



RTO Realtime Operating Systems

Teil 1: Einführung, Echtzeit, Scheduling

Erstellt von Josef Langer

Einführung

- Embedded Operating System
 - ▶ Ist ein Betriebssystem für eingebettete Systeme. Es soll zuverlässig und effizient bzgl. der Ressourcen wie Stromverbrauch, Speicher oder Prozessorleistung sein
- Realtime Operating System (RTOS)
 - ▶ Ist typischerweise ein Embedded Operating System, das Echtzeitanforderungen genügen muss.
- Am Markt erhältliche RTOS
 - ▶ ca. 200 in Wikipedia gelistet
 - ▶ Siehe: https://en.wikipedia.org/wiki/Comparison_of_real-time_operating_systems

Definitionen

- Definition Echtzeit laut DIN ISO/IES 2382
 - ▶ *„Unter Echtzeit versteht man den Betrieb eines Rechensystems, bei dem Programme zur Verarbeitung anfallender Daten ständig betriebsbereit sind, derart, dass die Verarbeitungsergebnisse innerhalb einer vorgegebenen Zeitspanne verfügbar sind. Die Daten können je nach Anwendungsfall nach einer zeitlich zufälligen Verteilung oder zu vorherbestimmten Zeitpunkten anfallen.“*
- Das bedeutet:
 - ▶ Bestimmte Ereignisse müssen **innerhalb einer vorbestimmten Zeitspanne** abgearbeitet werden
 - ▶ Echtzeit bedeutet **NICHT**: möglichst rasch, sehr kurze Zeiten
 - ▶ **Sondern: rechtzeitig** (das könnten auch Sekunden sein)

Was macht ein Echtzeitsystem aus?

- Anforderungen:
 - ▶ Hardware und Software müssen sicherstellen, dass die zeitlichen Bedingungen eingehalten werden. Es geht nicht darum, dass die Abarbeitung sehr schnell ist, sondern **garantiert ausreichend schnell, um keine Deadline zu verletzen**.
- Rechtzeitigkeit
 - ▶ Der Ablauf von Datenaufnahme, Datenverarbeitung und Datenausgabe muss **rechtzeitig** erfolgen. Eingangsdaten verlieren unterschiedlich rasch an Gültigkeit (Heizung, Bremse,...)
- Determinismus
 - ▶ „Ein Echtzeitsystem arbeitet zeitlich determiniert, wenn **zu jeder Kombination aus Eingangsgrößen die Reaktionszeit** des Echtzeitsystems in festen zeitlichen Größen **vorhersagbar ist**.“
(M. Witzak, Echtzeit-Betriebssysteme)

Zusätzliche Praxis-Anforderungen

- In der Praxis ist relevant:
 - ▶ Auswertung von mehreren Ereignisgrößen parallel
 - ▶ die System arbeiten gleichzeitig oder quasi-gleichzeitig
 - ▶ Parallelisierung von Berechnungsaufgaben
 - ▶ Unterteilung in wichtige und unwichtige Aufgaben

Unterteilung von Echtzeitsystemen

- **Harte Echtzeit:**
 - ▶ Systeme wie Flugzeuge, Kraftwerke,... , die eine harte zeitliche Schranke vorgeben. Verletzungen von Zeitvorgaben können zu schwerwiegenden Problemen und Schäden führen
- **Weiche Echtzeit**
 - ▶ Systeme, die prinzipiell als Echtzeitsysteme zu betrachten sind aber dehnbare Zeitlimits haben. Z.B.: Bankomaten,... Das System muss aber 100% deterministisch arbeiten.

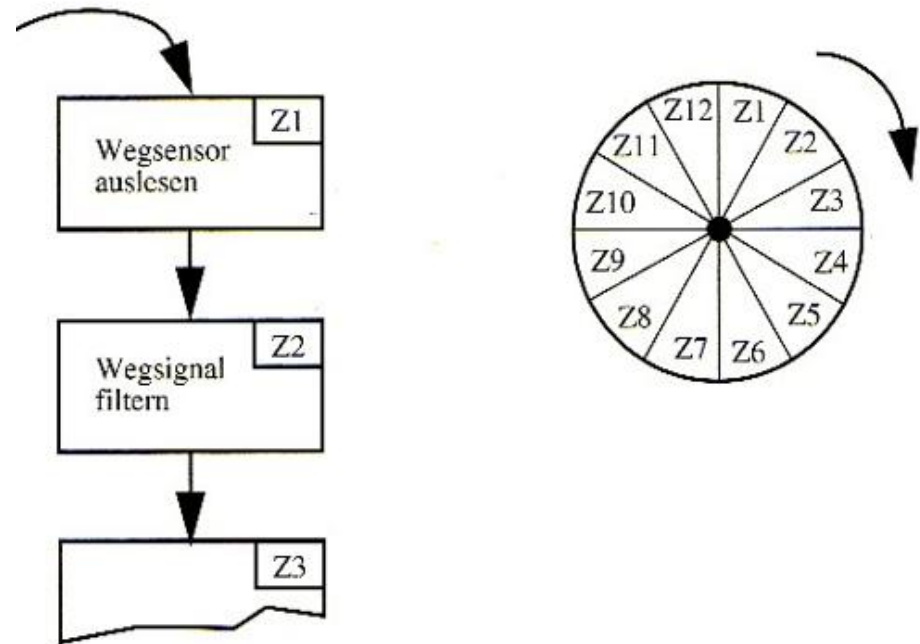
Anmerkung: die Einteilung ist in der Literatur nicht eindeutig, auch hard / firm / soft real time ist gebräuchlich.

Echtzeitprogrammierung

- Timer/Clock für Zeitüberwachung
- Parallelität durch Hardwarebausteine und spezielle Funktions-Register des Microcontrollers
- Übergeordnetes Programm, das die verschiedenen Berechnungsaufgaben gem. der zeitlichen Abfolge und Priorität verwaltet
- Applikationsprogramm (das sich nur bedingt um Echtzeit kümmern muss)

Synchrone Programmierung (Zeitscheiben)

- **Vorteil:**
 - ▶ Arbeitet 100% deterministisch.
 - ▶ Beweis der Funktion des Echtzeitsystems möglich.
- **Nachteil:**
 - ▶ Keine Flexibilität
 - ▶ Problem bei Zeitscheibenüberläufen



Asynchrone Programmierung (Ereignis gesteuert)

- Vorteile:
 - ▶ Flexible Zuweisung von Aufgaben und gute Ausnützung der Prozessor-Leistung (Power-Management).
- Nachteile:
 - ▶ Je mehr Module existieren, desto schwieriger ist Rechtzeitigkeit vorherzusagen
 - ▶ Determiniertheit ist schwer nachzuweisen.
- Abarbeitung der Module nicht rein zeitlich gesteuert sondern durch ein **Echtzeitbetriebssystem** (Real Time Operating System = RTOS).

