

Advanced Embedded Systems

Zusammenfassung

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

Inhaltsverzeichnis

Entwicklung	4
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
Visualisierung	5
MCUXpresso	5
Task List View	5
Queue List View	5
FreeRTOS Trace Hooks	5
Segger System View	6
Viewer Hooks	6
Firmware	6
Architektur	6
Laufzeitmodelle	6
Module (Baublöcke)	8
Anforderung	8
Schnittstelle (.h/.hpp)	8
Device Konfigurieren & Erstellen	8
Bibliotheken	9
Archiv & Quelltexte	9
Startup Code	9
Runtime Library	9
Standard-Bibliotheken	9
Systeme	9
Transformierende Systeme	10
Reaktive Systeme	10
Interaktive Systeme	10
Kombiniertes System	10
Architektur	10
Systemaufbau	10
Mikrocontroller	10
Rechnerarchitektur	10
von Neumann Architektur	10
Harvard Architektur	10
System on Chip (SoC)	11
ARM Cortex Familie	11
Übersicht	11

Vektor Tabelle	11
Cortex M4 (TinyK22)	11
Cortex M0/M0+ (LPC845)	12
Cycle Counter	12
MCULib	12
DWT	12
FreeRTOS	12
Echtzeit	12
Harte & Weiche Echtzeit	13
Periodische Echtzeit	13
Architektur	13
Philosophie	13
Block Diagramm	13
Kernel	14
einfache API	14
Kontext Wechsel	15
Interrupts	15
Priorität FreeRTOS – Cortex-M	15
ARM Cortex-M Interrupts	15
BASEPRI (Cortex M4)	15
Task (<i>Threads</i>)	15
API	16
Preemptive Priority Scheduling	16
Time Slicing	16
Suicide Task	16
IDLE-Task	16
Timer	17
Queue	17
Semaphore & Mutex	17
Counting Semaphore	17
Prioritäten	17
Konzepte	17
Synchronisation	17
Kommunikation	17
Realtime	17
Gadfly / Polling	18
Interrupt	18
Benutzer	19

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



Entwicklung

Cross-Development

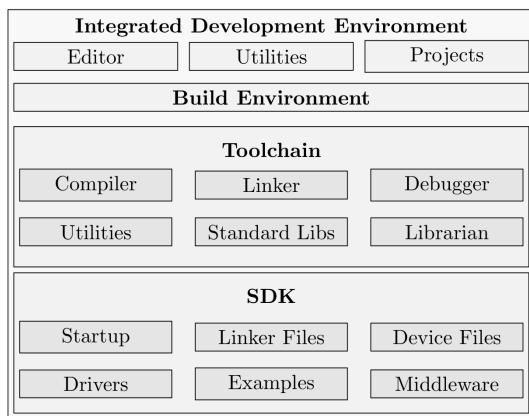
Cross-Development bedeutet die Entwicklung einer Firmware auf einem **Host** für einen **Target**. Grund dafür ist, dass das *embedded Target* nicht genügend Ressourcen (CPU Leistung, Speicher) für die direkte Entwicklung hat.

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, I²C, 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.

TCB#	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		0x61a2 (100.0%)
> 3	Tmr Svc	0x20003530	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...	# Rx...	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 [...]		

