


Advanced Embedded Systems

Zusammenfassung

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

Inhaltsverzeichnis

Entwicklung	3
Cross-Development	3
Integrated Development Environment	3
Eclipse (Open Source IDE)	3
Plugin System	3
Workspace	3
Begriffe	3
Firmware	4
Architektur	4
Laufzeitmodelle	4
Module (Baublöcke)	5
Anforderung	5
Schnittstelle (.h/.hpp)	5
Device Konfigurieren & Erstellen	6
Bibliotheken	6
Archiv & Quelltexte	6
Startup Code	6
Runtime Library	7
Standard-Bibliotheken	7
Systeme	7
Transformierende Systeme	7
Reaktive Systeme	7
Interaktive Systeme	7
Kombiniertes System	7
Mikrocontroller	8
ARM Cortex Familie	8
FreeRTOS	8
Echtzeit	8
Harte & Weiche Echtzeit	8
Periodische Echtzeit	8
Architektur	8
Philosophie	8
Block Diagramm	9
Kernel	9
einfache API	9
Kontext Wechsel	10
Interrupts	10
Priorität FreeRTOS – Cortex-M	10
ARM Cortex-M Interrupts	10
BASEPRI (Cortex M4)	10
Task (<i>Threads</i>)	10

API	11
Preemptive Priority Scheduling	11
Time Slicing	11
Suicide Task	12
IDLE-Task	12
Timer	12
Queue	12
Semaphore & Mutex	12
Counting Semaphore	13
Prioritäten	13
C - Konzepte	13
Synchronisation	13
Realtime	13
Gadfly / Polling	13
Interrupt	13
Benutzer	13

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.

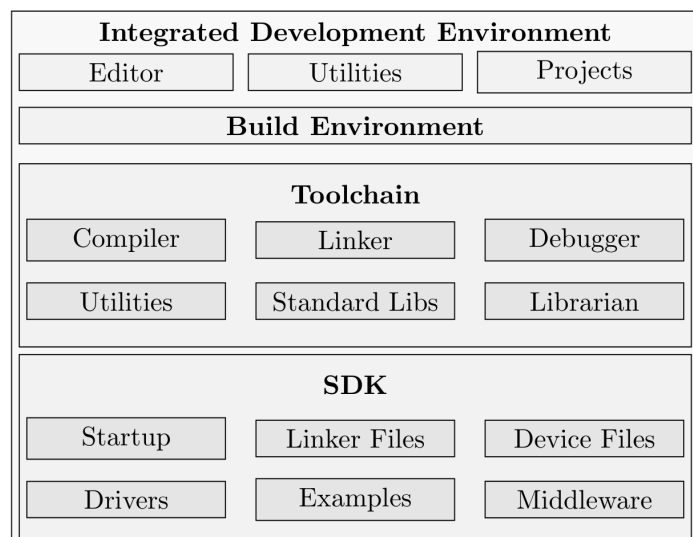
i Target & Host

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 Übersetzungsvorgang → *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

– Eierlegende Wollmilchsau → kann zu viel als nötig ist (abhängig von Workflow und Funktionsumfang) ;

i 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 Compiler 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, ...)

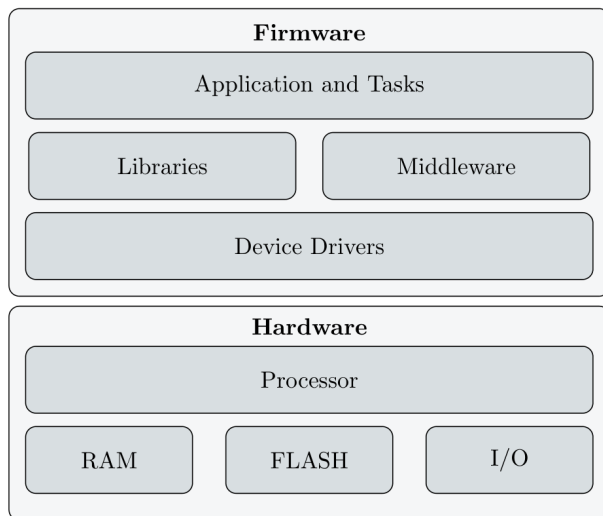
Firmware

i Firmware versus Software

Firmware. . .