Asynchrone Programmierung (Ereignis gesteuert)

- Eigenschaften des RTOS
 - ▶ Module besitzen verschiedene Zustände
 - Running, blocked, waiting
 - ▶ Prioritätssteuerung
 - ▶ Scheduler, der die Zuteilung der Rechenzeit regelt
 - ▶ Unterbrechung durch Interrupts
 - ▶ preemptives / kooperatives System
- Wünschenswert:
 - ▶ Dynamische Erkennung, wie viel Zeit einzelne Module zur Verfügung haben

Superloop

- Vorläufer eines Betriebssystems
- Verschiedene Aufgaben durch ISR abgebildet
- Sehr einfaches System
- Geringste Overhead

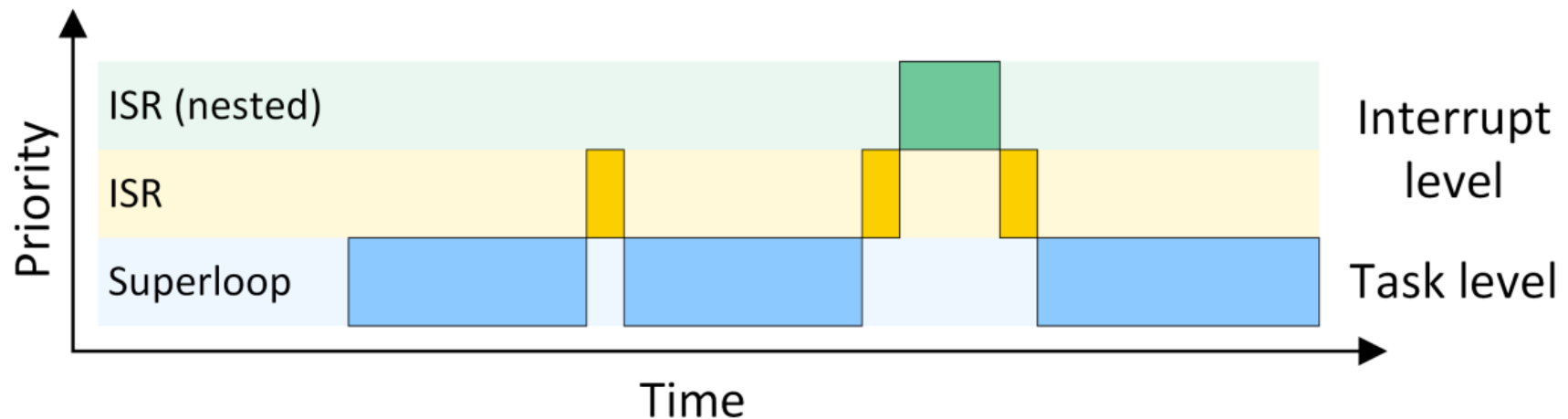
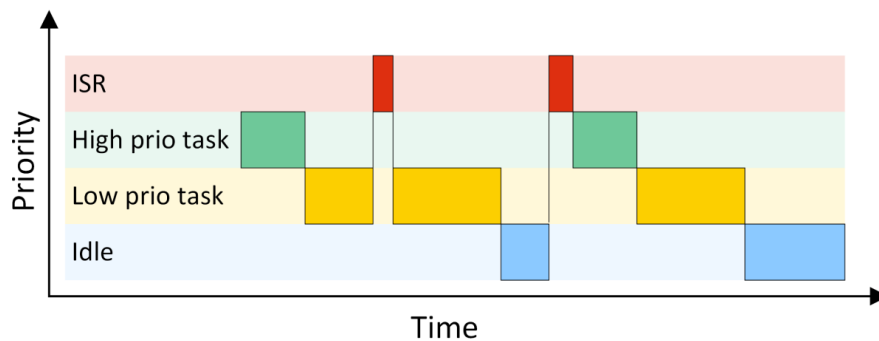


Abbildung: Seeger, embOS & smbOS-MPU, Real-Time Operating System User Guide & Reference Manual, 2018.

Kooperatives Multitasking

- Der Task-Switch muss durch laufenden Task zugelassen werden:
 - ▶ Falls der Task die CPU nicht freigibt, erfolgt kein Task-Switch.
 - ▶ Alle Anwendungen bleiben bei Fehler hängen – z.B. Win 3.11

Multitasking allgemein



Kooperatives Multitasking

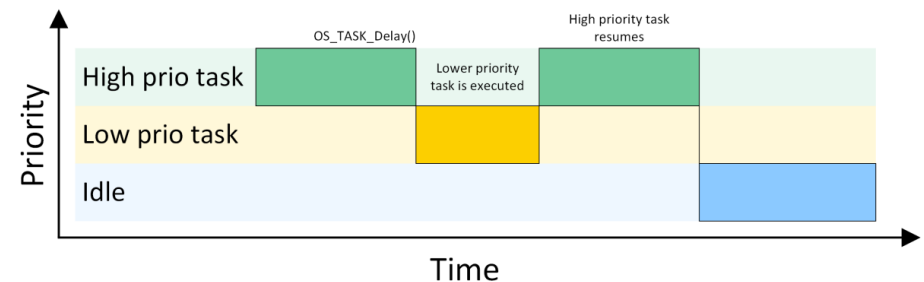


Abbildung: Seeger, embOS & smbOS-MPU, Real-Time Operating System User Guide & Reference Manual, 2018.

Preemptives Multitasking

- Unterbrechung der Tasks durch Zeitablauf (Round Robin Scheduling) oder durch Tasks mit höherer Priorität.
 - ▶ Keine Kooperation der Task nötig
 - ▶ Nachteil: Determinismus schwieriger nachweisbar