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

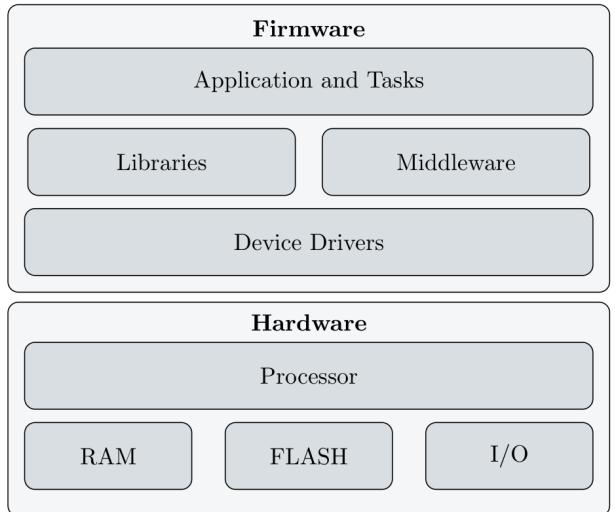
```
#define configUSE SEGGER SYSTEM VIEWER HOOKS (1)

#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

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.

Viewer Hooks

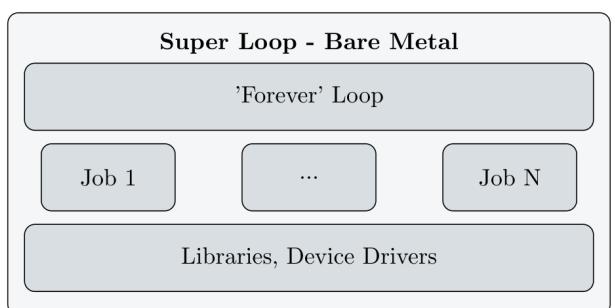
```
#if configUSE SEGGER SYSTEM VIEWER HOOKS
    SEGGER_SYSVIEW_OnUserStart(unsigned MarkerId);
    SEGGER_SYSVIEW_MarkStart(unsigned MarkerId);
#endif

#if configUSE SEGGER SYSTEM VIEWER HOOKS
    SEGGER_SYSVIEW_OnUserStop(unsigned MarkerId);
    SEGGER_SYSVIEW_MarkStop(unsigned MarkerId);
#endif
```

Laufzeitmodelle

Beschreibt wie eine Firmware ausgeführt wird.

Super Loop



Firmware

i Firmware versus Software

Firmware...

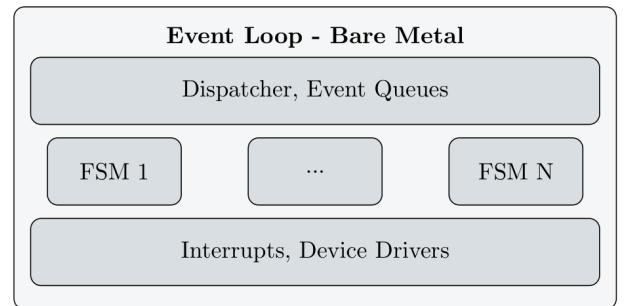
- wird direkt auf das Gerät einprogrammiert
- wenig veränderbar / ist firm (höhö)
- hat höhere Qualitätsanforderung → Aufwand ist hoch für Änderungen

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

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

- + Latenz zwischen Jobs kann reduziert werden

Loop mit Events



- + Sehr einfach und gut wartbar

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 Latenzen von Events & Interrupts klein gehalten werden.

- Jobs können länger dauern und somit andere Jobs **verzögern**
→ Latenz

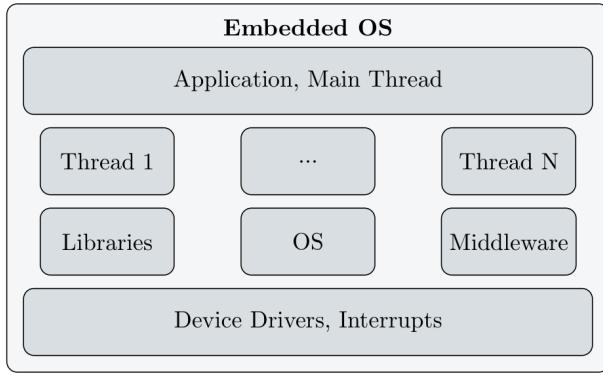
```
void ButtonInterrupt (void) {           ①
    QueueEvent( Button_Pressed );
}
}

void main(void) {
    InitHardware ();
    InitDriversAndInterrupts ();
    for (;;) {
        GoToSleep (); /* wait for event */
        ProcessQueues (); ②
    }
}
```

- 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 einfacher Erweiterung von neuen 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.
- Abstraktion mit HW & SW

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!

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.

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, ...`).

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

