen von Ressourcen
  - OS für Mikrocontroller oder Einplatinencomputer
  - Manche Features, z.B. Speicherverwaltung, Dateisystem können fehlen.
  - OS-Code oft im ROM, nicht auf Festplatte.
  - Anwendung und eigentliches OS bilden häufig eine Einheit

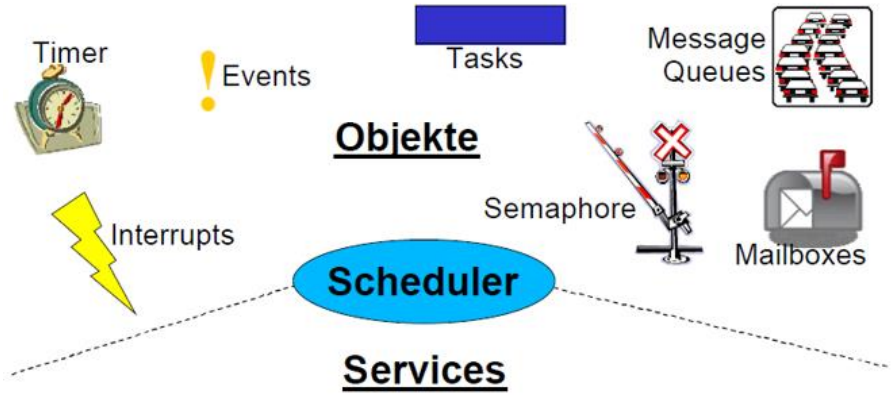
- Viele Embedded OS sind ***echtzeitfähig***
  - *Real-Time Operating Systems* (RTOS) garantieren, dass Aufgaben innerhalb einer bestimmbaren Frist erledigt werden.
  - → INF-M: *Eingebettete Echtzeitsysteme*, Prof. Künzner

# Anforderungen an ein Embedded OS

- Hohe Zuverlässigkeit: Stabiler Betrieb rund um die Uhr
- Einsparen von Ressourcen
- Prozesssynchronisation und –kommunikation
  - Semaphor, Mutex, usw.
- Echtzeitfähigkeit  [optional, aber häufig implementiert]
- Speicherverwaltung [optional]
- Ein- und Ausgabe [optional]
- TCP/IP [optional]

# Wichtige Elemente eines Embedded OS

- ***Task***
  - Programm wird in mehrere Tasks (== Jobs) zerlegt.
  - Tasks werden *scheinbar nebenläufig* ausgeführt (== *präemptives Multitasking*).
- ***Scheduler***
  - Entscheidet, welcher Task wann wie lange ausgeführt wird.
- ***Timer***
  - API zum "einfachen" Zugriff auf Hardware-Timer
- ***Kommunikation / Synchronisation zwischen Tasks***
  - Semaphore, Mutex, Queue,usw.
- ***Ressourcen:*** Zugriff auf
  - I/O
  - Speicher
  - TCP/IP



# FreeRTOS

- Populäres, schlankes Embedded OS / RTOS.
- Portiert auf verschiedene Mikrocontroller-Plattformen.
- Lizensiert durch GPL
- FreeRTOS bietet
  - Mehrere Tasks, Scheduling
  - Mutex, Semaphore
  - Timer
  - Dynamisches, **prioritätenbasiertes**, präemptives Scheduling