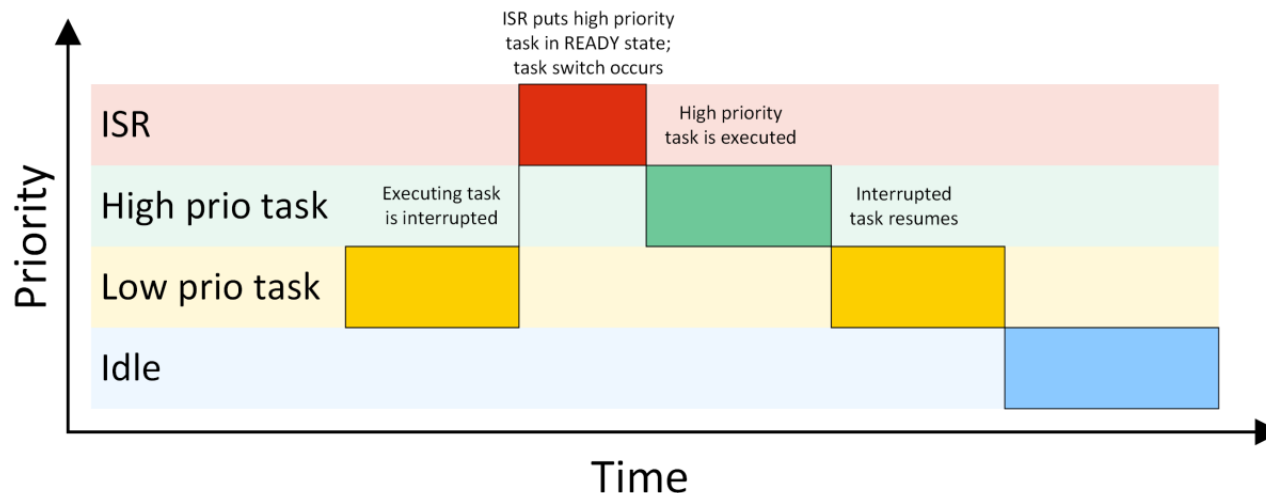
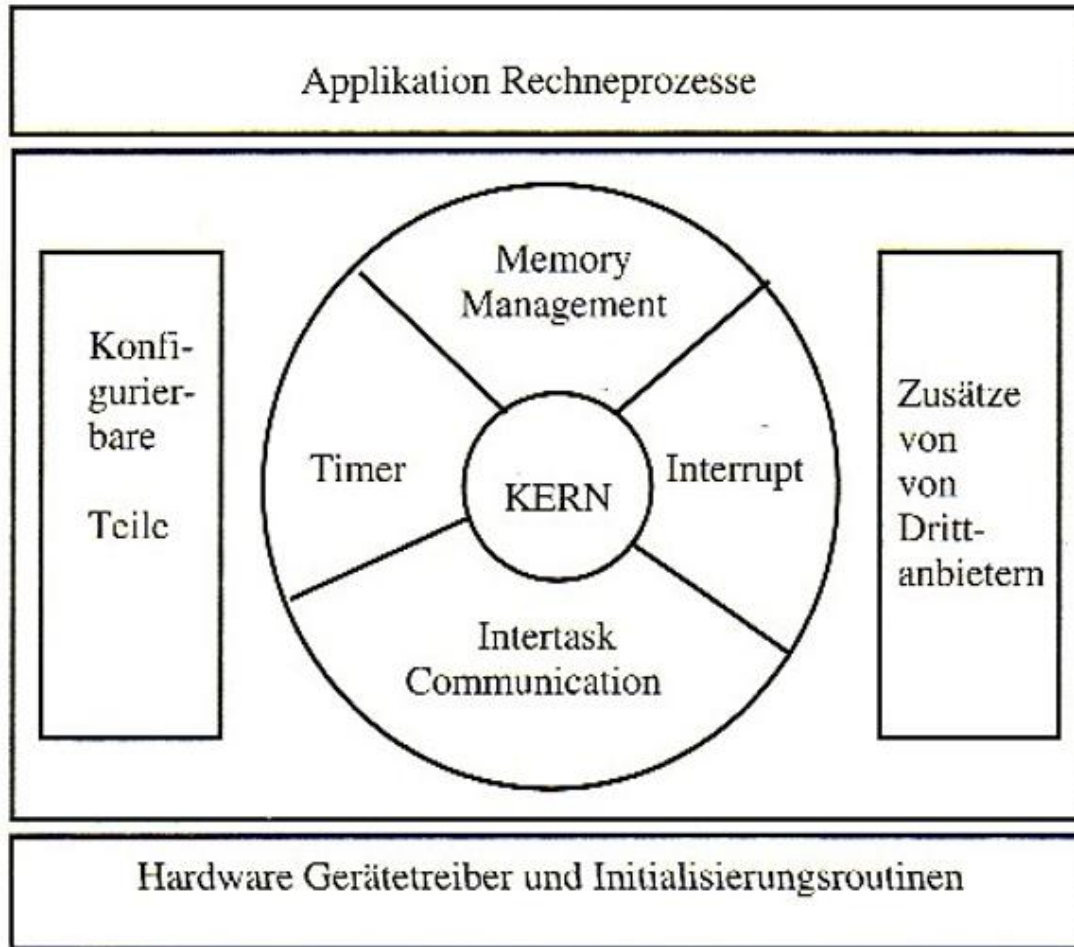


Abbildung: Seeger, embOS & smbOS-MPU, Real-Time Operating System User Guide & Reference Manual, 2018.

Anforderungen an ein RTOS

- Garantie der geforderten Rechtzeitigkeit
- Sicherstellung der Unterbrechungsfähigkeit (Interrupts)
- Sichere Verwaltung des Systemspeichers und der Rechenprozesse
- Definierte Reaktion auf Fehlerzustände (Fehlerverwaltung und –behandlung)
- Bereitstellung von Ressourcen zur Steuerung zeitlicher Abläufe
- I/O-Management und Hardwaresteuerung (Gerätetreiber)

Struktur eines RTOS

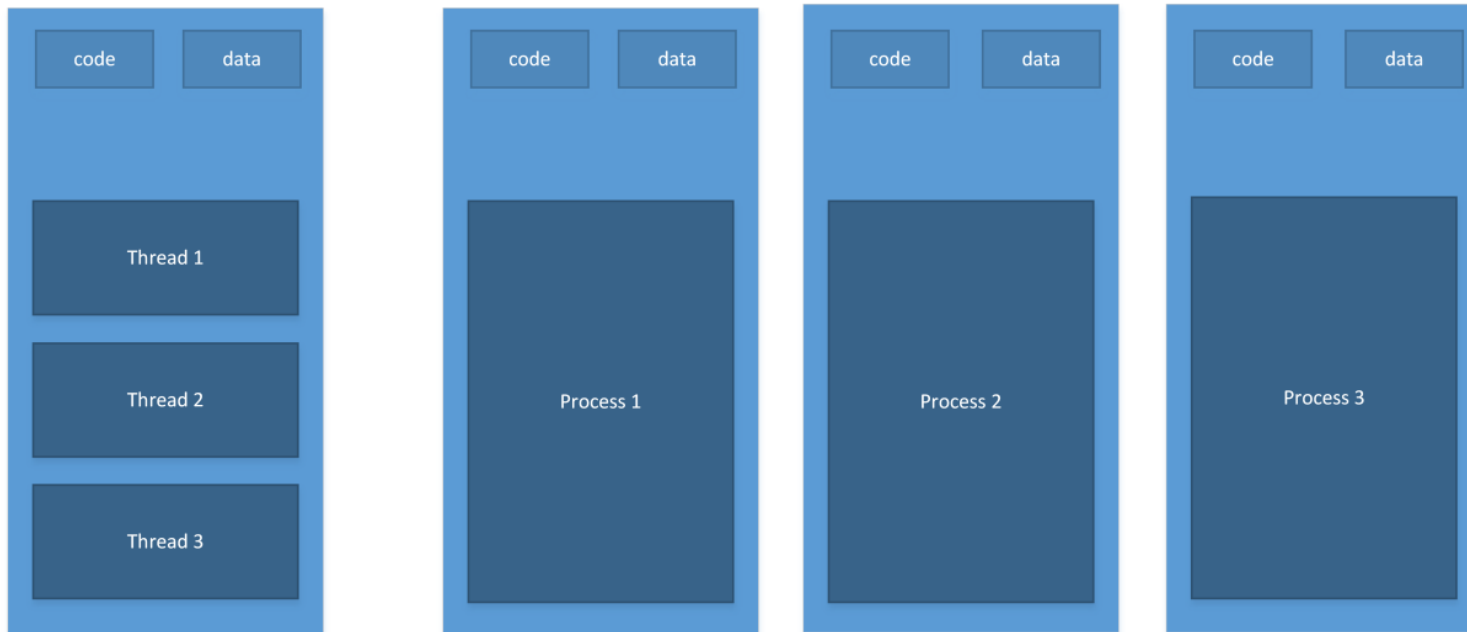


Was ist ein Task?

- Ein Programm, das auf einem CPU Kern läuft,
- mit eigenem Stack,
- mit eigenen Variablen
(die im TCB Task-Control-Block gespeichert sind),
- hat einen Taskzustand
 - ▶ wie running, waiting, blocked
- und eine Priorität.