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

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 Programmteile oder Programme (z.B. Datenbanken, Betriebssysteme oder Kommunikations-Stacks etwa für USB oder TCP/IP).

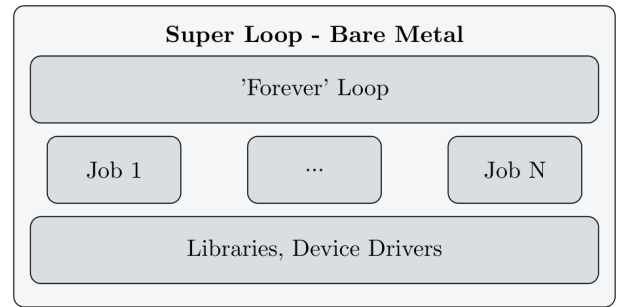
! Hardware-Abstraktion

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

Laufzeitmodelle

Beschreibt wie eine Firmware ausgeführt wird.

Super Loop



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

+ Sehr einfach und gut wartbar

– Jobs können länger dauern und somit andere Jobs **verzögern**

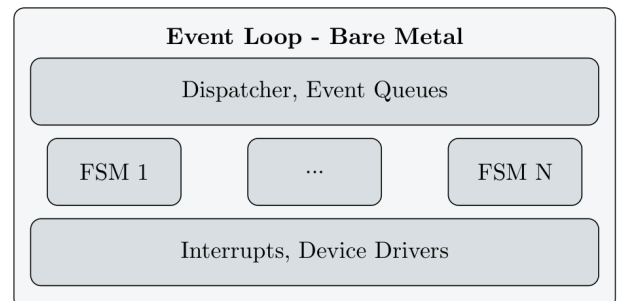
→ Latenz

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 Latenzen von Events & Interrupts klein gehalten werden.

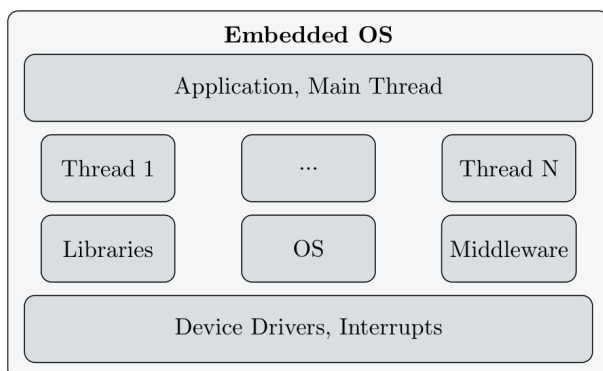
```
void ButtonInterrupt (void) {           ①
    QueueEvent( Button_Pressed );      ②
}

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 einfacher Erweiterung von neuen Funktionen.

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

void main(void) {
```

```
InitHardware ();
InitDriversAndInterrupts ();
CreateTask(mainTask);
StartOS ();
}
```

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 Gdflly

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 Nötigstes 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 :(

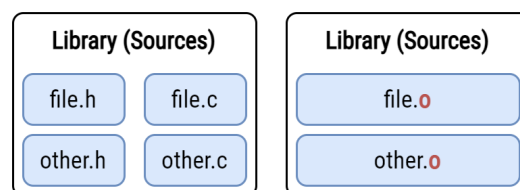
Bibliotheken

Include	Inhalt
<assert.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



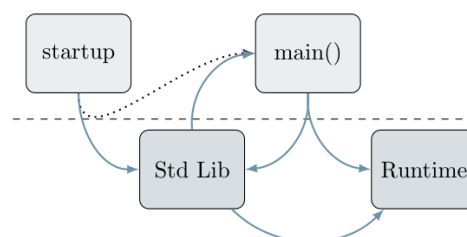
Quelltexte

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

Archiv

+ Keine Kompilierung nötig ; weniger falsch 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 → Anwendungsspezifisch
- **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.

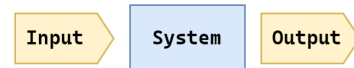
Transformierende Systeme

Verarbeitet ein Eingangssignal (*Input*) und gibt ein Ausgangssignal (*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

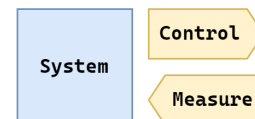


Reaktive Systeme

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

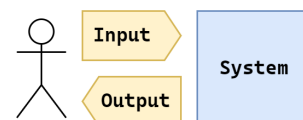


Interaktive Systeme

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.