**Supported Architecture**

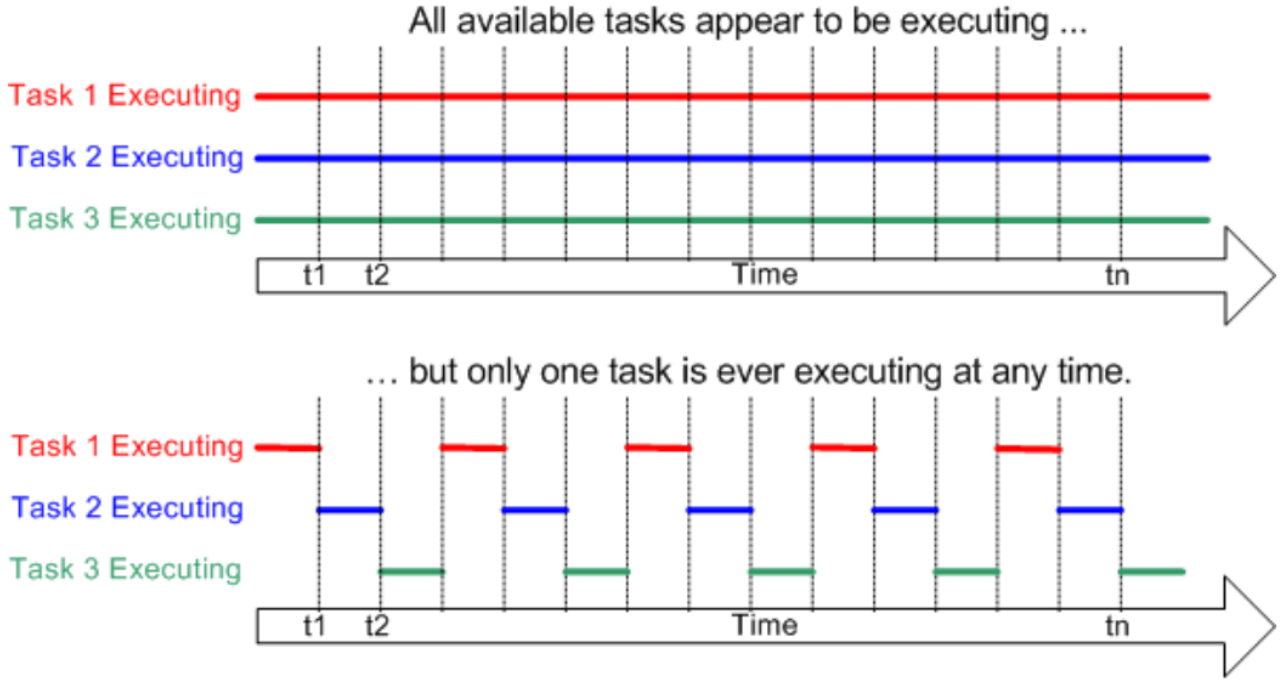
- **Altera Nios II**
- **ARM architecture**
  - ARM7
  - ARM9
  - ARM Cortex-M0
  - ARM Cortex-M0+
  - ARM Cortex-M3
  - ARM Cortex-M4
  - ARM Cortex-M7
  - ARM Cortex-A

- **AtmelAtmel**
- AVR
- -- AVR32
- -- SAM3
- -- SAM4
- -- SAM7
- -- SAM9
- -- SAM D20
- -- SAM L21
- **Intel**
- -- x86
- -- 8052

- **Freescale**
- -- Coldfire V1
- -- Coldfire V2
- -- HCS12
- -- Kinetis
- **IBM**
- -- PPC405
- -- PPC404
- **Infineon**
- -- TriCore
- -- Infineon XMC4000

# Präemptives Multitasking

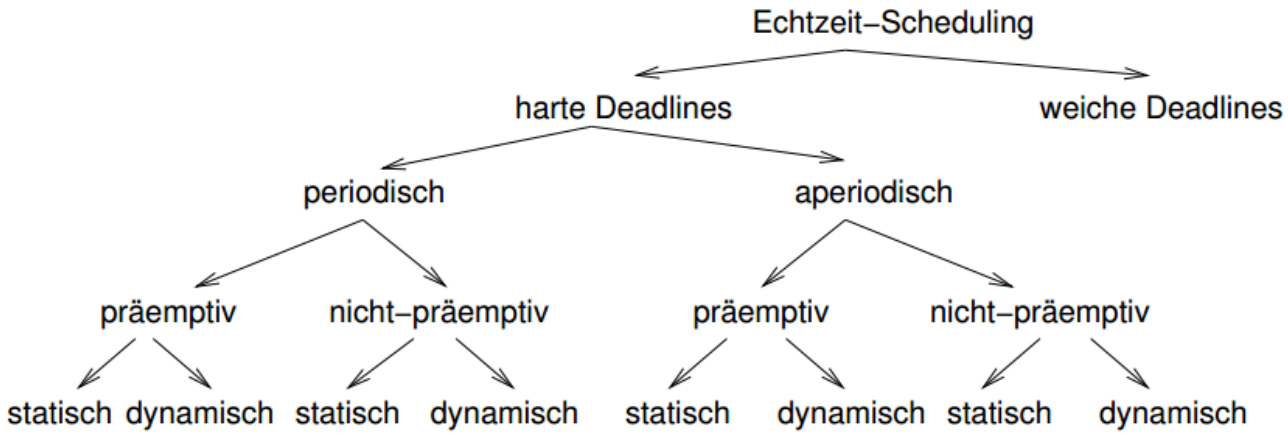
- ***Task*** == ausführbares Programm
  - Jeder Task kann unabhängig von anderen Tasks ausgeführt werden.
- ***Multitasking:*** OS kann mehrere Tasks ausführen
- Tasks werden ***scheinbar nebenläufig*** ausgeführt
  - OS schaltet zwischen den Tasks um.