Thread vs. Process

- Thread / Process / Task
- Viele Embedded Systeme ohne MMU
-> Task = Thread, Process nicht unterstützt



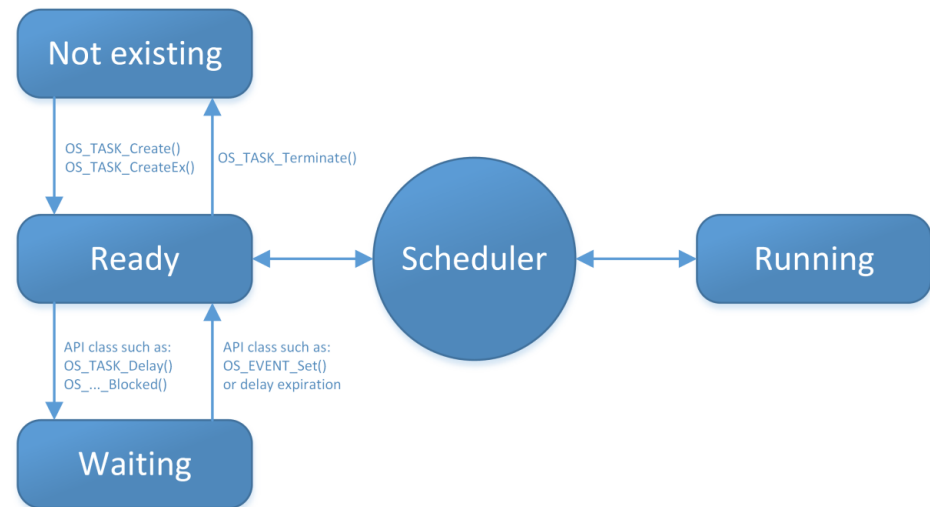
Task Zustände

- Embos

- ▶ Ready – Task ist erzeugt und bereit
- ▶ Waiting – verzögert für eine Zeitspanne oder wartet auf Ereignis
- ▶ Running – wird von der CPU abgearbeitet

- FreeRTOS

- ▶ Running
- ▶ Ready
- ▶ Blocked
- ▶ Suspended
- ▶ Deleted



Task Switch AVR und FreeRTOS

Beispiel:

- Task A ist im Zustand Running
- Task B ist im Zustand Ready
- Nach Ablauf des Betriebssystem-Timers wird Task B aktiviert

Register AVR allg.:

General
Purpose
Working
Registers

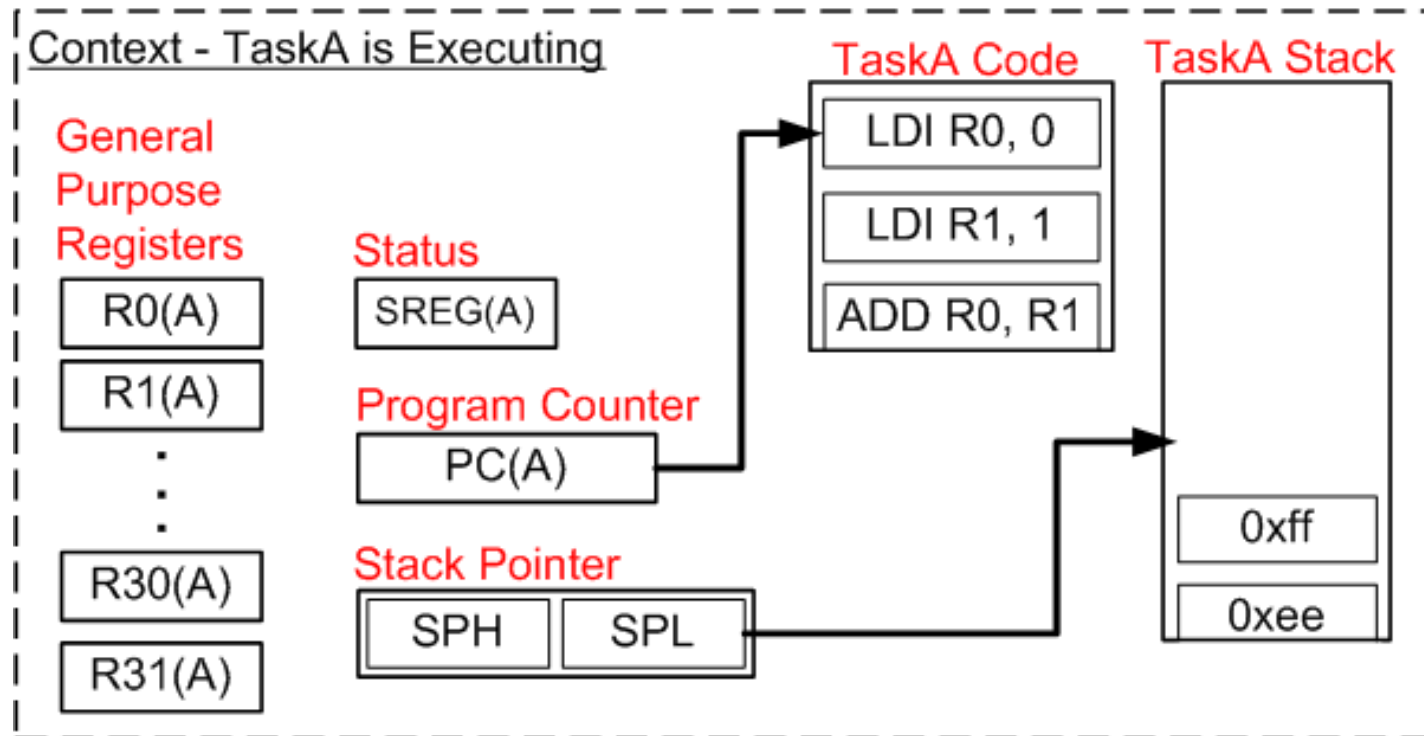
7	0	Addr.	
R0		0x00	
R1		0x01	
R2		0x02	
...			
R13		0x0D	
R14		0x0E	
R15		0x0F	
R16		0x10	
R17		0x11	
...			
R26		0x1A	X-register Low Byte
R27		0x1B	X-register High Byte
R28		0x1C	Y-register Low Byte
R29		0x1D	Y-register High Byte
R30		0x1E	Z-register Low Byte
R31		0x1F	Z-register High Byte

SREG:

Bit	7	6	5	4	3	2	1	0	
	I	T	H	S	V	N	Z	C	SREG
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

