

# Advanced Embedded Systems

*Zusammenfassung*

Joel von Rotz & Andreas Ming /  [Quelldateien](#)

## Inhaltsverzeichnis

---

<b>Entwicklung</b>	<b>4</b>
Cross-Development . . . . .	4
Integrated Development Environment . . . . .	4
Eclipse (Open Source IDE) . . . . .	4
Plugin System . . . . .	4
Workspace . . . . .	4
Begriffe . . . . .	4
File-Formate . . . . .	4
Intel Hex Binary .hex . . . .	4
Archive Library .a . . . .	4
Disassembly .dis . . . .	4
<b>Visualisierung</b>	<b>5</b>
MCUXpresso . . . . .	5
Task List View . . . . .	5
Queue List View . . . . .	5
FreeRTOS Trace Hooks . . . . .	5
Segger System View . . . . .	5
Viewer Hooks . . . . .	5
<b>Firmware</b>	<b>6</b>
Architektur . . . . .	6
Laufzeitmodelle . . . . .	6
Module (Baublöcke) . . . . .	7
Anforderung . . . . .	7
Schnittstelle (.h/.hpp) . . . . .	7
Device Konfigurieren & Erstellen . . . . .	8
Bibliotheken . . . . .	8
Archiv & Quelltexte . . . . .	8
Startup Code . . . . .	8
Runtime Library . . . . .	8
Standard-Bibliotheken . . . . .	8
<b>Systeme</b>	<b>9</b>
Transformierende Systeme . . . . .	9
Reaktive Systeme . . . . .	9
Interaktive Systeme . . . . .	9
Kombiniertes System . . . . .	9
<b>Architektur</b>	<b>9</b>
Systemaufbau . . . . .	9
<b>Mikrocontroller</b>	<b>10</b>
Rechnerarchitektur . . . . .	10
von Neumann Architektur . . . . .	10
Harvard Architektur . . . . .	10
System on Chip (SoC) . . . . .	10
ARM Cortex Familie . . . . .	10
Übersicht . . . . .	10

Interrupt Vektor Tabelle . . . . .	11
Cortex M4 (TinyK22) . . . . .	11
Cortex M0/M0+ (LPC845) . . . . .	11
SysTick . . . . .	11
SVCall . . . . .	12
<b>Cycle Counter</b>	<b>12</b>
MCULib . . . . .	12
DWT . . . . .	12
<b>FreeRTOS</b>	<b>12</b>
Echtzeit . . . . .	12
Harte & Weiche Echtzeit . . . . .	13
Periodische Echtzeit . . . . .	13
Architektur . . . . .	13
Philosophie . . . . .	13
Block Diagramm . . . . .	13
Kernel . . . . .	14
API . . . . .	14
Kontext Wechsel . . . . .	14
Interrupts . . . . .	14
ISR & FreeRTOS . . . . .	15
Kernel Interrupt Priorität . . . . .	15
ARM Cortex-M Interrupts . . . . .	15
Critical Sections . . . . .	15
Task ( <i>Threads</i> ) . . . . .	16
API . . . . .	16
Preemptive Priority Scheduling . . . . .	16
Time Slicing . . . . .	17
Suicide Task . . . . .	17
IDLE-Task . . . . .	17
Timer . . . . .	17
Erstellen . . . . .	17
Callback . . . . .	17
API . . . . .	17
Verwaltung . . . . .	17
Queue . . . . .	18
Erstellen & Löschen . . . . .	18
Anschauen & Entfernen . . . . .	18
Hinzufügen . . . . .	18
Queue Registry . . . . .	18
<b>Parallelität</b>	<b>18</b>
Reentrancy . . . . .	18
Interrupt (de-)aktivieren . . . . .	18
Critical Sections . . . . .	18
Priority Protokolle . . . . .	19
Priority Inversion (Problem) . . . . .	19
Priority Inheritance . . . . .	19
Deadlock . . . . .	20
Priority Ceiling . . . . .	20
Semaphore & Mutex . . . . .	20
Semaphore . . . . .	20
Mutex . . . . .	20
<b>Konzepte</b>	<b>21</b>
Synchronisation . . . . .	21
Kommunikation . . . . .	21
Realtime . . . . .	21
Gadfly / Polling . . . . .	22

Interrupt . . . . .	22
<b>Benutzerschnittstellen</b>	<b>22</b>
Hardware . . . . .	22
Design Prozess . . . . .	23
Überlegungen . . . . .	23
<b>Grafik</b>	<b>23</b>
Geschichte . . . . .	23
Model-View-Controller . . . . .	23

Wer das ankreuzelt, ist der Kreuzelkönig und hat das Kreuzchen ankreuzen kreuzlig verdient!



# Entwicklung

## Cross-Development

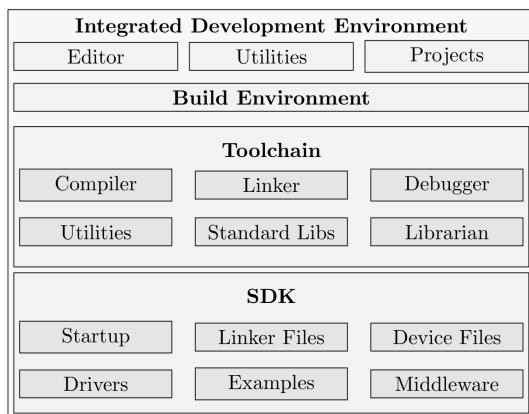
*Cross-Development* bedeutet die Entwicklung einer Firmware auf einem **Host** für einen **Target**. Grund dafür ist, dass das *embedded Target* nicht genügend Ressourcen (CPU Leistung, Speicher) für die direkte Entwicklung hat.

### Target & Host

**Target** (wofür): Zielsystem, für das man entwickelt.  
**Host** (womit): bezeichnet die Umgebung, auf der man die Entwicklung vornimmt.

## Integrated Development Environment

Eine IDE besteht aus vier Hauptteilen: IDE spezifische Funktionen, die Build Environment, die (GNU-)Toolchain und die SDK des entsprechenden Targets.



**Toolchain**: Kollektion von Tools wie Compiler, Linker, Debugger, etc. → einzelne Werkzeuge zum Zusammensetzen der Firmware

**Build Environment**: Steuert die Toolchain und den Übersetzungs-vorgang → *make, Makefiles*

**IDE**: “Fancy Editor”, beinhaltet Tools für bessere Produktivität, wendet Build Environment an → Intellisense, Workspace, Projekte

**SDK**: Software Development Kit → Treiber (UART, I<sup>2</sup>C, SPI,...), Beispiele (Board spezifisch), Projekt und Debugger Konfiguration (CMSIS-SVD, CMSIS-DAP,...), Device Files (Liste von Register und deren Adressen)

## Eclipse (Open Source IDE)

Plugin-basierter Editor → deckt mehrere Programmiersprachen und Environments ab.

- + Sehr modular (Plugin System), kann auf eigenen Workflow (ungefähr) angepasst werden
- + als IDE vereinfacht die Entwicklung
- Benötigt unnötig Rechenzeit beim Warten

### Geschichte

IDE wurde hauptsächlich von IBM (International Business Machines) auf der Code Basis vom *VisualAge IDE* in 2001 entwickelt und später mit Zusammenarbeit (Konsortium) von *Borland*, *QNX*, *Red Hat*, *SuSe* und andere entstand Eclipse.

→ Grund für Erfolg war das Plugin System und die Anpassbarkeit

## Plugin System

Haupt-Gimmick von Eclipse ist das Plugin System, welches die Erweiterung der bestehenden Entwicklungsumgebung durch weitere Werkzeuge wie zum Beispiel *Hex Editor* erlaubt.

→ Ermöglicht eine feinere Anpassung der Entwicklungsumgebung

## Workspace

Eclipse IDE arbeitet mit *Workspaces* → Kollektion von Projekten und Einstellungen (aktive Plugins, verwendete Version, spezifische Komplier Einstellungen).

### Warnung

Pro IDE Version ein eigener Workspace → wegen Versionskonflikte

## Begriffe

**Workspace** – Arbeitsplatz, Kollektion von Projekten, Einstellungen und aktive Plugins

**Views** – Einzelne Module/Fenster (z.B. *Variables* oder *FreeRTOS Task View*)

**Perspectives** – vordefinierte Gruppe & Platzierung von Views (z.B. Debug, Develop,...)

## File-Formate

### Intel Hex Binary .hex

Intel HEX besteht aus Zeilen mit ASCII-Text, welche von Newlines getrennt sind. Jede Textzeile enthält hexadezimale Zeichen, die mehrere Binärzahlen kodieren, welche Daten, Speicheradressen oder andere Werte darstellen können.

### Archive Library .a

Statische Bibliothek, welche während dem Linker-Prozess mit dem Programm kombiniert wird.

### Disassembly .dis

Die Disassembly beinhaltet Referenzen zum C-Programm in Bezug zu den Assembler-Befehlen.

## Visualisierung

```
#define configUSE_TRACE_FACILITY
```

- FreeRTOS Views in MCUXpresso
- Konsole-Ausgabe via uxTaskGetSystemState
- *Runtime Statistics* – Konsole-Ausgabe bei Context Switch mit configGENERATE\_RUN\_TIME\_STATS
- *Segger Real Time Transfer* – Datenausgabe während dem Betrieb
- FreeRTOS Trace Hooks
- Segger SystemView
- Percepio Tracealizer

## MCUXpresso

### Task List View

Tasks werden bei xTaskCreate automatisch dem *Task List View* in MCUXpresso hinzugefügt, solange configUSE\_TRACE\_FACILITY gesetzt wurde.

TCE#	Task Name	Task Handle	Task State	Priority	Stack Usage	Event Object	Runtime
> 1	App	0x20002d10	Blocked	2 (2)	204 B / 296 B		0x2 (0.0%)
> 2	IDLE	0x200030a0	Running	0 (0)	72 B / 792 B		0x67e2 (100.0%)
> 3	Tmr Svc	0x20003300	Suspended	5 (5)	220 B / 792 B	TmrQ (Rx)	0x0 (0.0%)

### Queue List View

Anzeige von **Queues, Semaphore & Mutex**

```
#define configQUEUE_REGISTRY_SIZE 10
```

①

① Anzahl mögliche Registry-Enträge

```
static xQueueHandle SQUEUE_Queue ;
Queue = xQueueCreate(SQUEUE_LENGTH , SQUEUE_ITEM_SIZE);
if(Queue == NULL) {
    for(;;) /* out of memory? */
}
vQueueAddToRegistry(Queue, "ShellQueue");
vQueueUnregisterQueue(Queue);
```

#	Queue Name	Address	Len...	Item Size	# Tx...	# R...	Queue Type
1	HostRxQueue	0x20002d80	2/512	0x1 (1 B)	0	0	Queue
	Head:	0x20002dd4					
	Read from:	0x20002fd3					
	Tail:	0x20002fd4					
	Write to:	0x20002dd6					
2	TmrQ	0x20003368	0/10	0x10 (16 B)	0	1	Queue
#	Address	Queue Data [...]	Queue Data [...]	Queue Data [BIN]	Queue Data [...]	Queue Data [...]	Queue Data [...]