# Scheduling

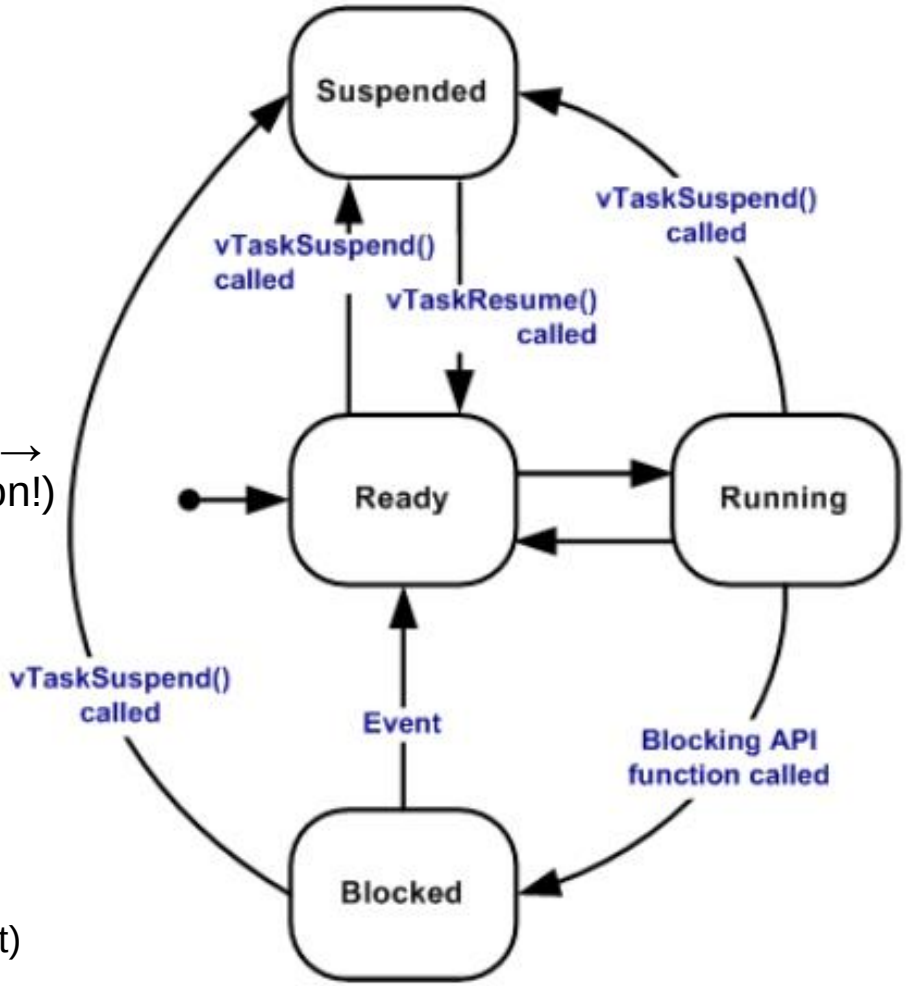
- **Aufgabe**
  - Entscheidet, welcher Task wann wie lange ausgeführt wird.
  - Zuteilung von Rechenzeit an die ***aktiven*** Tasks

- **Klassifikation**
  - ***Periodisch vs. aperiodisch***
    - Periodische Tasks werden in regelmäßigen Zeitabständen ausgeführt.
    - Aperiodisch: Task wird bei Eintreten eines Ereignisses ausgeführt.
  - ***Präemptiv:***
    - Scheduler kann Task vor dessen Ende unterbrechen.
  - ***Statisch / dynamisch***
    - Scheduling findet erst zur Laufzeit statt.