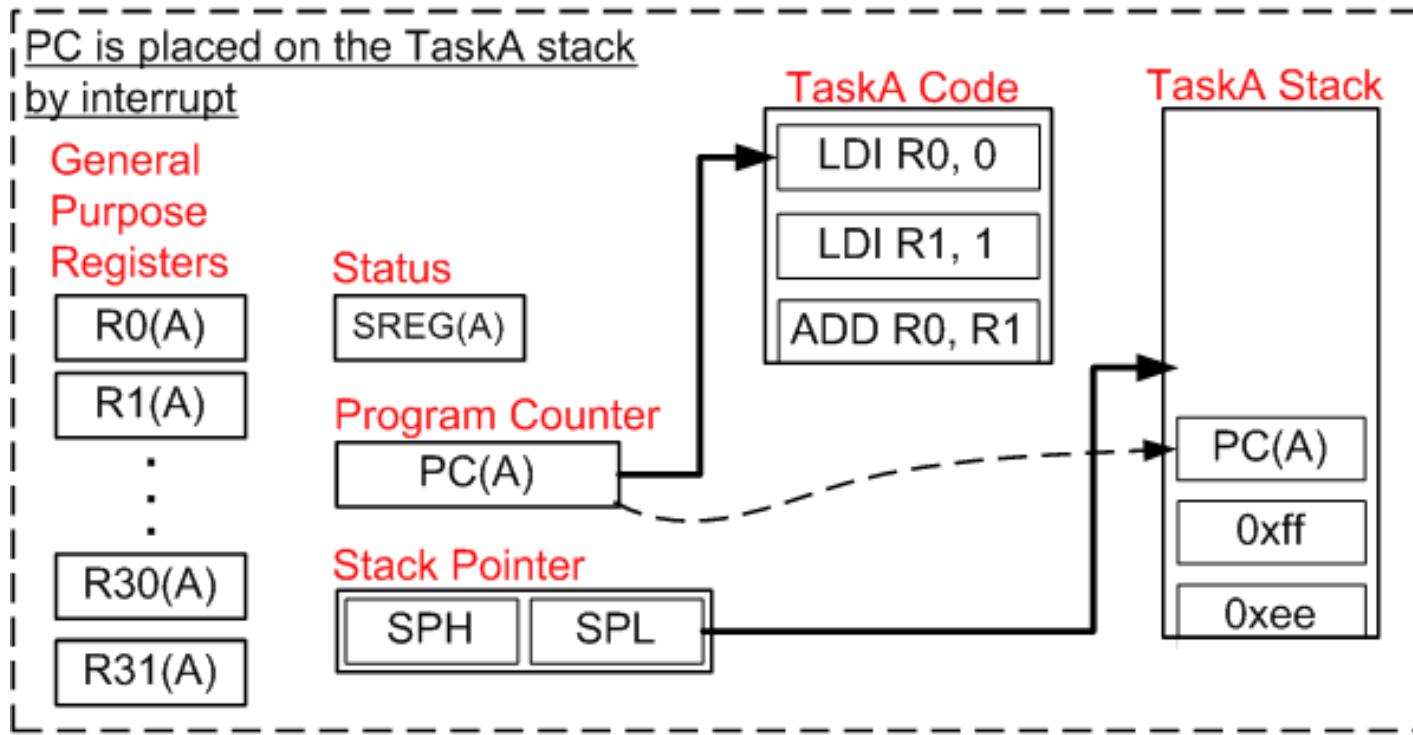
Task Switch AVR und FreeRTOS

Context von Task A



Task Switch AVR und FreeRTOS

Interrupt des Betriebssystem-Timers



Task Switch AVR und FreeRTOS

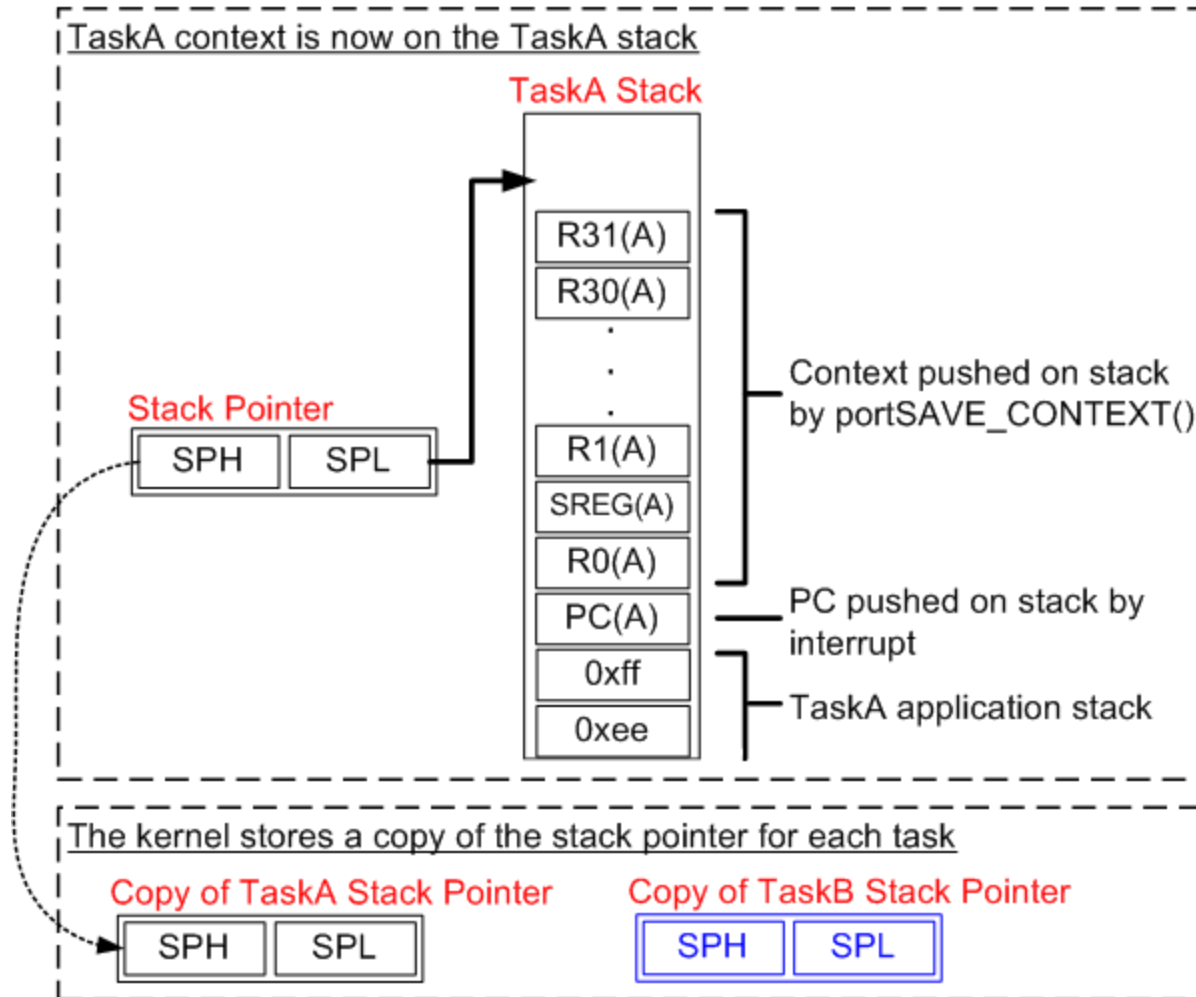
Source Code der ISR des Betriebssystem Timers:

```
/* Interrupt service routine for the RTOS tick.
```

```
*/void SIG_OUTPUT_COMPARE1A( void )  
{  
    vPortYieldFromTick();  
    asm volatile ( "reti" );  
}
```

```
/*-----*/  
void vPortYieldFromTick( void )  
{  
    portSAVE_CONTEXT();  
    vTaskIncrementTick();  
    vTaskSwitchContext();  
    portRESTORE_CONTEXT();  
    asm volatile ( "ret" );}  
/*-----*/
```