## FreeRTOS Trace Hooks

Damit FreeRTOS Daten senden kann, müssen die Hooks zuerst instrumentiert werden.

```
#define configUSE_TRACE_HOOKS
```

①

① defaults zu configUSE\_PERCEPIO\_TRACE\_HOOKS

## Segger System View

```
#define configUSE_SEGGER_SYSTEM_VIEWER_HOOKS (1)

#if configUSE_SEGGER_SYSTEM_VIEWER_HOOKS
    McuSystemView_Init();
    SEGGER_SYSVIEW_Start();
#endif

#if configUSE_SEGGER_SYSTEM_VIEWER_HOOKS
    SEGGER_SYSVIEW_PrintfTarget("button %d ; event %d\n",
                                buttons, event);
    SEGGER_SYSVIEW_WarnfTarget("button %d ; event %d\n",
                                buttons, event);
    SEGGER_SYSVIEW_ErrorfTarget("button %d ; event %d\n",
                                buttons, event);
#endif
```

① Initialisierung

② Anwendung

### Viewer Hooks

```
#if configUSE_SEGGER_SYSTEM_VIEWER_HOOKS
    SEGGER_SYSVIEW_OnUserStart(unsigned MarkerId);
    SEGGER_SYSVIEW_MarkStart(unsigned MarkerId);
#endif

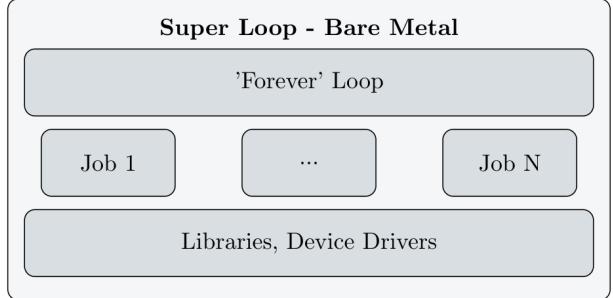
#if configUSE_SEGGER_SYSTEM_VIEWER_HOOKS
    SEGGER_SYSVIEW_OnUserStop(unsigned MarkerId);
    SEGGER_SYSVIEW_MarkStop(unsigned MarkerId);
#endif
```

## Firmware

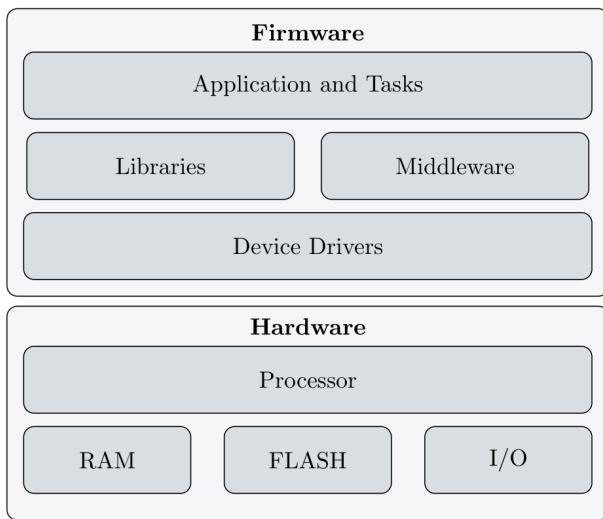
### Firmware versus Software

Firmware...

- wird direkt auf das Gerät einprogrammiert
- wenig veränderbar / ist firm (höhö)
- hat höhere Qualitätsanforderung → Aufwand ist hoch für Änderungen



## Architektur



Das Schichtenmodell stellt die Beziehung der Hard- & Firmware anhand Schichten und Modulen dar → Modulare Ansicht

Die **Interaktion zwischen HW & FW** verläuft über die **Device Drivers** (Gerätetreiber), welche die Ansteuerungen von Peripherien repräsentiert. Ein Treiber-Modul deckt meistens eine Art von Peripherie ab.

**Bibliotheken** erlauben die Wiederverwendbarkeit von Software

**Middleware** sind anwendungsneutrale Progammteile oder Programme (z.B. Datenbanken, Betriebssysteme oder Kommunikations-Stacks etwa für USB oder TCP/IP).

### Hardware-Abstraktion

Durch Verwendung von Gerätetreiber um auf die Hardware zuzugreifen, erhält man eine *Hardware-Abstraktion*. Diese macht die Software unabhängiger von der Hardware.

```
void main(void) {
    InitHardware();
    InitDriver();
    for(;;) {
        DoJob1();
        DoJob2();
        /*...*/
        DoJobN();
    }
}
```

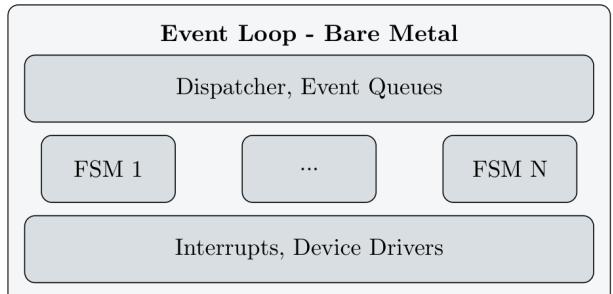
- + Sehr einfach und gut wartbar
- Jobs können länger dauern und somit andere Jobs **verzögern** → Latenz

Alternative wäre eine Finite State Machine (FSM):

```
for(;;) {
    DoJob1_part1();
    DoJob2();
    DoJob1_part2();
    DoJob3();
    DoJob1_part3();
    /*...*/
    DoJobN();
}
```

- + Latenz zwischen Jobs kann reduziert werden

## Loop mit Events



Endlosschleife wird mit einem *ereignisgesteuerten* Loop realisiert. Jobs werden via Hardware-Ereignisse veranlasst und direkt in Interrupts gemacht, oder über Queues oder einen anderen Benachrichtigungsmechanismus dem *Dispatcher* übergeben.

Mit FSM kann die Latzenzen von Events & Interrupts klein gehalten werden.

## Laufzeitmodelle

Beschreibt wie eine Firmware ausgeführt wird.

### Super Loop

```
void ButtonInterrupt (void) {
    QueueEvent( Button_Pressed );
}
```

①  
②

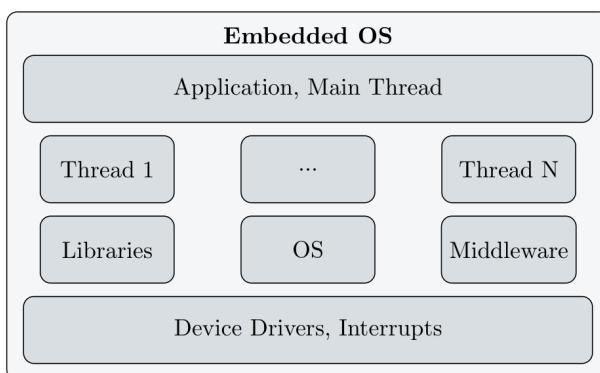
```
void main(void) {
    InitHardware ();
    InitDriversAndInterrupts ();
    for (;;) {
        GoToSleep (); /* wait for event */
        ProcessQueues ();
    }
}
```

③

- ① Interrupt wird kurz gehalten
- ② Event wird in die Queue gegeben
- ③ Queue wird verarbeitet

- + ‘Main’-Loop kann in einen stromsparenden Modus gehen und später durch Interrupts/Events aufgeweckt werden → Spart Energie und Rechenleistung
- Interrupts müssen klein gehalten werden
- ! Eine Mischform mit beiden Systemen ist möglich

## Betriebssystem / RTOS



Mit einem *Betriebssystem* werden Tasks *entkoppelt* und laufen in eigenen *Threads*, welche *quasi-gleichzeitig* ausgeführt werden. Dies erlaubt eine einfache Erweiterung durch neue Funktionen.

```
void mainTask(void) {
    CreateTask(sensorTask);
    CreateTask(otherTask);
    /* ... */
    for (;;) {
        /* do work */
    }
}

void main(void) {
    InitHardware ();
    InitDriversAndInterrupts ();
    CreateTask(mainTask);
    StartOS ();
```

①

- ① blockierende Funktion (daher kein while-Loop)

+ Skalierbarkeit

- Benötigt viel Ressourcen/Speicher
- Aufwand
- Deadlocks

## Module (Baublöcke)

Ziel der Modularisierung ist die Wiederverwendbarkeit bestehender Funktionen/Gerätetreiber, damit schneller neue Anwendungen und Produkte entwickelt werden können.

Vorsicht

Eine *Wiederverwendung* ist nur möglich mit einer guten *Modularisierung*.

## Anforderung

1. Interface  
Schnittstelle sollte einfach anzuwenden, erweiterbar und verständlich sein.
2. Synchronisation  
*Synchron*: Polling oder Gadfly  
*Asynchron*: Hardware Interrupts oder mit Events oder Callbacks
3. Organisation  
Die Quelltexte sollten einfach organisiert sein → Aufteilung in einzelne Dateien
4. Konfiguration  
Konfigurierbarkeit erlaubt es Hardware Schnittstellen oder Bibliotheken einzustellen oder anzupassen

## Schnittstelle (.h/.hpp)

1. Abstraktion  
Schnittstelle beschreibt **was** gemacht wird → Implementation wird nicht preisgegeben, aber dafür **Funktionalität**
2. Kapselung  
Daten können **indirekt** geändert oder abgefragt werden → *Setter & Getter*

Data Hiding

Nicht alle Informationen sollten sichtbar sein → nur Nötiges preisgeben! Uneingeschränkter Variablenzugriff ist zu vermeiden.

3. In sich geschlossen  
Self-contained → Schnittstelle beinhaltet alles, was nötig ist.

## ! Refactoring

Neu umgeschrieben oder geändert, ohne die eigentliche Funktionalität zu ändern → Verbesserung von Lesbarkeit und Wartbarkeit. Sollte durch konsequente Implementation von Anfang an vermieden werden.

## Device Konfigurieren & Erstellen

*Device Handle, Device Konfiguration, Device Erstellen, void pointer*

Analog zu C++ mit Klassen & Objekten gibt es in C **Device Handles**. Diese Handles werden zur **Identifikation/Unterscheidung** von Schnittstellen gleicher Art verwendet (z.B. `uart0, uart1, ...`).