# Scheduling in FreeRTOS

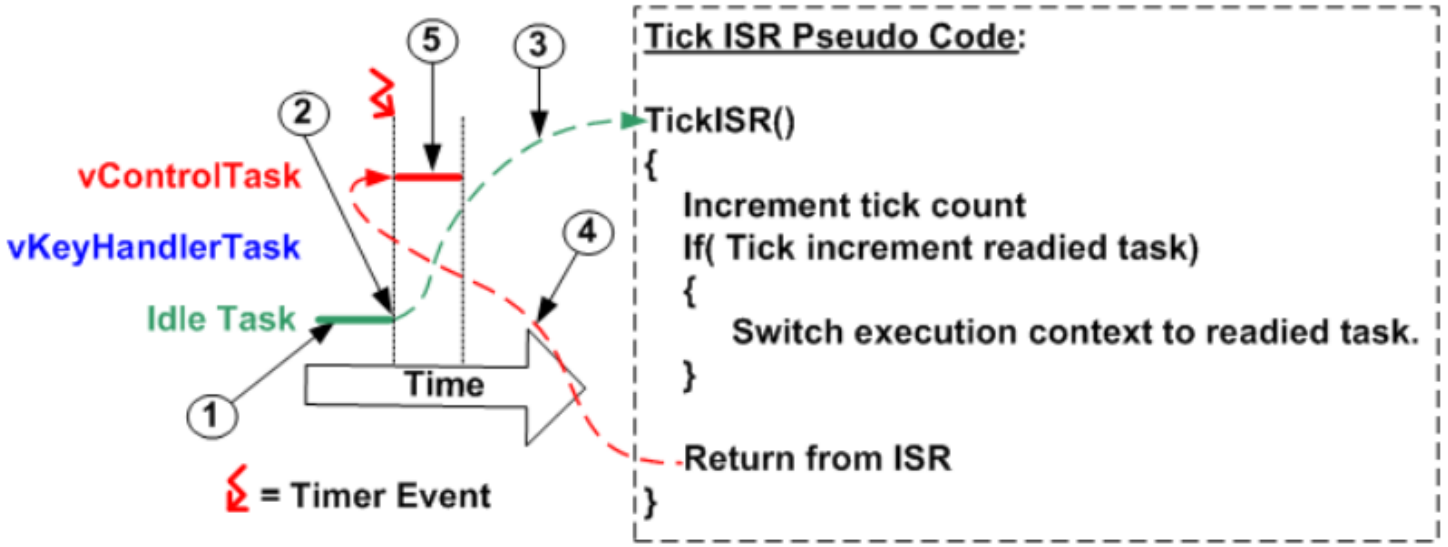
- 4 ***Taskzustände***
  - `READY, RUNNING, SUSPENDED, BLOCKED`
- Jeder Task hat ***konfigurierbare Priorität***
  - Prio 5 höher als Prio 2
- ***Scheduling Policy***: Wer wird `RUNNING`***?***
  - Führe `READY`-Task mit höchster Prio aus
  - Falls mehrere `READY` Tasks mit höchster Prio → Round-Robin Wechsel (Standard-Konfiguration!)
  - Falls kein `READY`-Task: Führe *Idle* Task aus
- Wann enden Tasks?
  - Scheduler unterbricht `RUNNING` Task, da
    - `READY` Task mit höherer Prio wartet
    - Round-Robin bei 2 Tasks mit höchster Prio
  - `RUNNING` Task unterbricht sich selbst.
    - *sleep*: Möchte einige zEit schlafen
    - *block*: Warten auf Ressource (z.B. serieller Port)
    - *event*: Wartet auf Ereignis (z.B. Tastendruck)