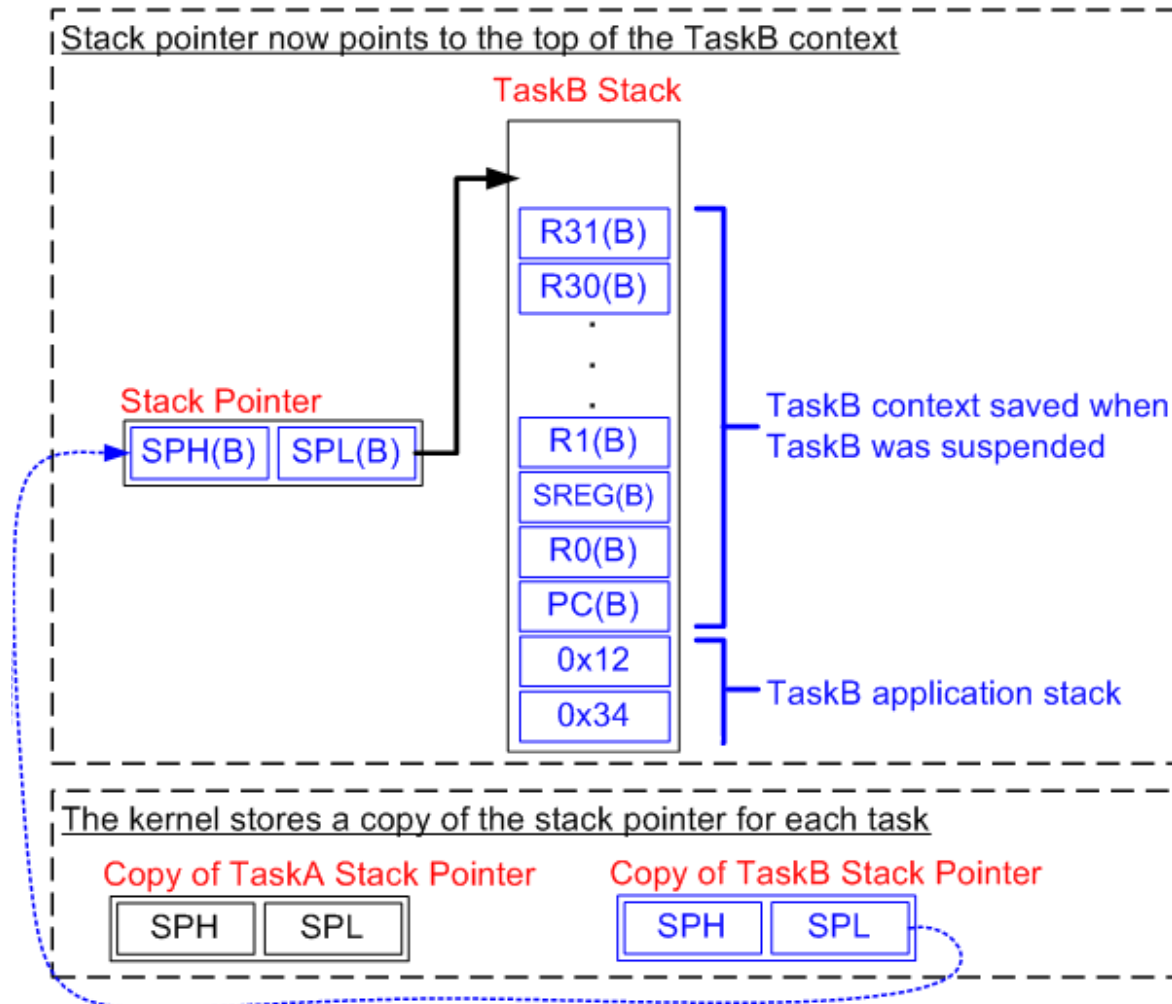
Task Switch AVR und FreeRTOS



Task Switch AVR und FreeRTOS

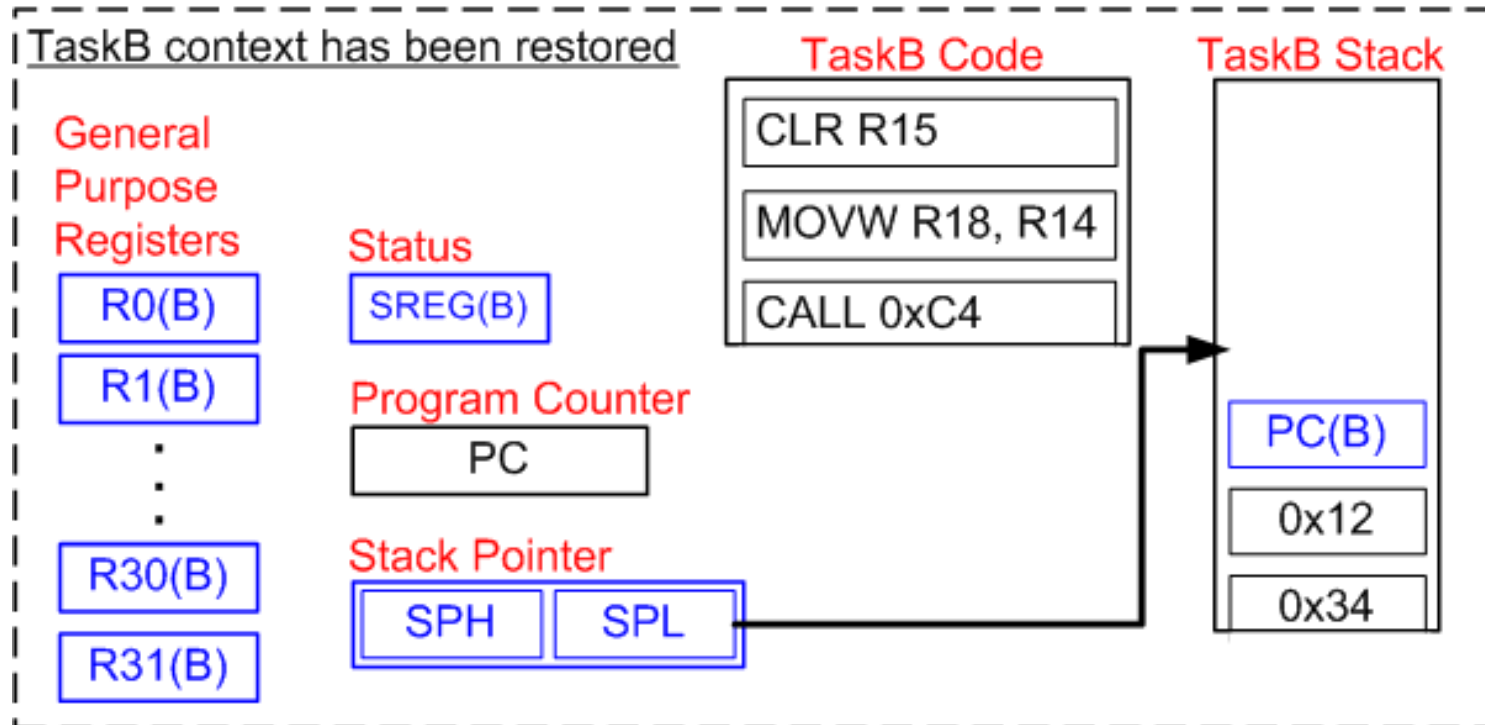
- TaskA-Context auf Stack speichern
 - ▶ `portSAVE_CONTEXT();`
- Erhöhen des TickCounters
 - ▶ `vTaskIncrementTick();`
- Nächsten Task auswählen
 - ▶ `vTaskSwitchContext();`
- Wiederherstellen des Context für TaskB
 - ▶ `portRESTORE_CONTEXT();`