```

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

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

```

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

① Initialisierung Konfiguration → Defaultwerte zuweisen

② Erstellung *Device Handle* → Speicher allozieren & konfigurieren

- ③ Deinitialisierung des Device → Speicher freigeben & handle = NULL

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

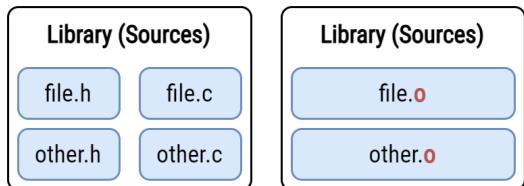
Bibliotheken

- <asset.h> – assert-Makro
- <math.h> – Mathe: 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



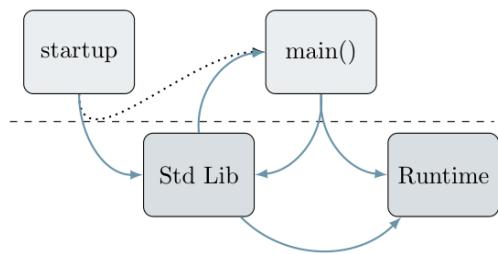
Quelltexte

- + **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

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

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.

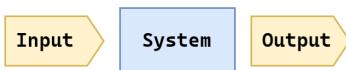
Anwendung: Echtzeitsystem, Wetterstation, Steuerung für Roboterarm, etc.

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 Speicherverbrauch

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

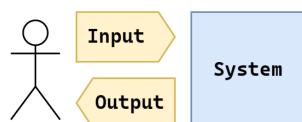


Interaktive Systeme

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



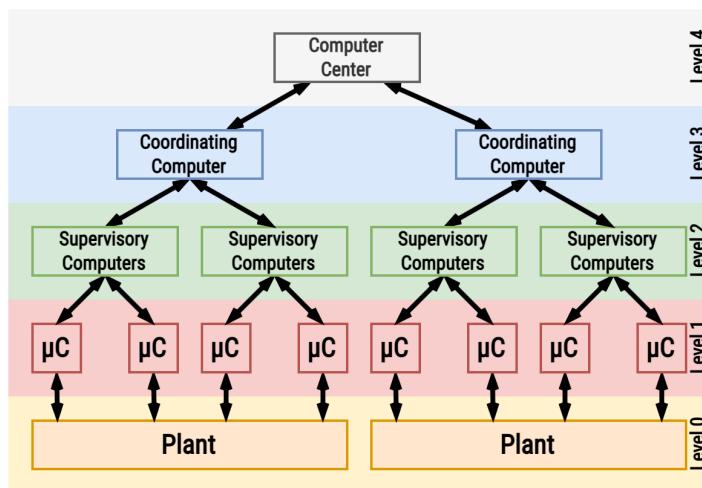
Kombiniertes System

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



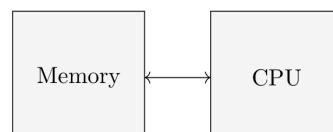
- ④ Production Scheduling (Planung & Verwaltung einer Produktion)
- ③ Production Control (Kontrolle von mehreren Prod.-Anlagen)