Mikrocontroller

ARM Cortex Familie

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.

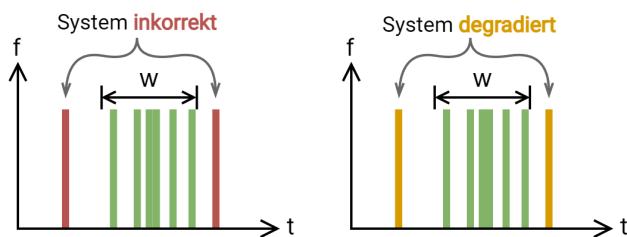
Echtzeit

! 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 $05:30 \pm 1$ Minute)
- **Relative** Rechtzeitigkeit – Relative Zeit nach Ereignis (z.B. 5 Minuten ($\pm 10s$) 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.

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).

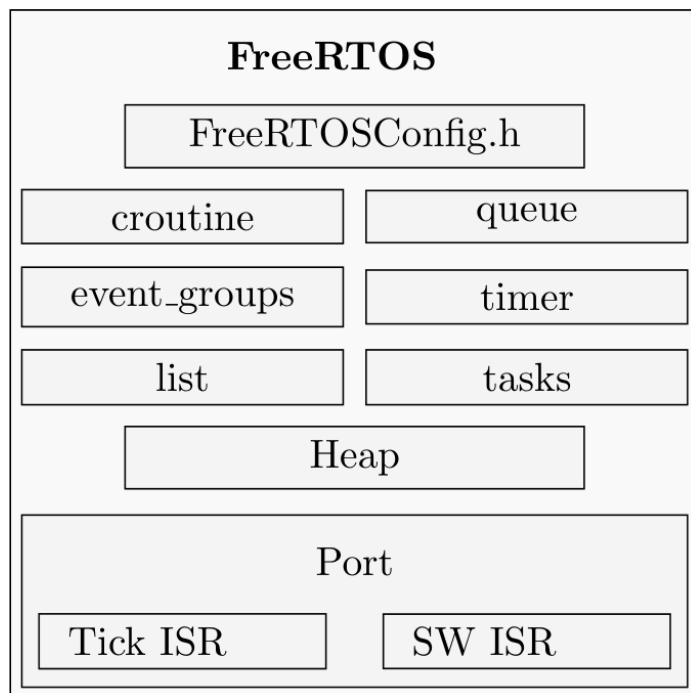
Kooperative – Ein Kontext Switch findet nur statt, wenn ein Task blockiert oder explizit ein Yield aufruft. Ein 'Yield' ist die Aufforderung an den Kernel, einen Kontext Wechsel vorzunehmen.

! 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**einfache 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.

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 *suspended* → *active*

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

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

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 *geyielded* 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.

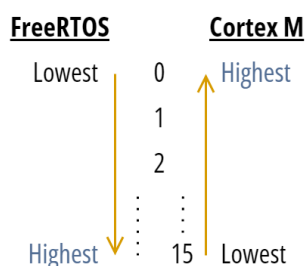
Kontext Wechsel

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

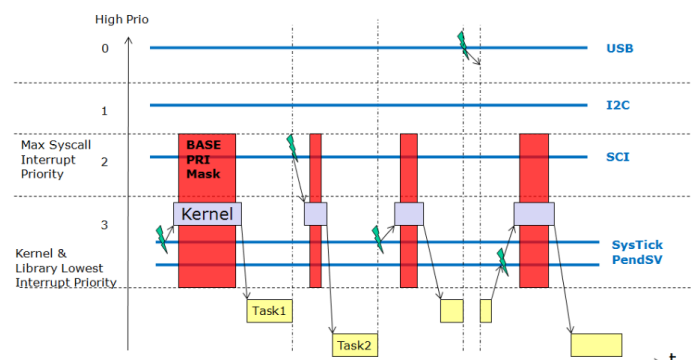
1. via Tick-Interrupt → Time Slicing oder Preemption ; synchrones Scheduling
2. via Anwendung (*SysCall*) → indirektes/asynchrones Scheduling, z.B. durch Senden einer Meldung ein dringlicher Task aktiviert
3. via Anwendung direkt → Yield