Task Switch AVR und FreeRTOS



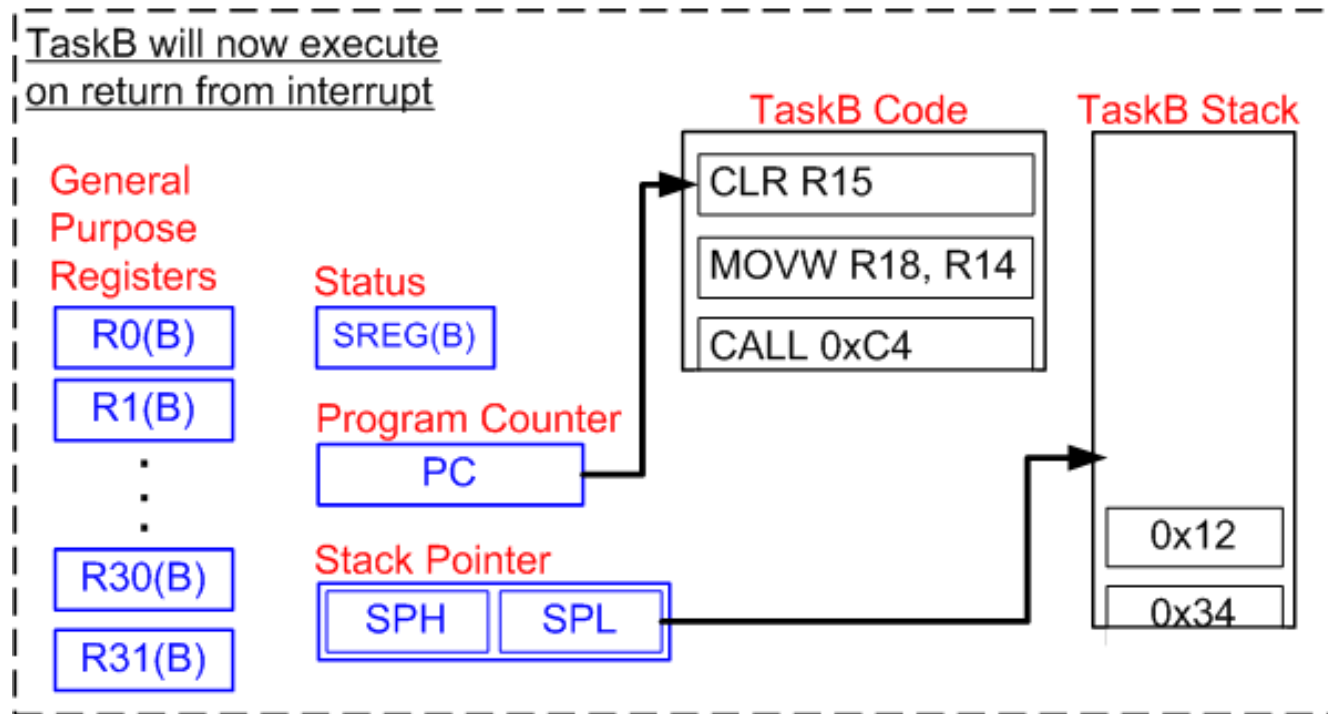
Task Switch AVR und FreeRTOS

Gesamter Context Task B hergestellt



Task Switch AVR und FreeRTOS

Beenden der Betriebssystem-Timer-ISR:



Task Switch AVR und FreeRTOS

- Zusammenfassung:
 - ▶ Task A wird ausgeführt
 - ▶ Unterbrechung von Task durch Betriebssystem-Timer-ISR
 - ▶ Task-Switch (A sichern, B herstellen)
 - ▶ Ende der Betriebssystem-Timer-ISR
 - ▶ Task B wird ausgeführt

Stack – PSP vs. MSP

Process Stack vs. Master Stack

Register	Value
Core	
R0	0x20000090
R1	0x00000000
R2	0xCCCC0006
R3	0xCCCC0007
R4	0x20000090
R5	0x00000000
R6	0xCCCC0006
R7	0xCCCC0007
R8	0xCCCC0008
R9	0xCCCC0009
R10	0xCCCC000A
R11	0xCCCC000B
R12	0xCCCC0008
R13 (SP)	0x20001820
R14 (LR)	0xFFFFFFF9
R15 (PC)	0x0800077C
xPSR	0x0100000F
Banked	
System	
Internal	
Mode	Handler
Privilege	Privileged
Stack	MSP
States	183130
Sec	0.01831300
FPU	

```
180: }
181:

OS_StartLEDBlink.c RTOSInit_STM32F4xx.c SEGGER_SYSVIEW_embOS.c SEGGER_SYSVIEW.c SEGGER_RTT.c SEGGER_SYSVIEW_Config_embOS.c

189 /*****
190 *
191 *     SysTick_Handler()
192 *
193 *     Function description
194 *     This is the code that gets called when the processor receives a
195 *     _SysTick exception. SysTick is used as OS timer tick.
196 */
197 #ifdef __cplusplus
198 extern "C" {
199 #endif
200 void SysTick_Handler(void);    // Avoid warning, SysTick_Handler is not prototyped in any CMSIS header
201 #ifdef __cplusplus
202 }
203 #endif
204 void SysTick_Handler(void) {
205     OS_INT_EnterNestable();
206     OS_TICK_Handle();
207     #if (OS_VIEW_IFSELECT == OS_VIEW_IF_JLINK)
208         JLINKMEM_Process();
209     #endif
210     OS_INT_LeaveNestable();
211 }
212
213 /*****
214 *
```

Stack – PSP vs. MSP

Process Stack vs. Master Stack

- PSP / MSP

- ▶ Tasks im PSP Context
- ▶ Interrupts im MSP Context

Bsp: ISR verschachtelt,
Stack = 300 Bytes, 20 Tasks

- ▶ Zus. Speicherbedarf:
wenige Bytes abhängig von
 μC Architektur

- nur MSP

- ▶ Tasks und ISR im MSP Context
- ▶ Jeder Task muss Stack für ISR zur Verfügung stellen
- ▶ Berechnung Worst Case –
maximale Verschachtelung
der ISR

Bsp: ISR verschachtelt,
Stack = 300 Bytes, 20 Tasks

- ▶ Zus. Speicherbedarf:
 $300 \times 20 = 6.000$ Bytes

Beispielprogramm STM, embos

```

34 #include "RTOS.h"
35 #include "BSP.h"
36
37 static OS_STACKPTR int StackHP[128], StackLP[128]; // Task stacks
38 static OS_TASK      TCBHP, TCBLP;                // Task control blocks
39
40 static void HPTask(void) {
41     while (1) {
42         // BSP_ToggleLED(0);
43         OS_TASK_Delay(50);
44     }
45 }
46
47 static void LPTask(void) {
48     while (1) {
49         // BSP_ToggleLED(1);
50         OS_TASK_Delay(200);
51     }
52 }
53
54 /*****
55 *
56 *     main()
57 */
58 int main(void) {
59     OS_Init(); // Initialize embOS
60     OS_InitHW(); // Initialize required hardware
61     //BSP_Init(); // Initialize LED ports
62     OS_TASK_CREATE(&TCBHP, "HP Task", 100, HPTask, StackHP);
63     OS_TASK_CREATE(&TCBLP, "LP Task", 50, LPTask, StackLP);
64     OS_Start(); // Start embOS
65     return 0;
66 }
67
68 /***** End of file *****/
69