- ② Plant Supervisory (Kontrolle & Überprüfung über stärkeren Rechner)
- ① Direct Control (Mikrocontroller)
- ⑤ Field Level (Maschine, Pumpe, Roboter,...)

Mikrocontroller

Rechnerarchitektur

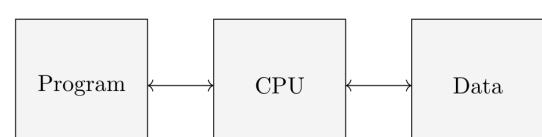
von Neumann Architektur

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.



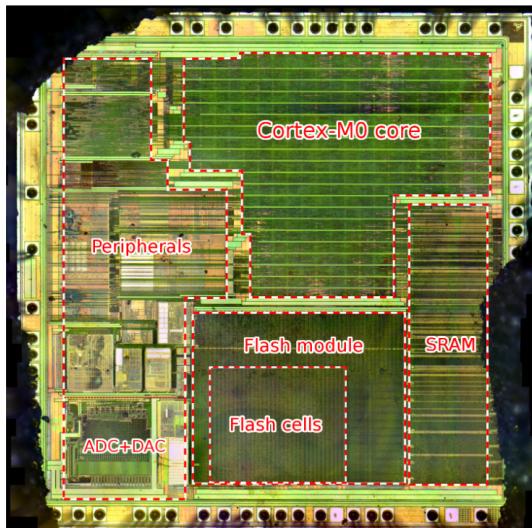
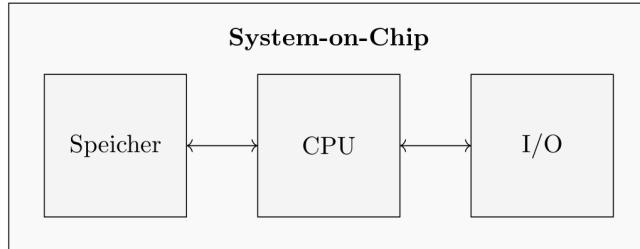
Harvard Architektur

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 RAM-Speicher blieben unter 100MHz stehen.



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.

ARM Cortex Familie

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

Vektor Tabelle

Cortex M4 (TinyK22)

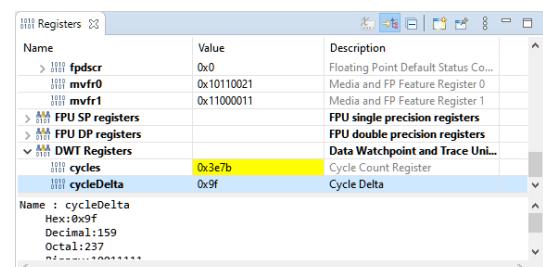
Exception number	IRQ number	Offset	Vector
16+n	n	0x0040+4n	IRQn
.	.	.	.
.	.	0x004C	IRQ2
18	2	0x0048	IRQ1
17	1	0x0044	IRQ0
16	0	0x0040	Systick
15	-1	0x003C	PendSV
14	-2	0x0038	Reserved
13			Reserved for Debug
12			SVCall
11	-5	0x002C	
10			Reserved
9			
8			
7			
6	-10	0x0018	Usage fault
5	-11	0x0014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

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	
12			
11	-5	SVCall	0x2C
10			
9			
8			
7		Reserved	
6			
5			
4			
3	-13	HardFault	0x10
2	-14	NMI	0x0C
1		Reset	0x08
		Initial SP value	0x04
			0x00

DWT

Data Watchpoint and Trace Unit



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 Fehlalarme).
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

Cycle Counter

MCULib