Device Handle sind generische void-Zeiger, welche auf eine Speicherstelle mit den Informationen zeigt.

```
typedef void *LED_Device_t;
LED_Device_t led = NULL;
void LED_On(LED_Device_t led);
```

Mit Konfiguration-struct können bei der Initialisierung zusätzliche Einstellungen gemacht werden.

```
typedef struct {...} LED_Config_t;
LED_Config_t config = NULL;
void LED_GetConfig(LED_Config_t *config);           ①
config.color = BLUE;                                ②
LED_Device_t LED_Init(LED_Config_t *config);        ③
LED_Device_t LED_DeInit(LED_Device_t led);          ④
```

- ① Initialisierung Konfiguration → Defaultwerte zuweisen
- ② Spezifizierung Konfiguration → Spezifische Werte zuweisen
- ③ Erstellung *Device Handle* → Speicher allozieren & konfigurieren
- ④ Deinitialisierung des Device → Speicher freigeben & `handle = NULL`

- + Funktionen & Konfiguration können einfach hinzugefügt werden
- Speicherhandling :(

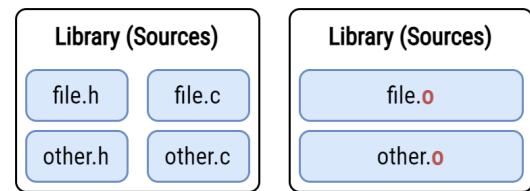
## Bibliotheken

- `<assert.h>` – assert-Makro
- `<math.h>` – Mathematik: `sin()`, `cos()`, ...
- `<setjmp.h>` – Sprung: `setjmp()`, `longjmp()`
- `<stdarg.h>` – Variable Argumente: `va_start()`, ...
- `<stdlib.h>` – Diverses: `malloc()`, `free()`, ...
- `<stdio.h>` – Ein-/Ausgabe: `printf()`, `scanf()`, ...
- `<string.h>` – String-Operationen: `strcpy()`, ...
- `<stdbool.h>` – Typ `bool`
- `<stdint.h>` – Integer Typen: `int32_t`, ...

## ⚠ Klein aber Klein

Embedded Systems sind oft funktionsumfänglich limitiert, daher sind oft Lokalisierung `<locale.h>` oder Zeitverwaltung `<time.h>` nicht unterstützt.

## Archiv & Quelltexte



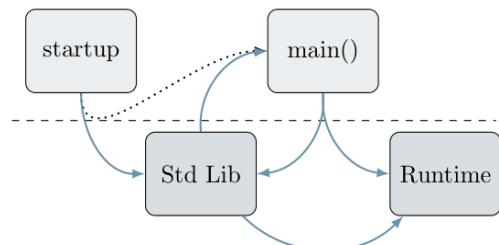
## Quelltexte

- + **Bibliotheken** in **Quelltextformat**, wie z.B. Open Source Biblios, dass diese **konfiguriert** werden können
- + Direkte Integration
- + höchste Transparenz
- benötigt Zeit um vollständig kompiliert
- grosse Einarbeitungszeit

## Archiv

- + Keine Kompilierung nötig
- + weniger falsches machbar
- Bibliothek muss zum System & Compiler passen
- keine Transparenz & keine Konfigurierbarkeit
- Einfluss auf Debugging (keine Debug Informationen)

## Startup Code



Der Startup Code ruft normalerweise `main()` *nicht* direkt auf, sondern über eine spezielle Initialisierungsfunktion wie `_start()` (wobei die Funktion Hersteller-abhängig) oder `__libc_init_array()` für C++.

## Runtime Library

*Runtime* Routinen sind vorgefertigte Programm-Snippets, um Operationen durch Software zu ermöglichen, welche in der Hardware nicht unterstützt werden. Beispiel sind Float-Operationen auf einem Controller ohne FPU.

→ [C-Laufzeitroutinen](#)

## Standard-Bibliotheken

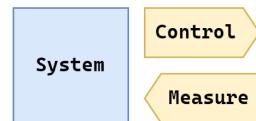
- **GNU Lib glibc**: Vollständige Bibliothek und GNU GPL Lizenz, deshalb für Embedded nicht verwendet
- **Newlib newlib**: Embedded-optimierte Standard Bibliothek

- **Newlib-nano** newlib-nano: auf Grösse optimiert gegenüber newlib. Oft langsamer, dafür sehr kleiner Speicherverbrauch
- **Proprietäre** Bibliotheken wie RedLib

### i Semihosting

Semihosting dient zur Verwendung von IO- und File-Funktionalität wie `printf()` oder `fopen()` mit einem Mikrocontroller, wobei alle diese Operationen auf dem Host ausgeführt werden.

- **none**: keine Callbacks implementiert → Anwendungspezifisch
- **nohost**: Callbacks sind leer implementiert
- **semihost**: Callbacks nutzen Semihosting

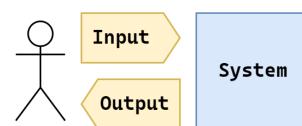


### Interaktive Systeme

Interaktive Systeme werden von Benutzer interagiert.

- **hohe Systemlast** → z.B. Auswertung Interaktion auf Benutzeroberfläche
- **optimiertes HMI** (Human-Machine-Interface) → wenn von Benutzer angewendet
- **'kurze' Antwortzeit**.

Beispiele: Ticket-Automat, Taschenrechner, Smart-Phone, Fernsehbedienung



## Systeme

### ! Was ist ein *embedded* System?

Ein Rechner (CPU, MCU, etc.) integriert in ein System. Für eine Aufgabe/Zweck optimiert und meistens **kein** normaler Computer! Meistens von aussen nicht direkt zugreifbar, anders als beim Computer.

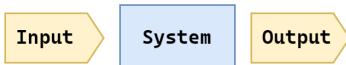
Anwendung: Echtzeitsystem, Wetterstation, Steuerung für Roboterarm, etc.

## Transformierende Systeme

Verarbeitet ein Eingabesignal (*Input*) und gibt ein Ausgabesignal (*Output*) aus. Wichtige Charakteristiken:

- **Verarbeitungsqualität** → effiziente Datenverarbeitung
- **Durchsatz** → kleine Latenz zwischen IO
- **optimierte Systemlast** → für die Aufgabe ausgelegt ist ; nicht überdimensioniert
- **optimierter Speicherverbrauch** → wenig Speicher ⇒ langsames System, daher effizienter Speichergebrauch

Beispiele: Verschlüsselung, Router, Noise Canceling, MP3/MPEG En-/Decoder



## Reaktive Systeme

Ein Reaktives System reagiert auf gemessene Werte, also von externen Events. Diese sind typisch **Echtzeitsysteme**.

- **kurze Reaktionszeit** garantieren → meist in Notfallsituationen verwendet
- in **Regelkreisen** auffindbar

Beispiele: Airbag, Roll-Over Detection, ABS, Brake Assistance, Engine Control, Motorsteuerung

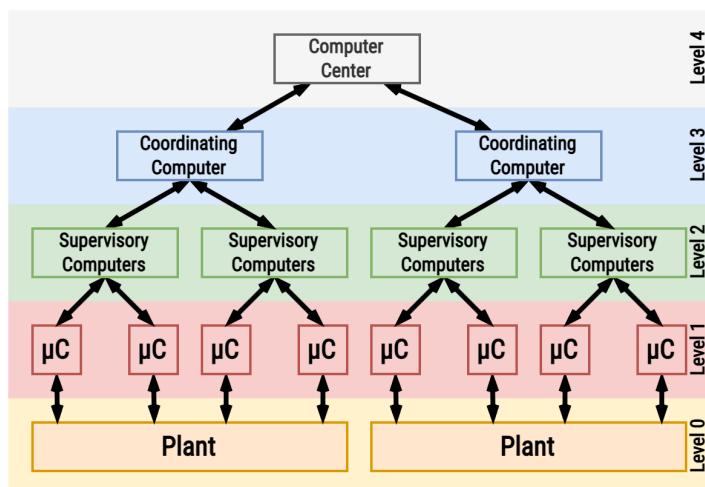
### Kombiniertes System

Ein kombiniertes System ist, wär hätte es gedacht, eine Kombination von den erwähnten Systemen und anderen.

Beispiele: Smartphone → interaktives Teilsystem für Homescreen- & App-Interaktionen, transformierendes Teilsystem für Audio-Decodierung für Musikhören und weiteren kleineren Teilsystemen.

## Architektur

### Systemaufbau



- ④ Production Scheduling (Planung & Verwaltung einer Produktion)
- ③ Production Control (Kontrolle von mehreren Prod.-Anlagen)
- ② Plant Supervisory (Kontrolle & Überprüfung über stärkeren Rechner)

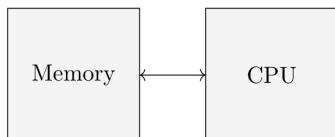
- ① Direct Control (Mikrocontroller)
- ② Field Level (Maschine, Pumpe, Roboter,...)

## Mikrocontroller

### Rechnerarchitektur

#### von Neumann Architektur

Gemeinsamer Speicher für Programm & Daten → einfacherer Aufbau, Speicher kann flexibel aufgeteilt werden und universell genutzt werden, **aber** begrenzte Leistung, da CPU nur auf Daten oder Programm zugreifen kann.



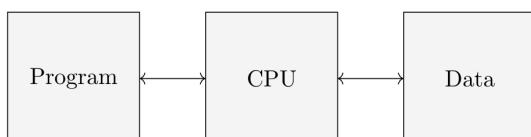
ADC/DAC & SRAM sind Kostentreiber!

#### ! Speicher sind teuer

Speicher sind teuer und werden in gewissen Fällen sogar weggelassen, bzw. ein externer Speicher ist nötig.  
Es gibt Lösungen wie Flashless MCU, XiP (execute in Place), PoP (Package on Package), HyperFlash (gecacheter, schnellerer Flash) und QSPI.

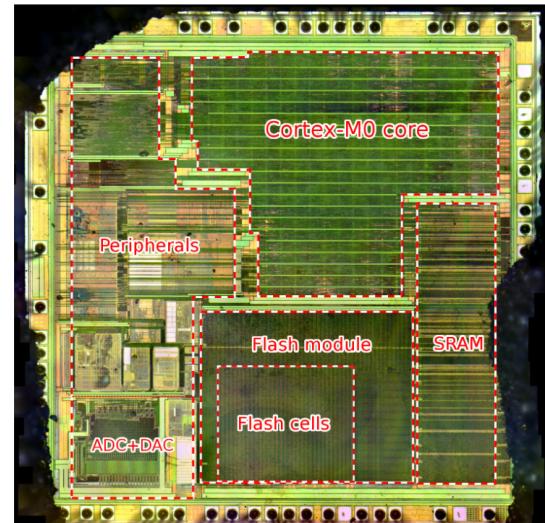
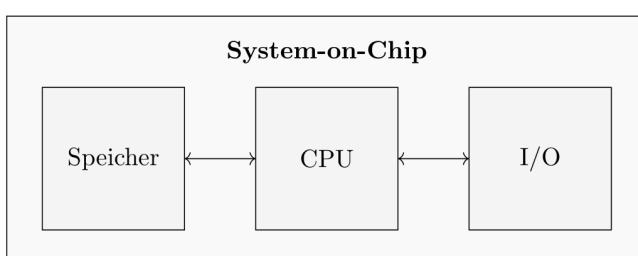
### Harvard Architektur

Separater Zugriff auf Daten- & Programmspeicher → höhere Geschwindigkeit und bessere Sicherheit durch Trennung, **aber** höhere Komplexität



### System on Chip (SoC)

Ein Baustein der einen Rechner mit möglichst wenigen externen Komponenten realisiert → reduziert Platz, Anzahl Bausteine und Kosten, **aber** verschiedene Varianten (über 1000) existieren (nicht immer verfügbar), zusätzliche Bus-Leitungen nötig und Taktraten des CPUs wurden höher (>1GHz), aber diese der RAM-Speicher sind bei 100MHz stehengeblieben (*Unterschiedliche Fertigungsprozesse*).