Valid task state transitions

# FreeRTOS Timing

- FreeRTOS abstrahiert HW-Timer (bei AVR: `Timerx`)
  - Definierbare Zeitspanne bis zum Aufruf einer Callback-Funktion
- FreeRTOS misst die Zeit mit einer **tick count** Variable
  - Bei AVR: `Timer1` Interrupt (siehe Vorlesung 04) erhöht diese Variable
- Scheduler wird bei jeder Erhöhung aufgerufen und entscheidet welcher Task in den `RUNNING` State kommt.
- Task für 100 ms blockieren: `vTaskDelay(pdMS_TO_TICKS(100));`



# Beispiel mit 2 Tasks

Definition eines Tasks

Loop-Methode bleibt leer

Wird nur einmal ausgeführt

Wird immer wieder ausgeführt

Task blockiert. Falls anderer Task READY, käme dieser zum Zug. Kein `delay` verwenden!

```
1  #include <Arduino_FreeRTOS.h>
2
3  void TaskA(void *pvParameters); // define two tasks, prototypes
4  void TaskB(void *pvParameters);
5
6  void setup() {
7      // Set up two tasks to run independently.
8      xTaskCreate(TaskA
9                  , "Der Task A"   // a name just for humans
10                 , 128            // Stack size
11                 , NULL
12                 , 2              // priority
13                 , NULL);
14
15     xTaskCreate(TaskB, "Der Task B", 128, NULL, 1, NULL);
16
17     // Now task scheduler is automatically started.
18 }
19
20 void loop() {}
21
22 /*---------------------- Tasks --------------------- */
23 void TaskA(void *pvParameters) {
24
25     // initialize what you need ("setup-method of a task")
26
27     for (;;) { // a task never exits
28         // go to BLOCKED, sleep for 1000 ms
29         vTaskDelay(pdMS_TO_TICKS(1000));
30     }
31 }
32
33 void TaskB(void *pvParameters)  {
34     for (;;) {}
35 }
```

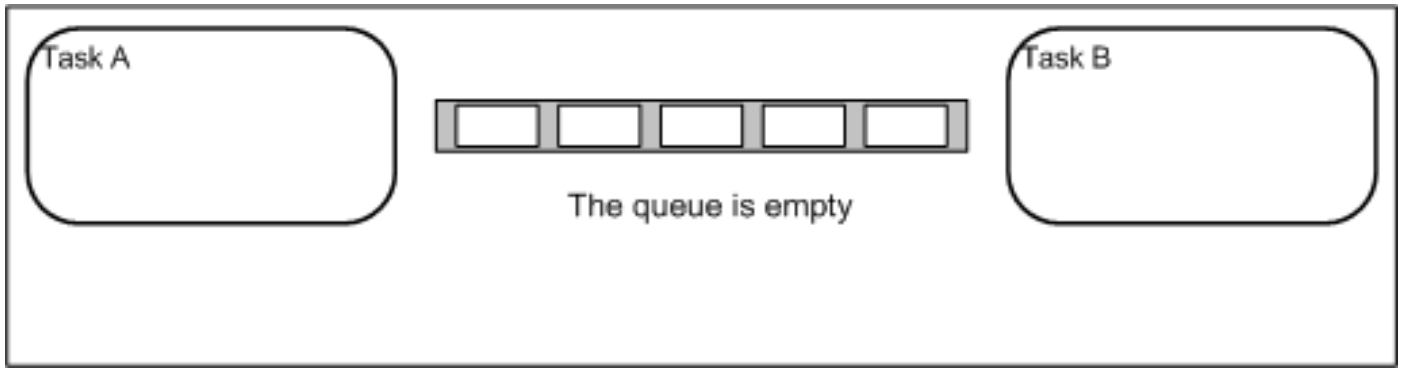
# Kommunikation zwischen Tasks

- ***Interprozess-Kommunikation (IPC)***
  - Nachrichtenaustausch zwischen Tasks
  - *Mutual Exclusion:* Kein gleichzeitiger Zugriff auf gemeinsame Ressource
  - *Synchronization*: Geordnete Abarbeitung bei Abhängigkeiten, z.B. Task *A* muss auf Task *B* warten