```
#include "cycles.h"

CCOUNTER_START();
/* do stuff */
CCOUNTER_STOP();
```

①

②

- ① Includierung
- ② Anwendung

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


```

Installed S... Properties Problems Console Terminal Image Info

SW02_MK22F51212_CycleCounter\Link Debug [GDB SEGGER Interface Debugging]

nop test: delta: 11, overhead: 0 cycles: 11; time: 0 us

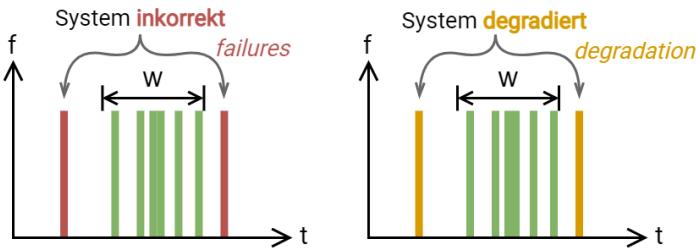
square root: delta: 3275, overhead: 0 cycles: 3275; time: 156 us

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)

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.

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.

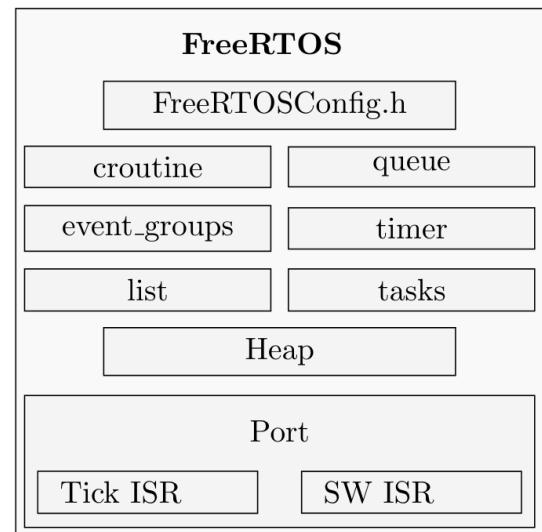
! Benötigte Interrupts

Damit das Betriebssystem korrekt läuft, werden zwei Interrupts benötigt:

Tick Interrupt – periodischer Interrupt, welcher einen Kontext Switch (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

```
void vTaskStartScheduler(void);  
void vTaskEndScheduler(void);
```

(1)

(2)

- ① Setzt den Scheulder 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.

vTaskSuspendAll()

einfache API

```
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 *suspended* → *active*

pdFALSE – Kernel *suspended*, da *vTaskSuspendAll* mehrmals aufgerufen wurde.

vTaskStartScheduler()

taskENTER_CRITICAL(), taskEXIT_CRITICAL()

```
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()

```
#define taskDISABLE_INTERRUPTS () \  
    portDISABLE_INTERRUPTS ()
```

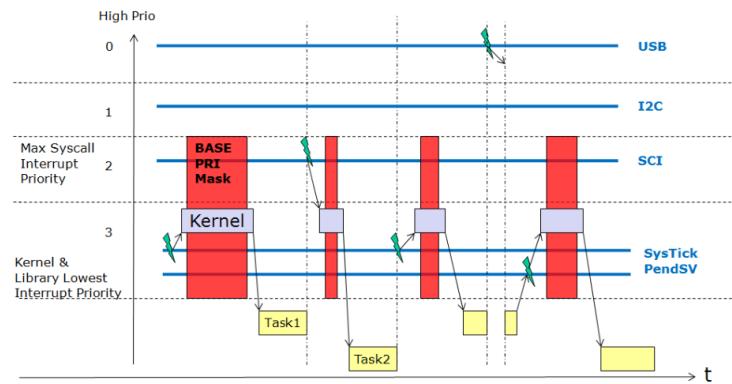
```
#define portDISABLE_INTERRUPTS () \
    portSET_INTERRUPT_MASK () \
#define portSET_INTERRUPT_MASK () \
    __asm volatile("cpsid i") /* M0+ */ \
#define portCLEAR_INTERRUPT_MASK () \
    __asm volatile("cpsie i") /* M0+ */
```

🔥 Interrupts Cortex M0+ & M4

Cortex M4 besitzt ein BASEPRI-Register, welches Interrupts ab dem gegeben Wert deaktiviert.

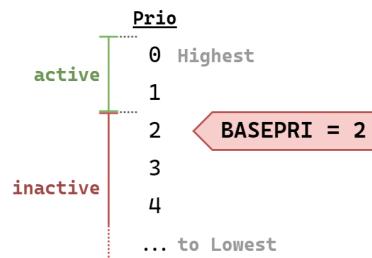
Cortex M0+ hat dies nicht und **deaktiviert daher alle Interrupts**.