## ARM Cortex Familie

### i Übersicht ARM

- 1990 ARM inc. ⇒ 2016 gekauft von SoftBank ⇒ 2022 geplatzter Kauf von Nvidia
- Joint Venture: Acorn Computer + Apple + VLSI Technology
- IP Schmiede ⇒ STM, NXP, TI, Atmel, EM, ...
  - stellt Unterlagen zur Verfügung, aber keine Hardware
- ARM Produkte: Cortex 32bit/64bit RISC

### ARM Cortex-A Application

### ARM Cortex-R Realtime

### ARM Cortex-M Microcontroller

### Übersicht

⇒ Siehe letzte Seiten

Cortex	Mul	Div	DSP	Sat	FPU	TZ	CoreM	Arch
M0	1, 32	no	no	no	no	no	2.33	v6
M0+	1, 32	no	no	no	no	no	2.46	v6-M
M1	1, 32	yes	no	no	no	no	1.85	v6-M
M3	1	yes	no	part	no	no	3.34	v7-M
M4	1	yes	yes	yes	(f)	no	3.42	v7E-M
M7	1	yes	yes	yes	(f, d)	no	5.01	v7E-M
M23	1, 32	yes	no	no	no	yes	2.64	v8-Mbase
M33	1	yes	yes	yes	(f, d)	yes	4.02	v8-M

Annotations:

- Integ. Multiplikation (Mul)
- Division (Div)
- Saturation operationen / kein Overflow → ARn
- "200+100=255"
- andere Hardw.
- Performance-Score
- 82 - Zyklen, weniger Hardw.
- 1-Zyklus, mehr Hardw.
- Kosten

## Cortex M0/M0+ (LPC845)

Exception number	IRQ number	Vector	Offset
16+n	n	IRQn	0x40+4n
.	.	.	.
18	2	IRQ2	0x48
17	1	IRQ1	0x44
16	0	IRQ0	0x40
15	-1	SysTick, if implemented	0x3C
14	-2	PendSV	0x38
13		Reserved	0x2C
12		SVCall	
11	-5		
10			
9			
8			
7			
6			
5			
4			
3	-13	HardFault	0x10
2	-14	NMI	0x0C
1		Reset	0x08
		Initial SP value	0x04
			0x00

## Interrupt Vektor Tabelle

### Cortex M4 (TinyK22)

Exception number	IRQ number	Offset	Vector
16+n	n	0x0040+4n	IRQn
.	.	.	.
18	2	0x004C	IRQ2
17	1	0x0048	IRQ1
16	0	0x0044	IRQ0
15	-1	0x0040	Systick
14	-2	0x003C	PendSV
13		0x0038	Reserved
12			Reserved for Debug
11	-5	0x002C	SVCall
10			Reserved
9			
8			
7			
6	-10	0x0018	Usage fault
5	-11	0x0014	Bus fault
4	-12	0x0010	Memory management fault
3	-13	0x000C	Hard fault
2	-14	0x0008	NMI
1		0x0004	Reset
		0x0000	Initial SP value

## SysTick

SysTick ist ein 24Bit Timer Interrupt, der von ARM als Zeitbasis für Betriebssysteme vorgesehen ist. Typische Zeiten sind Perioden von **10ms** oder **1ms**. Durch den Tick Interrupt erhält das OS eine Gelegenheit für einen Kontext Wechsel, falls im *preemptive* Modus.

```
void vPortTickHandler(void) {
    portDISABLE_INTERRUPTS(); /* disable interrupts */
    if (xTaskIncrementTick() != pdFALSE) { /* increment
        → tick count */
        traceISR_EXIT_TO_SCHEDULER();
        taskYIELD();
    }
    portENABLE_INTERRUPTS(); /* enable interrupts */
}
```

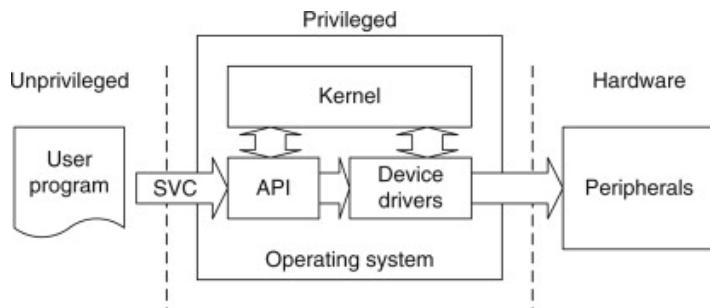
### SysTick

xTaskIncrementTick() gibt pdTRUE zurück, falls ein Kontext Wechsel stattfinden soll.

Im non-preemptive Modus ist der Rückgabewert immer pdFALSE.

## SVCall

Der **SVC** (*SuperVisor Call*) ist von ARM dafür vorgesehen um OS Kernel oder Gerätetreiberfunktionen aufzurufen. Die svCall Exception wird durch die `svc <number>` Instruktion ausgelöst, welche über ein zusätzliches Argument (*Zahl*) verfügt.



Name	Value	Description
> fpdscr	0x0	Floating Point Default Status Co...
> mvfr0	0x10110021	Media and FP Feature Register 0
> mvfr1	0x11000011	Media and FP Feature Register 1
> FPU SP registers		FPU single precision registers
> FPU DP registers		FPU double precision registers
DWT Registers		Data Watchpoint and Trace Unit...
> cycles	0x3e7b	Cycle Count Register
> cycleDelta	0x9f	Cycle Delta
Name : cycleDelta		
Hex:0x9f		
Decimal:159		
Octal:237		
Binary:10011011		

Zählt totale Zyklen (`cycles`) und seit letztem Debug-Step-/Breakpoint (`cycleDelta`)

## FreeRTOS

FreeRTOS ist ein open source Echtzeit-Betriebssystem für Embedded Systems. Zu Beginn war FreeRTOS unter der GNU Public License (GPL) (GNU Lesser Public License (LGPL)) Lizenz erhältlich, was eine Nutzung in kommerziellen Projekten trotz GPL ermöglichte. Nach Amazons Übernahme wurde die Lizenz zur MIT-Lizenz.

### Echtzeit

**das richtige Resultat** – Die Verarbeitung der Eingabe und die folgende Ausgabe muss korrekt sein (keine Fehleralarme).

*Ein Airbag sollte nur bei einem schweren Aufprall ausgelöst werden.*

**Zur richtigen Zeit** – Je nach System muss das Zeitfenster eingehalten werden.

*Der Airbag hat ein enges Zeitfenster: nicht zu früh, wegen Massenbewegung, oder zu spät, um keine Verletzungen zu verursachen.*

**Systemlast-Unabhängig** – Egal was das System sonst tut, muss es das Richtige zur richtigen Zeit tun.

*Wenn der Airbag-Computer mit n Airbags zu tun hat, muss es trotzdem in den vorgegebenen Grenzen reagieren.*

**Deterministische & vorhersehbare Weise** – Das System muss unter gleichen Ausgangsbedingungen immer gleich reagieren und die Reaktionszeit muss berechenbar sein → zu jedem Zeitpunkt vorhersehbar & bekannt

### Was ist Echtzeit?

Ein Computer ist als Echtzeitsystem klassifiziert, wenn er auf externe Ereignisse in der **echten** Welt reagieren kann: mit dem **richtigen Resultat**, zur **richtigen Zeit**, unabhängig der Systemlast, auf eine deterministische und vorhersehbare Weise.

- **Absolute** Rechtzeitigkeit – Absoluter Zeitpunkt (z.B. jeden Tag 05:30 ± 1 Minute)
- **Relative** Rechtzeitigkeit – Relative Zeit nach Ereignis (z.B. 5 Minuten (± 10s) nach Einschalten wieder ausschalten)

## Cycle Counter

### MCULib

```
#include "cycles.h"          ①
CCOUNTER_START();           ②
/* do stuff */
CCOUNTER_STOP();
```

- ① Inkludierung  
② Anwendung

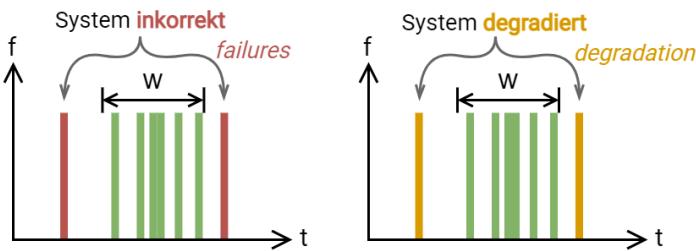
```
CCOUNTER_START();
d0 = sqrt(34.5);
CCOUNTER_STOP();
Cycles_LogTime("square root");

nop test: delta: 11, overhead: 0 cycles: 11; time: 0 us
square root: delta: 3275, overhead: 0 cycles: 3275; time: 156 us
```

## DWT

Data Watchpoint and Trace Unit

## Harte & Weiche Echtzeit



- **Harte** Echtzeit (links) – Zeitbedingung einhalten (innerhalb Zeitfenster  $w$ ). **Beispiel** Airbag soll 20ms nach Aufpralldetektion ausgelöst werden.
- **Weiche** Echtzeit (rechts) – Immer noch in Ordnung, wenn Zeitbedingung nicht eingehalten. **Beispiel** Video Encoder wiedergibt mit Framerate 25 F/s. Framerate darf nicht unter 10 F/s sein und in 10% der Zeit Framerate unter 25 F/s → System ist immer noch als korrekt angesehen.

## Periodische Echtzeit

Echtzeitsystem müssen das richtige Resultat zur richtigen Zeit liefern.

Realität alles parallel → Computer/Recheneinheit arbeitet seriell  
→ *quasi-parallel* mehrere Dinge erledigen

Folgendes Beispiel zeigt eine Möglichkeit:

```
for (;;) {
    if (time ==530) { /* start at 05:30 am */
        StartIrrigation (); /* turn relay on */
    }
    if (time >530 && time <535) {
        /* irrigate from 05:30 am to 05:35 am */
        /* control the water pump , needs to
           be called every 10 ms: */
        ControlIrrigation ();
        /* wait 5 ms (additional 5 ms will be added) */
        WaitMs (5);
    }
    MeasureHumidity ();
    /* needs to be called every 5 ms */
    WaitMs (5);
    if (time ==535) { /* stop at 05:35 am */
        StopIrrigation (); /* turn relay off */
    }
}
```

## Architektur

### Philosophie

#### ! Preemptives & Kooperatives Scheduling

**Preemptive** – Es läuft immer der Task mit der höchsten Priorität. Tasks mit der gleichen Priorität teilen sich die

Rechenzeit (fully preemptive with round robin time slicing).

**Kooperativ** – Ein Kontext Switch findet nur statt, wenn ein Task blockiert oder explizit ein Yield aufruft. Ein 'Yield' ist die Aufforderung an den Kernel, einen Kontext Wechsel vorzunehmen.