- FreeRTOS Mechanismen für IPC, siehe Betriebssysteme
  - ***(Binärer) Semaphor***
    - Löst das Producer-Consumer Problem
    - Verwaltung zählbarer Ressourcen
  - ***Mutex*** [kein Teil der Übung]
    - "Nur" Mutual Exclusion für gemeinsame Ressource, kein Zählen
    - Gleiche Implementierung wie binärer Semaphor jedoch
      - "Give" und "Take" durch gleichen Task
      - Adressiert Priority Inversion (kein Thema der Vorlesung)
  - ***Queue*** / Mailbox / Message Passing
    - Nachrichtenaustausch über Queues
  - ***Direct Task Notifications*** [kein Teil der Übung]

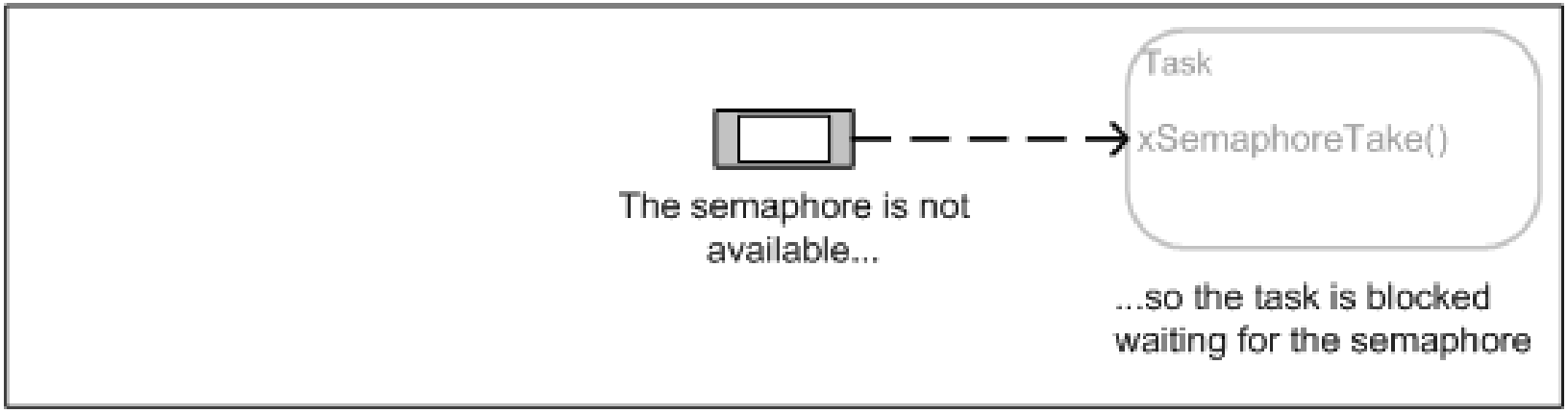
# FreeRTOS: Queue

- Wichtigste IPC Form bei FreeRTOS
- Senden von Nachrichten
  - zwischen Tasks
  - zwischen Interrupts und Tasks
- Meist: Thread-safe FIFO
- Methoden
  - `xQueueCreate()` [1, S. 162]
  - `xQueueReceive()` [1. Seite 186]
  - `xQueueSend()` [1, Seite 199]