ARM Cortex-M Interrupts



i 1

BASEPRI (Cortex M4)

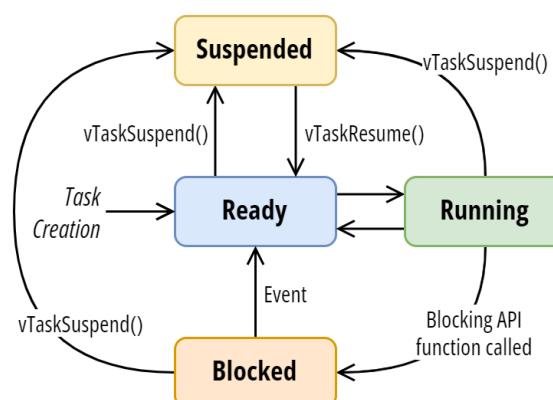


Task (Threads)

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

① Ignore value of params – Unterdrückt Compiler-Warnung

! Task-States



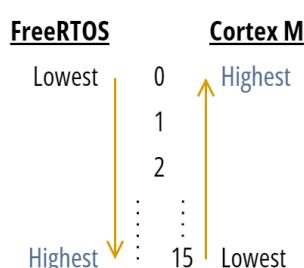
Kontext Wechsel

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

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

Interrupts

Priorität FreeRTOS – Cortex-M

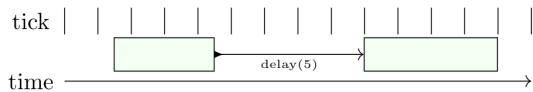


API**xTaskCreate**

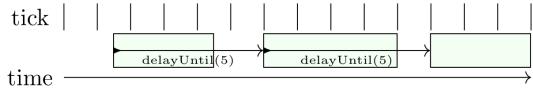
```
BaseType_t res;
TaskHandle_t taskHndl;

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

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

vTaskDelay()

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

vTaskDelayUntil()

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

vTaskSuspend()

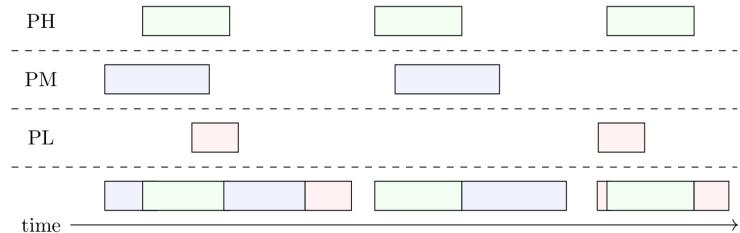
```
void vTaskSuspend ( TaskHandle_t xTaskToSuspend );
vTaskSuspend(NULL); \\ <1>
vTaskSuspend(blinkyTaskHandle); \\ <2>
```

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

vTaskResume()

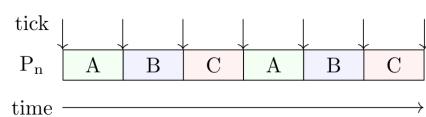
```
void vTaskResume(TaskHandle_t xTaskToResume);
```

Aktiviert den suspendierten Task.

Preemptive Priority Scheduling