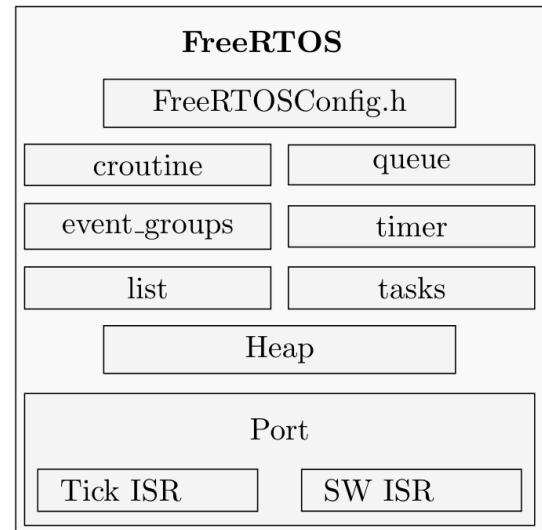
#### ! Benötigte Interrupts

Damit das Betriebssystem korrekt läuft, werden zwei Interrupts benötigt:

**Tick Interrupt** – periodischer Interrupt, welcher einen Kontext Switch veranlasst (Preemption) (SysTick)

**Software Interrupt** – Interrupt, welcher vom Kernel oder von der Anwendung ausgelöst werden kann (svcall, PendableSrvReq)

## Block Diagramm



Implementiert...

- FreeRTOSConfig.h – ... Makros zur Konfiguration des Betriebssystems
- croutine – ... Co-Routinen sind Mini-Threads, welche den Stackspeicher untereinander teilen
- event\_groups – ... Event Flags zur Signalisation von Events
- list – ... die Listenverwaltung für z.B. wartende Objekte
- queue – ... Queues, Mutex, Semaphore
- timer – ... den Software Timer
- task – ... den Scheduler
- heap – ... Speicherverwaltung des Heap Speichers zur Bereitstellung des Stackspeichers für die Tasks
- Port – ... spezifischen und Architektur-abhängigen Teil des Betriebssystems

## Kernel

## API

### vTaskStartScheduler()

```
void vTaskStartScheduler(void);           ①
void vTaskEndScheduler(void);           ②
```

- ① Setzt den Scheduler von *Init* in den *Running* Zustand  
 ② Beendet den Scheduler und springt zum Aufruf von vTaskStartScheduler

#### vTaskEndScheduler

Wird der Scheduler beendet, werden `setjmp()` und `longjmp()` verwendet, was nicht in jedem Port implementiert ist. Zudem werden die Ressourcen der Tasks nicht automatisch freigegeben!

### vTaskSuspendAll()

```
void vTaskSuspendAll(void);
```

Versetzt den Kernel von *active* in den *suspended* Zustand → Interrupts sind noch aktiv, aber der Tick Interrupt löst keinen Kontext Switch mehr aus.

! Kann mehrfach / verschachtelt werden

### vTaskResumeAll()

```
portBASE_TYPE vTaskResumeAll(void);
```

`pdTRUE` – Kernel *resumed* → Kontext Switch veranlasst `pdFALSE` – Kernel *suspended*, da `vTaskSuspendAll` mehrmals aufgerufen wurde.

### taskENTER\_CRITICAL(), taskEXIT\_CRITICAL()

Um *Critical Section* mit dem Kernel zu erstellen, steht folgendes API zur Verfügung:

```
void taskENTER_CRITICAL(void);
void taskEXIT_CRITICAL(void);

void vPortEnterCritical(void) {
    portDISABLE_INTERRUPTS();
    uxCriticalNesting++;
}

void vPortExitCritical(void) {
    uxCriticalNesting--;
    if (uxCriticalNesting == 0) {
```

```
        portENABLE_INTERRUPTS();
    }
}
```

#### Keine FreeRTOS API in Critical Sections

Innerhalb einer Critical Section sollten keine FreeRTOS API Aufrufe getätigt werden.

### taskDISABLE\_INTERRUPTS(), taskENABLE\_INTERRUPTS()

Folgendes API steht zum Ein- und Ausschalten von Interrupts zur Verfügung:

```
void taskDISABLE_INTERRUPTS(void);
void taskENABLE_INTERRUPTS(void);
```

#### Interrupts Cortex M0+ & M4

- Cortex M4 besitzt ein `BASEPRI`-Register, welches Interrupts ab dem gegeben Wert deaktiviert.
- Cortex M0+ hat dies nicht und *deaktiviert daher alle Interrupts* über `PRIMASK`.

### taskYIELD()

```
#define taskYIELD() portYIELD()
```

Um einen Kontext Switch in einem Task auszulösen, kann `taskYIELD()` verwendet werden. Dies ist auch im *kooperativen* Modus möglich.

#### Priorität

Wenn *yielded* wird,

1. wird entweder ein Task im *Ready* Zustand mit der höchsten Priorität
2. oder wenn der Task selbst der höchste ist, gleich wieder zu ihm zurück.

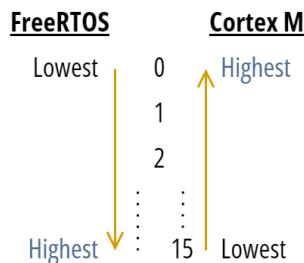
### Kontext Wechsel

Ein Kontextwechsel kann nur stattfinden, wenn der Scheduler oder das Betriebssystem läuft:

1. via Tick-Interrupt → Time Slicing oder Preemption ; synchrones Scheduling
2. via Anwendung (*Trap*, *SysCall*, *SystemCall*) → indirektes/asyncrones Scheduling, z.B. durch Senden einer Meldung wird ein dringlicher Task aktiviert
3. via Anwendung direkt → Yield

### Interrupts

Prioritäten:

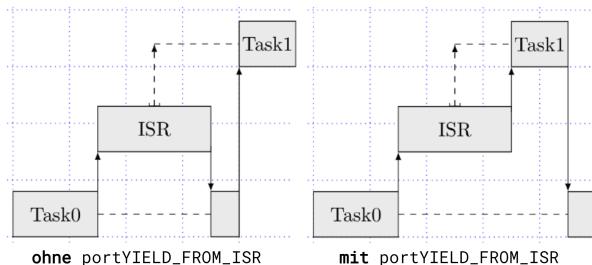


## ISR & FreeRTOS

Aus Interrupt Service Routinen darf man nur OS Funktionen benutzen, welche mit `FromISR` enden. Diese sind **nicht blockierend**.

```
void vTimerISR(void) {
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;
    xSemaphoreGiveFromISR(xSemaphore ,
    &xHigherPriorityTaskWoken);
    portYIELD_FROM_ISR(xHigherPriorityTaskWoken); ①
}
```

① `xHigherPriorityTaskWoken=pdTRUE` → höher priorisierter Task wurde aktiviert



## Kernel Interrupt Priorität

In FreeRTOS läuft der Kernel, und somit seine Interrupts, mit der tiefstmöglichen Interrupt Dringlichkeit → kein Einfluss auf Anwendungsspezifische Interrupts

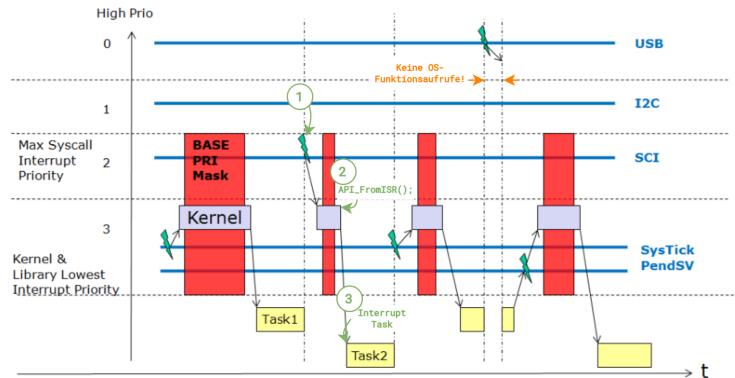
```
#define configPRIO_BITS 4
#define configLIBRARY_LOWEST_INTERRUPT_PRIORITY 15
#define configKERNEL_INTERRUPT_PRIORITY \
    (configLIBRARY_LOWEST_INTERRUPT_PRIORITY<<(8- \
    & configPRIO_BITS))
```

### ! Anzahl Interrupts

FreeRTOS verwendet 2-3 Interrupts: SysTick dient als Zeitbasis, PendSV für Kontext-Wechsel und SVCall auf dem Cortex-M3/4/7 um den Scheduler zu starten.

## ARM Cortex-M Interrupts

Immer wenn eine *Critical Section* erstellt wird (Roter Bereich), wird der Wert 0x80 in das BASEPRI geschrieben und so die entsprechenden Interrupts ausgeschalten. Um die *Critical Section* möglichst kurz zu halten, wird in diesem Bereich nicht die eigentliche Funktion des Interrupts ausgeführt, sondern ein Task gescheduled, der diesen ausführt.



### ! Wichtig

Auf einem Cortex-M0 maskiert RTOS alle Interrupts für seine *Critical Sections*.

Auf dem Cortex-M3/4/7 werden nur die Interrupts maskiert mit Priorität numerisch gleich oder grösser als

```
configMAX_SYSCALL_INTERRUPT_PRIORITY <prio>
```

Interrupts welche von der *Critical Section* nicht beeinflusst werden (Priorität numerisch kleiner `configMAX_SYSCALL_INTERRUPT_PRIORITY`), dürfen auch keine API-Aufrufe durchführen.

## Critical Sections

```
#define portDISABLE_INTERRUPTS() \
    portSET_INTERRUPT_MASK() ①
#define portENABLE_INTERRUPTS() \
    portCLEAR_INTERRUPT_MASK()
```

```
#define portENTER_CRITICAL() \
    vPortEnterCritical() ②
#define portEXIT_CRITICAL() \
    vPortExitCritical()
```

```
#define taskENTER_CRITICAL() \
    portENTER_CRITICAL() ③
#define taskEXIT_CRITICAL() \
    portEXIT_CRITICAL()
```

① Nicht verschachtelbar

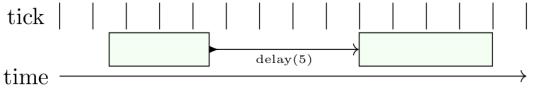
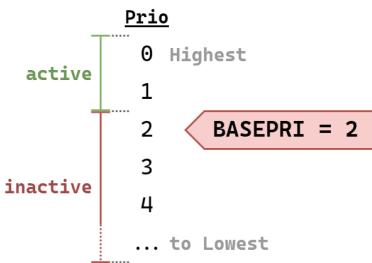
② Verschachtelbar

③ für Task Critical Sections

*Critical Sections* dienen zum Blockieren von ISR, wenn etwas wichtiges gemacht werden muss (Beispiel: `vPortTickHandler` verwendet CS um den Tick sicher zu inkrementieren und die Task-Liste zu überprüfen).

```
#define configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY (5)
#define configMAX_SYSCALL_INTERRUPT_PRIORITY \
    (configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY<<(8- \
    & configPRIO_BITS))
```

- M4 verwendet diesen Wert für das BASEPRI-Register



```
static void BlinkyTask(void * param ) {  
    for (;;) {  
        LED_Neg ();  
        vTaskDelay( pdMS_TO_TICKS (5));  
    }  
}
```