Interrupts

Priorität FreeRTOS – Cortex-M

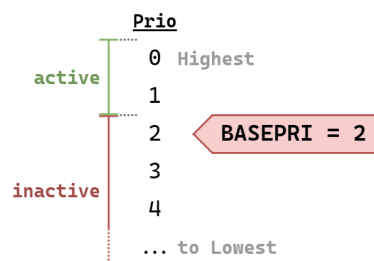


ARM Cortex-M Interrupts



i 1

BASEPRI (Cortex M4)

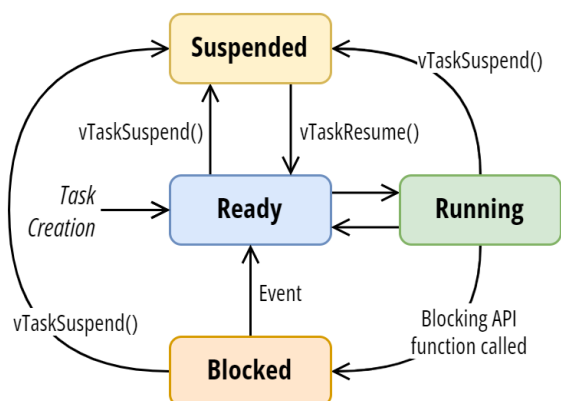


Task (Threads)

```
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



API

xTaskCreate

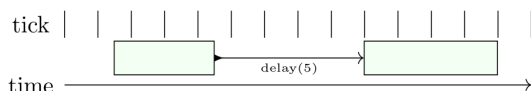
```

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()

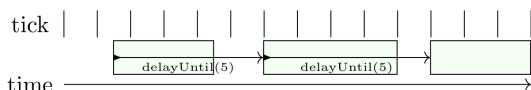


```

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()

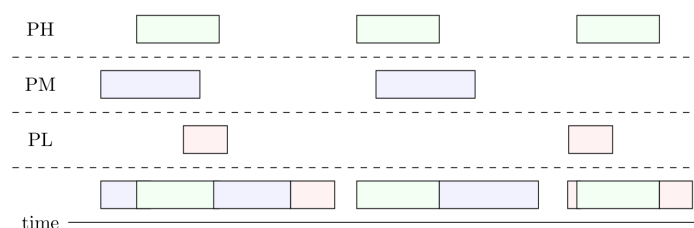
```

void vTaskResume(TaskHandle_t xTaskToResume);

```

Aktiviert den suspendierten Task.

Preemptive Priority Scheduling



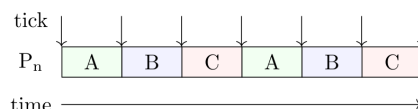
```

#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

Queue

```
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 Stack-speicher von `configMINIMAL_STACK_SIZE`.

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

! Idle Hook

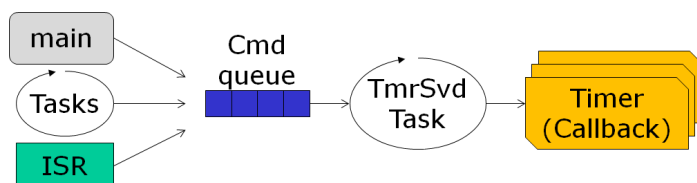
Der IDLE Task führt den *Idle Hook* auf, welche von der Anwendung definierte Aufgaben übernehmen kann → z.B. *Low Power Modus*

i Idle Yielding

```
#define configIDLE_SHOULD_YIELD
```

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

Timer



🔥 Timer Service Daemon

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

! Unterschied Semaphore & Mutex

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

Counting Semaphore

Prioritäten

C - Konzepte

Synchronisation

Realtime

Gadfly / Polling

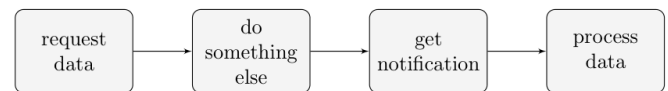
Gadfly Sync überprüft periodisch einen Zustand und fährt erst dann weiter, wenn die Bedingung erfüllt ist.

```
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 */
        }
    }
}
```

+ Benötigt keine unnötige Rechenzeit

- Benötigt keine unnötige Rechenzeit

Interrupt



Signalisierung → Zustand sichern → Verzweigung → Rettung benutzter Register → ISR Programm → Exit ISR → Rückkehr zum unterbrochenen Programm

Benutzer