```
#define configUSE_PREEMPTION (1) ①
#define configUSE_PREEMPTION (0) ②
```

- 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.

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* auf, welche von der Anwendung definierte Aufgaben übernehmen kann → z.B.

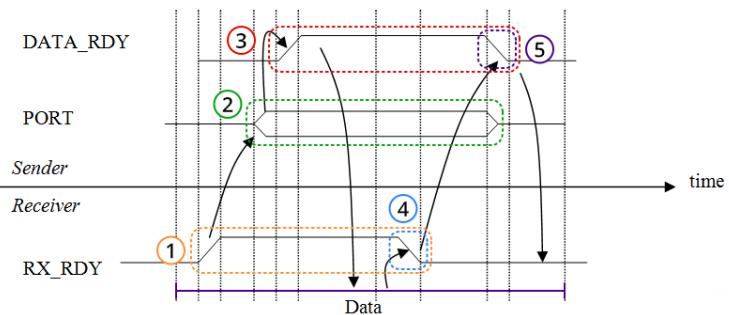
Low Power Modus

Idle Yielding

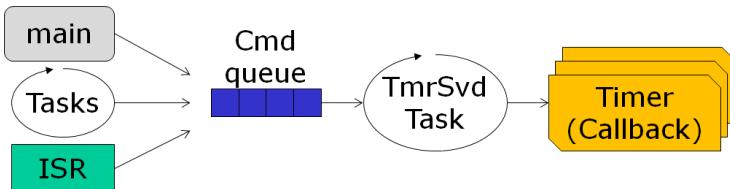
```
#define configIDLE_SHOULD_YIELD
```

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

Kommunikation



Timer



Timer Service Daemon

Mit configUSE_TIMERS wird die Timer-Funktionen aktiviert und aktiviert automatisch die *Timer Service Daemon*

Queue

Semaphore & Mutex

! Unterschied Semaphore & Mutex

Beide sind sehr ähnlich, ausser folgendes: Mutex hat Priority Inheritance,

Counting Semaphore

Prioritäten

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.

- 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

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) {}          ①
        while(PORT.B0) {}            ②
        buffer[i] = PORTA;
        PORTB.B1 = 1;                ③
    }
}
  
```

- ① Handshake Start B1 → Low
- ② Synchronisieren
- ③ Handshake Ende B2 → High

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.



```

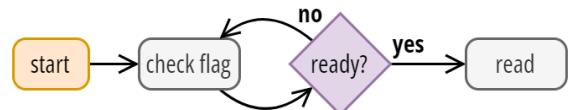
void read(void) {
    for(size_t i=0; i<sizeof(buffer); i++) {
        for(int j=0; j < 10000; j++) {
            /* wait some time */
            __asm("nop");
        }
        buffer[i] = PORTA;
    }
}
  
```

+ sehr simpel

- nicht Robust → Latenz von verschiedenen Prozessen (DMA, Speicherzugriff, Interrupts) hat starken Einfluss, Compiler-Optimierung kann die Bedingung 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 */
        }
    }
}
  
```

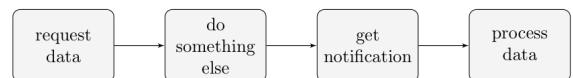
+ reduzierte System Latenz

+ 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) {
    AcknowledgeInterrupt;
  
```

(1)

```

isrFlag = true;
}

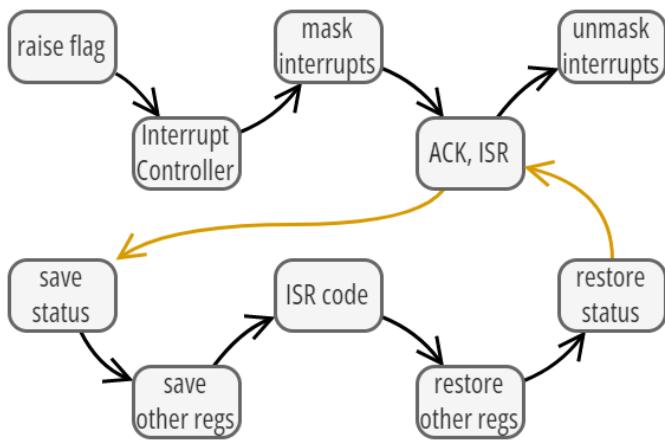
void read(void) {
    isrFlag = false;
    ConfigureGPIO (rising_edge);
    EnableInterrupt (gpio_isr);
    for(size_t i=0; i<sizeof(buffer); i++) {
        while (!isrFlag) {}                                ②
        buffer[i] = PORTA;
        isrFlag = false;                                 ③
    }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

```

Benutzer

- 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

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

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



Konkurrenz: Intel

Cortex-M4

Cortex-M3

Cortex-M0/M0+