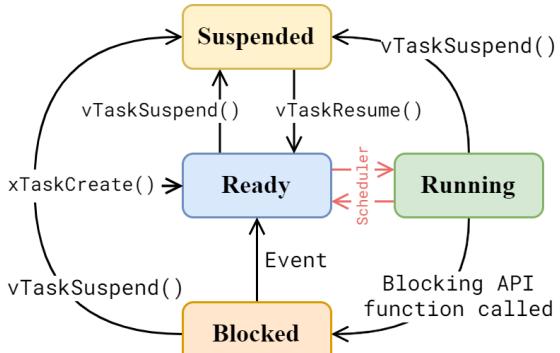
## Task (Threads)

```
static void MyTask(void *params) {  
    (void)params;  
    for (;;) {  
        /* do the work here ... */  
    } /* for */  
    /* never return */  
}
```

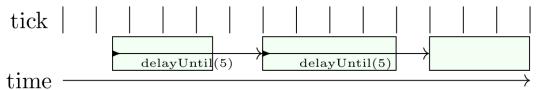
1

① *Ignore value of params* – Unterdrückt Compiler-Warnung

## ! Task-States



### vTaskDelayUntil()



```
static void BlinkyTask(void * pvParameters ) {
    TickType_t xLastWakeTime = xTaskGetTickCount ();
    for (;;) {
        LED_Neg ();
        vTaskDelayUntil (&xLastWakeTime , pdMS_TO_TICKS (5));
    }
}
```

### vTaskSuspend()

```
void vTaskSuspend ( TaskHandle_t xTaskToSuspend );
vTaskSuspend(NULL);
vTaskSuspend(blinkyTaskHandle);
```

- ① suspendiert den Task selber
- ② suspendiert einen anderen Task

### vTaskResume()

```
void vTaskResume(TaskHandle_t xTaskToResume);
```

Aktiviert den suspendierten Task.

API

## xTaskCreate

```
 BaseType_t res;
TaskHandle_t taskHndl;

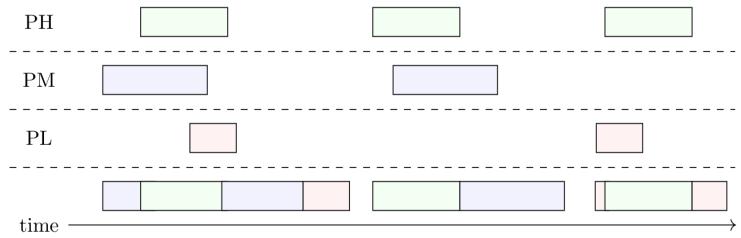
res = xTaskCreate (BlinkyTask , /*function*/
 "Blinky" , /* Kernel awareness name */
 500/ sizeof( StackType_t ), /* stack size */
 (void *)NULL , /* task parameter */
 tskIDLE_PRIORITY +1, /* priority */
 &taskHndl /* handle */

);

if (res != pdPASS) {
/* error handling here */
}
```

### vTaskDelay()

## Preemptive Priority Scheduling

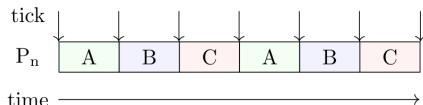


```
#define configUSE_PREEMPTION (1)  
#define configUSE_PREEMPTION (0)
```

① **Preemptive:** Der Scheduler kann Rechenzeit von einem laufenden Task wegnehmen.

② Cooperative: Der Task behält die CPU oder Rechenzeit, bis er selber die Kontrolle an den Scheduler abgibt (`taskYIELD();`)

## Time Slicing



```
#define configUSE_TIME_SLICING (1) //default
```

Sind mehrere Task mit der gleichen Priorität im *Ready* Zustand, so wird die Rechenzeit für jeden Task aufgeteilt und die Tasks im *round-robin* Stil abgearbeitet.

## Suicide Task

```
static void SuicideTask (void *params) {
    (void)params;
    /* ... do the work here ... */
    vTaskDelete(NULL); /* killing myself */
    /* won't get here as I'm dead ;-) */
}
```

## IDLE-Task

Wird der Scheduler mit `vTaskStartScheduler()` gestartet, wird zusätzlich der IDLE-Task aktiviert mit der Priorität `tskIDLE_PRIORITY` (tiefste Task Priorität) und einem Stackspeicher von `configMINIMAL_STACK_SIZE`.

Dieser dient als Ausläufer, falls keine anderen Tasks *Ready* sind.

### ! Idle Hook

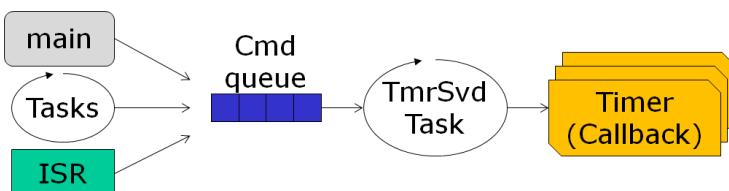
Der IDLE Task führt den *Idle Hook* aus, welche von der Anwendung definierte Aufgaben übernehmen kann → z.B. *Low Power Modus*

### i Idle Yielding

```
#define configIDLE_SHOULD_YIELD
```

Bestimmt bei einem preemptiven Scheduler, ob IDLE Task gleich die Kontrolle übergibt.

## Timer



```
#define configUSE_TIMERS (1)
#define configTIMER_QUEUE_LENGTH (3)
#define configTIMER_TASK_PRIORITY \
```

```
(configMAX_PRIORITIES-1)
#define configTIMER_TASK_STACK_DEPTH \
(configMINIMAL_STACK_SIZE)
```

- ① aktiviert Timer-Funktion und den *Timer Service Daemon*
- ② Maximale Anzahl gleichzeitige Task Kommandos an den Timer Deamon
- ③ Hohe Priorität

## Erstellen

```
TimerHandle_t xTimerCreate(
    const char *const pcTimerName,
    const TickType_t xTimerPeriodInTicks,
    const UBaseType_t uxAutoReload,
    void *const pvTimerID,
    TimerCallbackFunction_t pxCallbackFunction);
```

- ① Namen u.a. für Debugger
- ② Periode in Ticks
- ③ pdTrue = Wiederkehrend, pdFALSE = One-Shot
- ④ Identifikation oder Zeiger auf Daten

## Callback

```
static void vTimerCallback(xTimerHandle pxTimer) {
    /* ... */
}
```

## API

```
BaseType_t xTimerStart(TimerHandle_t xTimer, TickType_t
    ↳ xTicksToWait);
BaseType_t xTimerStop(...);
BaseType_t xTimerReset(...);
BaseType_t xTimerDelete(...);
```

Diese Befehle werden in die *Timer-Queue* gegeben. `xTicksToWait` entspricht der Timeout-Funktionalität, falls die Queue voll ist.

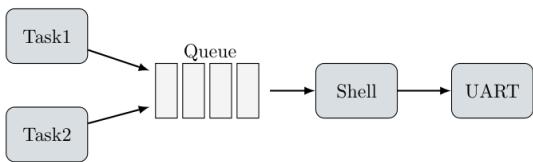
### i Interrupt

Alle API-Funktionen stehen auch mit `FromISR` zur Verfügung.

## Verwaltung

```
BaseType_t xTimerIsTimerActive(TimerHandle_t xTimer);
BaseType_t xTimerChangePeriod(TimerHandle_t xTimer,
    TickType_t xNewPeriod,
    TickType_t xTicksToWait);
TaskHandle_t xTimerGetTimerDaemonTaskHandle(void);
```

## Queue



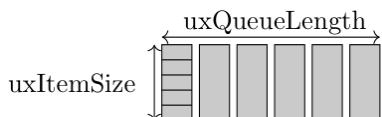
```

const void * pvItemToQueue,
portTickType xTicksToWait
);
  
```

- ① FIFO  
② LIFO

Returnwerte: pdTRUE, errQUEUE\_FULL

## Erstellen & Löschen



```

xQueueHandle xQueueCreate(
    unsigned portBASE_TYPE uxQueueLength,
    unsigned portBASE_TYPE uxItemSize
);
void vQueueDelete(xQueueHandle xQueue);
  
```

- ① Device Handle auf NULL prüfen!

## Anschauen & Entfernen

```

BaseType_t xQueueReset(xQueueHandle xQueue);
BaseType_t xQueuePeek(
    xQueueHandle xQueue,
    void * pvBuffer,
    portTickType xTicksToWait
);
BaseType_t xQueueReceive(
    xQueueHandle xQueue,
    void * pvBuffer,
    portTickType xTicksToWait
);
  
```

- ① Entfernt Element nicht  
② mit portMAX\_DELAY kann sehr lange gewartet werden (aber nicht unendlich lang!)  
③ xQueueReceive ist eine blockierende Funktion, es wird also darauf gewartet, dass ein Eintrag in die Queue gemacht wird.

## Hinzufügen

### ! Wichtig

Elemente werden *by value* in die Queue kopiert.

## Queue Registry

Zur Identifikation beim Debugging, können Queues mit entsprechendem Namen in ein Registry registriert werden

```

#define configQUEUE_REGISTRY_SIZE <number>
  
```

①

```

void vQueueAddToRegistry(
    QueueHandle_t xQueue,
    const char *pcQueueName );
void vQueueUnregisterQueue(
    QueueHandle_t xQueue );
  
```

②

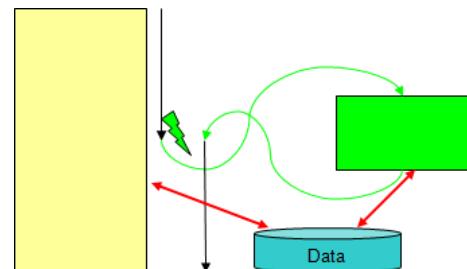
③

- ① Maximale Anzahl an Einträgen  
② Queue registrieren  
③ Queue aus Register löschen

## Parallelität

### Reentrancy

Jede Funktion, welche von einem unterbrechenden Prozess (ISR) verwendet wird, muss **reentrant** (wiedereintretbar) sein.