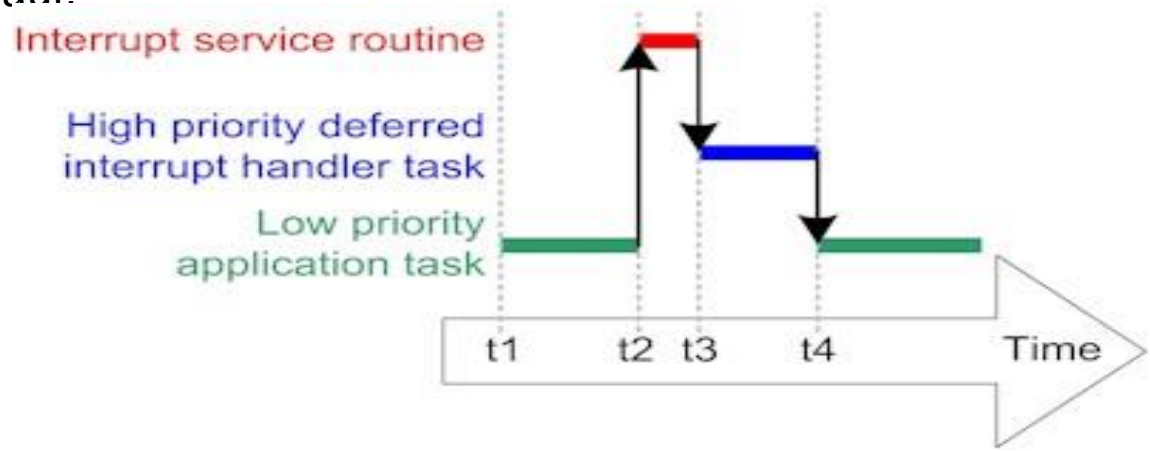
# (Binärer) Semaphor

- Für Synchronisation und Mutual Exclusion
- Abgrenzung zu Queue
  - Kein Interesse an Daten, die in der Queue liegen. Nur daran, ob etwas in der Queue ist oder nicht.
- Counting vs. Binary Semaphor
  - Counting Semaphor ähnelt einer Queue mit Länge > 1.
  - Binary Semaphor ähnelt einer Queue mit Länge 1
- Methoden für BinarySemaphore
  - `xSemaphoreCreateBinary()` [1, S. 212]
  - `xSemaphoreGive` [1, Seite 236]
  - `xSemaphoreTake` [1, Seite 244]



# Interrupts und FreeRTOS

- Interrupts können wie gewohnt implementiert werden
  - Konfiguration in der `setup`-Routine
- Die ISR sollte nicht zu lange sein!
  - Während Ausführung der ISR, arbeitet Scheduler nicht
- Oft besser: ***Deferred Interrupt Handling***
  - Eigentliche Reaktion auf Interruptereignis wird in FreeRTOS Task mit sehr hoher Priorität implementiert. Dieser Task ist die meiste Zeit im `BLOCKED` Zustand.
  - Bei Interruptereignis: ISR sorgt dafür, dass Task in `READY` Zustand wechselt. Das geht z.B. mit einer FreeRTOS *Notification*.
  - Nach Beenden der ISR, ist sofort wieder der Scheduler am Zug und ruft hoch-prioren Interrupt-Task auf.



# Betriebssystem: Vor- und Nachteil

- **Vorteile eines Betriebssysteme**
  - Modularität (Multitasking)
    - Große Software wird in kleinere, übersichtliche Tasks unterteilt.
    - Das Timing wird teilweise vom OS übernommen.
  - Benötigte Features (TCP, UDP, IO) sind manchmal bereits im Betriebssystem:
    - TCP/IP muss nicht selbst implementiert werden
    - Beispiel FreeRTOS+: https://www.freertos.org/FreeRTOS-Plus/index.html
  - Code relativ leicht übertragbar auf andere Mikrocontroller, falls diese gleiche OS unterstützen.

- **Nachteile eines Betriebssystem**
  - Einarbeitung kostet Zeit.
  - Hoher Overhead bzgl. Speicherverbrauch

- Kleine Mikrocontroller-Programme erstellt man besser ohne Betriebssystem!

# Übung heute

- Einfaches Kennenlernen von
  - Tasks
  - Prioritäten
  - Queues
  - Semaphore

# Quellenverzeichnis


[1] The FreeRTOS Reference Manual, https://www.freertos.org/wp-content/uploads/2018/07/FreeRTOS_Reference_Manual_V10.0.0.pdf (abgerufen am 07.07.2020)

[2] FreeRTOS Developer Docs, https://www.freertos.org/features.html (abgerufen am 07.07.2020)