- Greifen zwei Prozesse (z.B. ISR & Haupt) lesend zu → kein Problem
- Einer liest, der andere schreibt oder beide schreiben → Problem

### Interrupt (de-)aktivieren

⊕ Einfach

⊖ Interrupt-Latenz

### Critical Sections

```

#define CriticalVariable() \
    uint8_t cpuSR
  
```

①

1. definiert lokale Variable cpuSR für Sicherung des aktuellen Interrupt-Zustandes

```
#define EnterCriticalSection() \
do { \
    __asm( \
        "mrs r0, PRIMASK \n" \
        "cpsid i \n" \
        "strb r0, cpuSR \n" \
    ); \
} while(0)
```

①  
②  
③

1. PRIMASK wird in R0 abgespeichert
2. Interrupts deaktivieren
3. R0 in cpuSR abspeichern

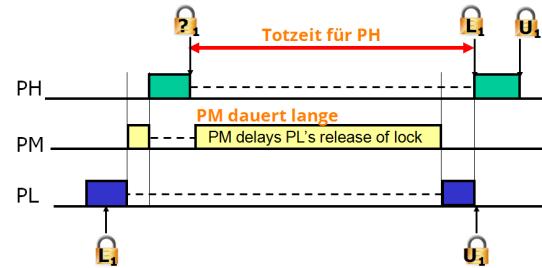
```
#define ExitCriticalSection() \
do { \
    __asm( \
        "ldrb r0, cpuSR \n" \
        "msr PRIMASK,r0 \n" \
    ); \
} while(0)
```

①  
②

1. Inhalt von cpuSR in R0 laden
2. R0 nach PRIMASK kopieren.

**+** Kann verschachtelt werden

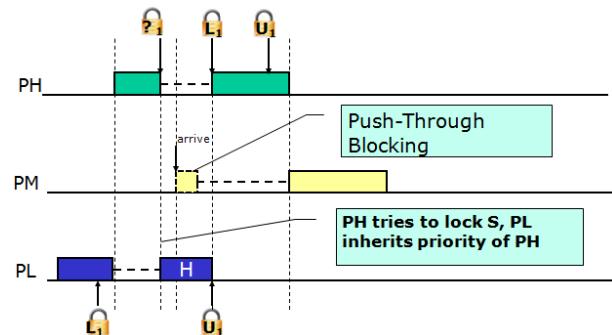
**+** cpuSR verhindert das vor ausgeschaltete Interrupts eingeschaltet werden (merkt sich vorheriger Status)



— Prioritätsproblem

### Priority Inheritance

Die Priorität des Tasks, welcher den Lock zu erlangen versucht, vererbt seine Priorität an den Task, welcher den Lock hält. Sobald auf den Lock angefragt wird, wird die Priorität vererbt und der Höhere Task unterbrochen.



Beispiel: *McuLib*

```
void doCounting(void) {
    McuCriticalSection_CriticalVariable(); // uint8_t cpuSR

    McuCriticalSection_EnterCriticalSection(); // PRIMASK -> cpuSR
    if (cntr < 100) cntr++;
    print(cntr);
    McuCriticalSection_ExitCriticalSection(); // cpuSR -> PRIMASK
}
```

### Priority Protokolle

Beim Zugriff Lock einer Ressource gilt:

Lock():  $?_x \rightarrow L_x$

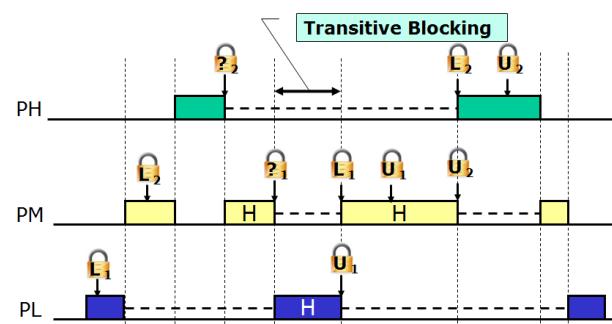
Lock erhalten:  $L_x$

Unlock():  $U_x$

### Priority Inversion (Problem)

PH wartet auf Lockfreigabe von PL, welcher von PM unterbrochen wurde → PM hat im Moment noch eine höhere Priorität als PL

Bei einer **verschachtelung von Locks**, kann durch einen zweiten involvierten Lock die Priorität auch auf einen dritten Task vererbt werden.



**+** Löst das Problem der Priority Inheritance

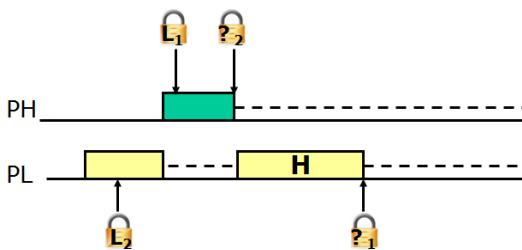
— mehr Rechenzeit

— löst Deadlocks nicht

### Rechenzeit

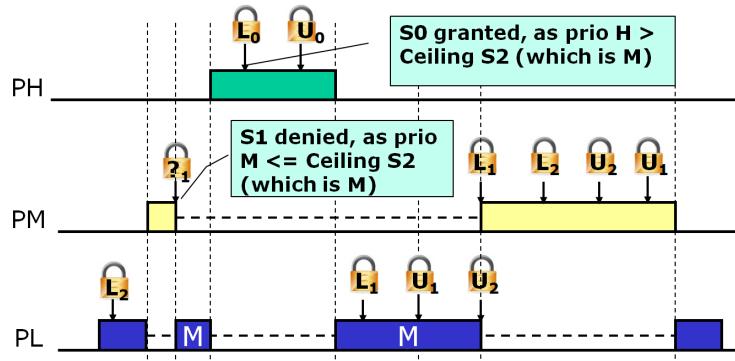
Bei den meisten Betriebssystemen kann zwischen vererbenden und nicht vererbenden Locks gewählt werden, um die Rechenzeit zu optimieren.

## Deadlock



→ Lösung *Priority Ceiling*

## Priority Ceiling



$S_0 : H \quad S_1 : M \quad S_2 : M$

Der *Ceiling*-Wert pro Ressource ist statisch und entspricht der höchsten Priorität eines Tasks, welche die Ressource nutzt.

Regeln der *Priority Ceiling*

1. **Normales Scheduling** – Solange ein Task  $T_i$  nicht eine seiner Critical Sections betreten will, kann er einen Task mit tieferer Priorität unterbrechen
2. **Deadlock-Fix** – Wenn ein Task  $T_i$  versucht, eine seiner Critical Sections zu betreten wird er suspendiert ausser seine Priorität ist höher ( $>$ ) als die Ceiling Priorität aller momentan von anderen Tasks als  $T_i$  gehaltenen Semaphoren
3. **Priority Inheritance** – Wenn ein Task  $T_i$  deshalb seine Critical Section nicht betreten kann, dann blockiert der Task  $T_j$ , welche die Semaphore hält, den Task  $T_i$ . Dabei vererbt der Task  $T_i$  seine eigene Priorität an den Task  $T_j$

## Semaphore & Mutex

Bei gemeinsamer Ressourcen-Nutzung von zwei verschiedenen Tasks.

### ! Unterschied Semaphore & Mutex

Beide sind sehr ähnlich, ausser dass Mutex über die *Priority Inheritance* verfügt.

Ebenfalls können **Semaphoren** zur **Synchronisation** von Tasks verwendet werden (Skript s.200).

### ! Queues ohne Daten

Semaphoren & Mutex basieren auf der Queue API, da diese bereits **reentrant** sind. Die Länge dieser Queue entspricht `semSEMAPHORE_QUEUE_ITEM_LENGTH` (Grösse 0).

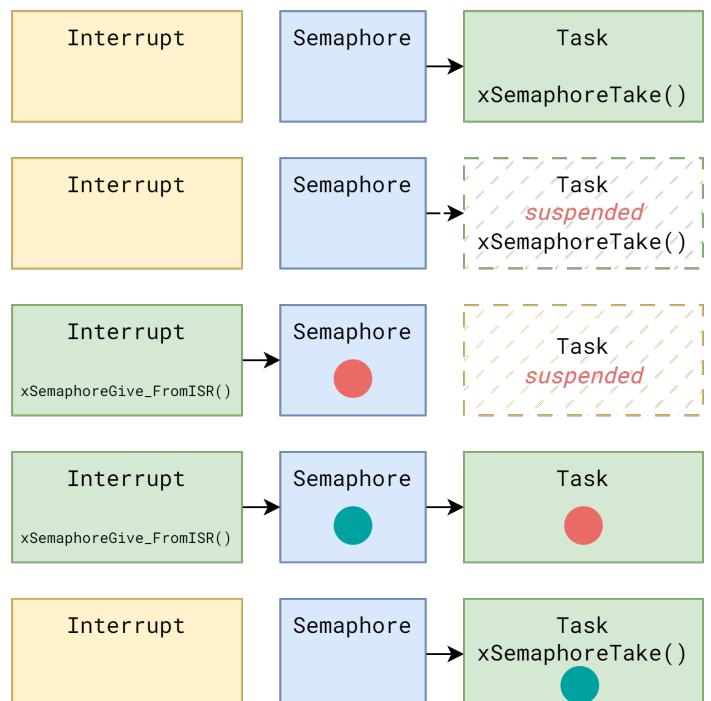
## Semaphore

Ein Semaphore kann freigegeben werden, muss aber nicht.

```
SemaphoreHandle_t xSemaphore;
xSemaphore = xSemaphoreCreateBinary();
if (xSemaphore == NULL) {
    /* Failed! Not enough heap memory? */
}
/* The semaphore can now be used */
```

- `xSemaphoreCreateBinary()`: erstellt eine Binäre Semaphore.
- `xSemaphoreCreateCounting()`: erstellt eine Counting Semaphore.
- `xSemaphoreTake()`: Einen Lock mit einer Binären oder Counting Semaphore anfordern.
- `xSemaphoreGive()`: Eine Binäre oder Counting Semaphore freigeben.
- `xSemaphoreDelete()`

Dieses Beispiel einer Binären Semaphore zeigt, wie über ein Token benachrichtigungen an einen Task gesendet werden können. Hierbei muss die Semaphore vom Task nicht zurückgegeben werden.



## Mutex

Ein Mutex **muss** nach dem Nehmen irgendwann zurückgegeben werden!

```
#define configUSE_MUTEXES (1)
#define configUSE_RECURSIVE_MUTEXES (1)
```

- xSemaphoreCreateMutex(): erstellt ein Mutex
- xSemaphoreCreateRecursiveMutex(): erstellt ein rekursiven Mutex
- xSemaphoreTakeRecursive(), xSemaphoreGiveRecursive: **nur für rekursive Mutex**

Ein Mutex ist wie ein Semaphore (einfach mit Priority Inheritance). Ein Mutex kann *normal* oder *rekursiv* sein.

### **i** Rekursive Mutex

Ein **rekursiven** Mutex kann verschachtelt werden, bzw. mehrmals vom **selben** Task aufgerufen werden. Um den Mutex zurückzugeben, muss dieser genau so viele Male zurückgegeben werden, wie er genommen wurde.

- In den Recursive-Funktionen wird eine Mutex-Count Variable de-/inkrementiert
- Besonders nützlich, falls die aufgerufene Funktion rekursiv ist

## Konzepte

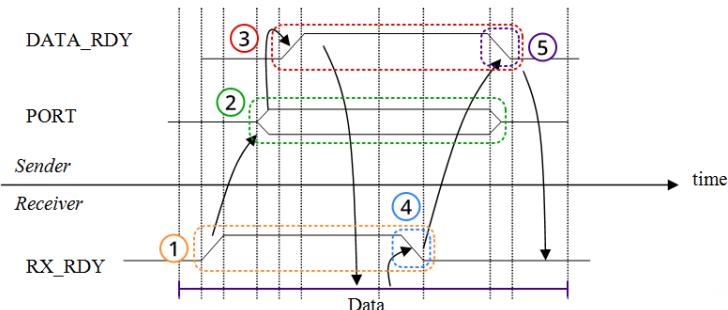
### Synchronisation

Die Systeme müssen auf die reale Welt abgestimmt und synchronisiert werden. Die Zeit der realen Welt ist kontinuierlich, während die Computerzeit diskret ist. Um die Computer mit der realen Welt zu verbinden, müssen sie mit der Zeit der realen Welt synchronisiert werden.

#### **🔥** Timing

Man darf Daten nicht schneller schicken, als diese von der Gegenseite angenommen werden können.

### Kommunikation



- ① Empfänger signalisiert Sender Ready
- ② Sender darf nur senden, wenn Empfänger Ready
- ③ Empfänger muss wissen, wann Daten bereit
- ④ Empfänger teilt Sender mit, dass Daten empfangen
- ⑤ Sender bestätigt dies → neuer Zyklus

### **i** Handshaking

Mit Handshaking können Kommunikationen *synchronisiert* werden.

```
void read(void) {
    PORTB.DDR1 = 1; /* pin B1 -> output */
    PORTB.B1 = 1; /* B1 initially high */
    for(size_t i=0; i<sizeof(buffer); i++) {
        /* initiate handshaking
           with setting B1 low */
    }
```

PORTB.B1 = 0;

while (! PORT.B0) {}

while(PORT.B0) {}

buffer[i] = PORTA;

PORTB.B1 = 1;

}

}

①

②

③

① Handshake Start B1 → Low

② Synchronisieren

③ Handshake Ende B2 → High

### **i** Handshake & Synchronisation

**Handshaking** ist ein definiertes Protokoll zwischen zwei Teilnehmern oder eine Realisierung einer Synchronisation, während **Synchronisation** ein Warten eines Prozesses oder Systems auf ein anderes ist, und ein Konzept beschreibt.

### Realtime

Realtime Sync wartet einfach eine fixe "echte" Zeit, ohne den Zustand des Gerätes zu prüfen.



```
void read(void) {
    for(size_t i=0; i<sizeof(buffer); i++) {
        for(int j=0; j <10000; j++) {
            /* wait some time */
            __asm("nop");
        }
        buffer[i] = PORTA;
    }
}
```

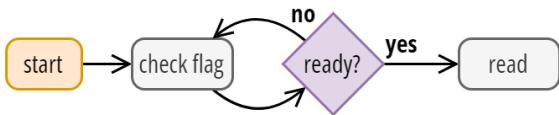
**+** sehr simpel

**-** nicht Robust → Latenz von verschiedenen Prozessen (DMA, Speicherzugriff, Interrupts) hat starken Einfluss, Compiler-Optimierung kann die Bedeutung ungültig machen

**-** Code macht während der Zeit gar nichts

## Gadfly / Polling

Gadfly Sync überprüft periodisch einen Zustand und fährt erst dann weiter, wenn die Bedingung erfüllt ist.



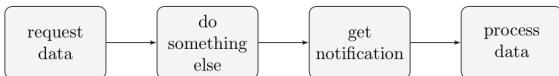
```

void read(void) {
    for (size_t i = 0; i < sizeof(buffer); i++) {
        while (!PORTB.B0) { /* reading 0: no hole */
            /* while there is no hole , wait for rising edge */
        }
        buffer[i] = PORTA; /* in the hole: read data */
        while (PORTB.B0) {
            /* reading 1: we have a hole */
            /* get out of hole , wait for falling edge */
        }
    }
}
  
```

- + reduzierte System Latenz
- + Daten gelesen, wenn verfügbar
- Benötigt unnötig Rechenzeit beim Warten

## Interrupt

Prozess wird aufgeteilt in Interrupt-Routine und Hauptprogramm.



```

volatile bool isrFlag = false;

void GPIO_ISR(void) { ①
    AcknowledgeInterrupt;
    isrFlag = true;
}

void read(void) {
    isrFlag = false;
    ConfigureGPIO(rising_edge);
    EnableInterrupt(gpio_isr);
    for(size_t i=0; i<sizeof(buffer); i++) { ②
        while (!isrFlag) {}
        buffer[i] = PORTA;
        isrFlag = false; ③
    }
}
  
```

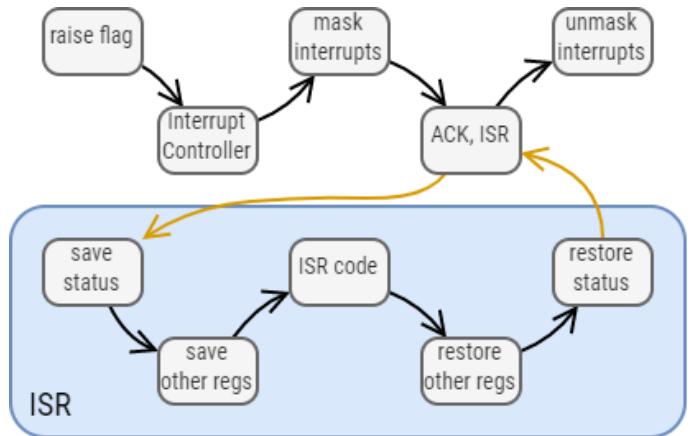
- ① Steigender Flanken Interrupt für Sprocket
- ② Synchronisation auf steigende Flanke
- ③ Vorbereitung für nächste Iteration

- + mehrere Prozess "gleichzeitig"

- "teurere" Implementation

! ISR **muss** kurz gehalten sein

! ISR-Prioritäten richtig setzen → Reduktion der *Interrupt Latenz*



**Signalisierung** – Unterbrechung wird in der Hardware verarbeitet  
Allfällige Instruktion im CPU wird unterbrochen, zurückgestellt oder abgeschlossen → Architektur & Pipeline abhängig

**Zustand sichern** – Programmzustand wird gesichert → PC, CPU Register, Prioritäten, etc. auf Stack (normalerweise)

**Verzweigung** – Hardware bestimmt wohin verzweigt werden soll → Passt PC, SP, R1...Rn, Prioritäten, etc. an

**Rettung benutzter Register** – Je nach Architektur: Teil der Register automatisch auf den Stack gelegt, anderer Teil muss manuell (falls ISR diese ändert). FPU wird oft separat verarbeitet

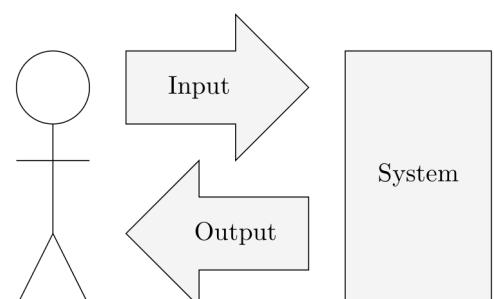
**ISR Programm** – Ausführung entsprechendes ISR-Programm  
ISR bestätigen → nicht nochmals Programm ausgeführt wird beim Verlassen

**Exit ISR** – Rückgang-Operation um zum alten Zustand zurückzukehren.

**Rückkehr zum unterbrochenen Programm** – Sprung zum vorher unterbrochenen Programm

## Benutzerschnittstellen

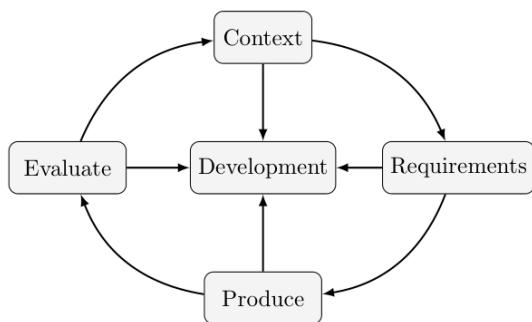
### Hardware



- **Akustik** – Kommunikation mit Audio
  - Lautsprecher, Buzzer für Ausgabe
  - Mikrofon für Eingabe

- **Optik** – Kommunikation mit Licht
  - LED, Display für Ausgabe
  - Kamera, Lichtsensor für Eingabe
- **Berührung** – Kommunikation via Elektromechanik
  - Taster, Schalter, Drehgeber, Tastaturen als Eingabe
  - Haptisches Feedback von den Eingaben
- **Composite** – Composite User Interface
  - Kombination von mehreren verschiedenen Elementen oder Sensoren
  - Virtual/Augmented Reality, Touchscreens, etc.

## Design Prozess



- **Kontext** – Was soll gemacht werden? Was ist das Ziel?  
Umgebung? Benutzer?
- **Anforderungen** – quantitative und qualitative Anforderungen
- **Produktion** – Basierend auf den Anforderungen werden Modelle oder Skizzen erstellt, wie Benutzerinteraktion und Schnittstellen aussehen sollen.
- **Evaluation** – Durch Benutzerbefragung Anforderungen evaluieren
  - Feedback wird wieder in den Kontext gebracht.

## Überlegungen

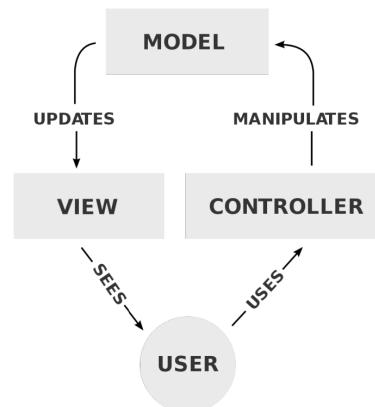
- **Kosten** – Wichtiger Faktor → Für welche Komponenten entscheidet man sich oder gibt es günstigere Alternativen?
- **Wahrnehmung** – Subjektive Wahrnehmung und Akzeptanz des Benutzers → Wie ist die *Usability*?
  - Verschiedene (redundante) Schnittstellen erlauben dem Benutzer eine Wahl (z.B. Sprachsteuerung + Bedienpanel)
- **Zuverlässigkeit** – Mechanik/Taster haben begrenzte Lebensdauer, optische Systeme sind anfällig für Staub, Wasser kann eindringen und Schaden anrichten.
  - Nach aussen gerichtete Schnittstellen sollten vor ESD geschützt werden
- **Andere** – Wie sollte die Langzeitverfügbarkeit der gewählten Komponenten? Welche Benutzergruppen können das Produkt bedienen, welche nicht (Blindheit, Farbenblinde, Gehörlose)?

## Grafik

### Geschichte

- Konzept GUI in 1970er Jahren von Xerox entwickelt
- Bekannt wurde es durch Apple mit Apple Lisa & Macintosh
- Microsoft probierte es & schaffte es schlussendlich
  - erster Versuch: Windows 1.0 → kein Erfolg
  - Windows 3.1 & Windows for Workgroups (Unterstützung für Netzwerke) brachte Erfolg
- Gleichzeitig innovative Ansätze wie NeXTSTEP

### Model-View-Controller



- **Model** – zentrales Element des Patterns
  - enthält Datenstrukturen und die Daten
  - verwaltet die Daten gemäss den Regeln der Anwendung
- **View**
  - Repräsentation und Anzeige der Daten
- **Controller**
  - Reagiert und verarbeitet Benutzereingaben

#### Vorteil

Module können einfach ausgetauscht werden → bessere Portierbarkeit

Raspberry, embedded Linux, Mobiltelefone,  
Router, Modem, Access Points.

ARM war erfolgreich, da sie grosse Leistungen  
instromsparende Controller packen konnten.



Konkurrenz: Intel