```

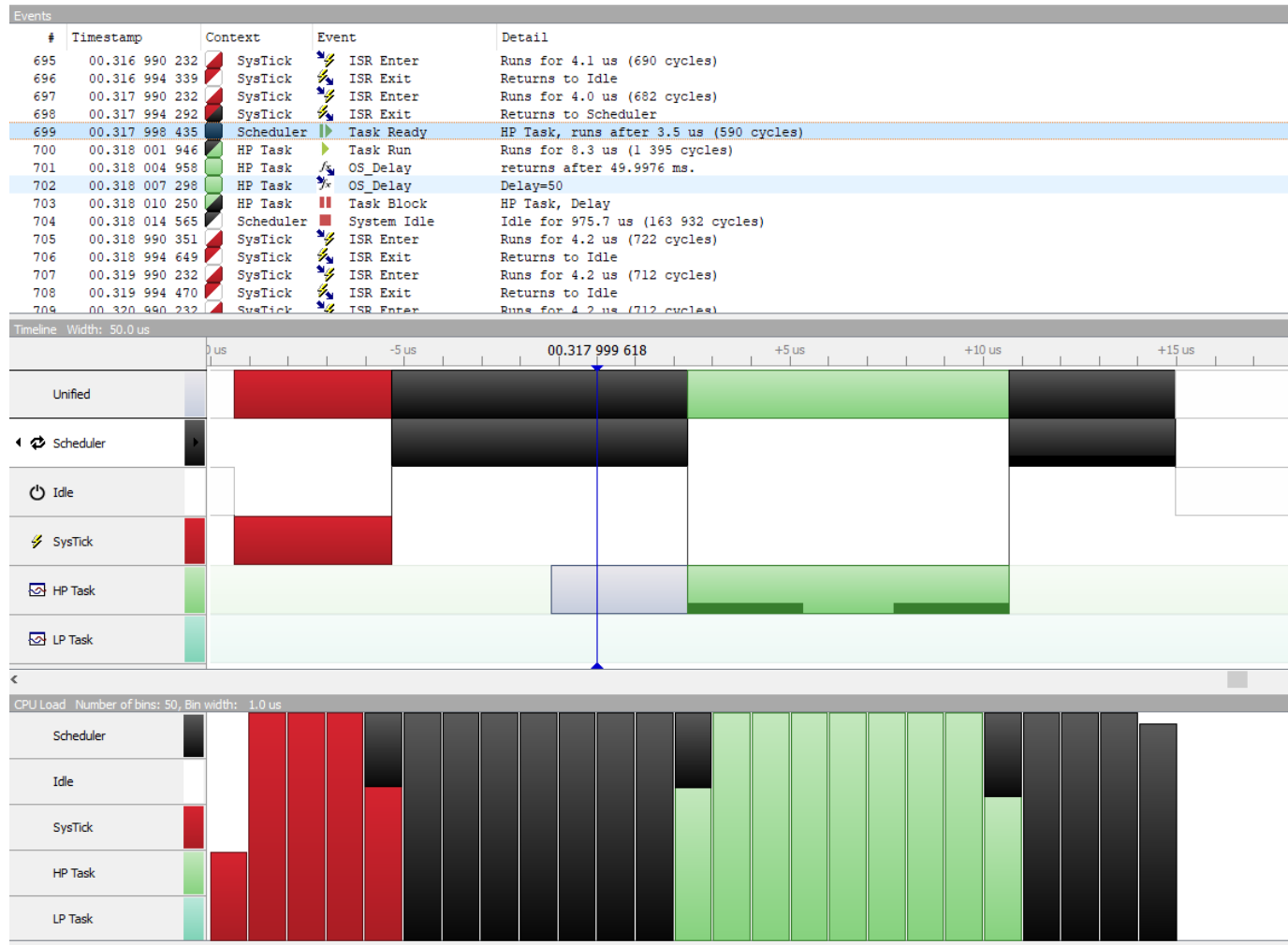
System Information | SEGGER SystemView V2.52b

11208 Events loaded successfully.

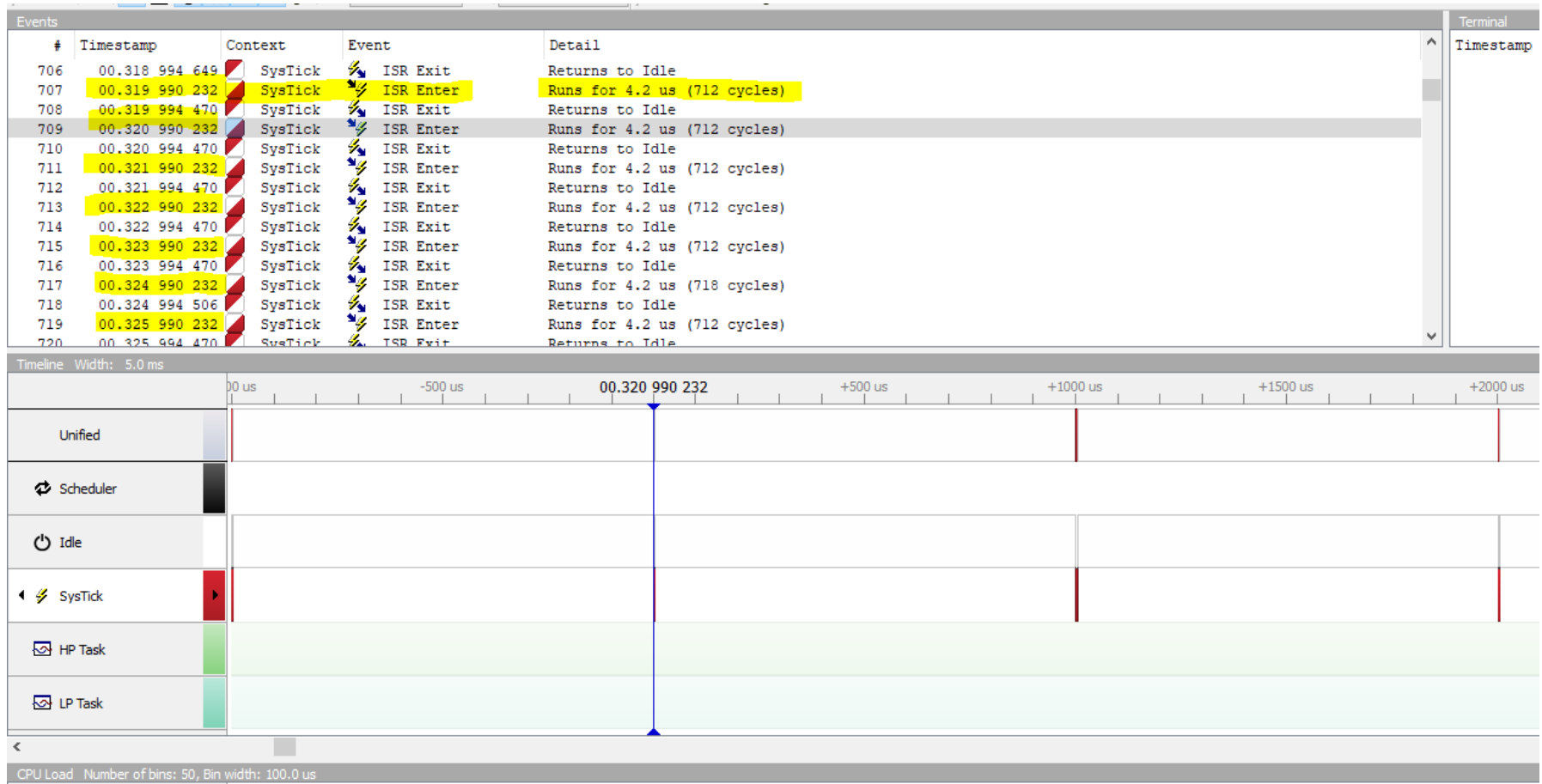
Property	Detail
Target System	
Name	embOS LED Blink
OS	embOS
Modules	No Modules
Device	MK66FN2M0xxx18
Cycle Freq...	168 000 000 Hz
Cycle Period	6 ns
Uptime	10.617 999
Recording	
Host Time	17 Nov 2015 19:12:09
Duration	05.277 975
Title	Minimal Sample
Author	SEGGER Microcontroller
Description	Minimal sample trace recording with embOS. HPTask: Toggle LED 0
Tasks	
Switch Cou...	133
Frequency	25 Hz
ISRs	
Load	0.35%
Frequency	1000 Hz
Total Time	18.2571 ms
Interrupt C...	5 278
Events	
Count	11 208
Frequency	2 120/s
Dropped	0
RTT	
Throughput	8.57 kB/s
Peak Throu...	8.77 kB/s
Trace Size	45.88 kB
Overflows	0

OK

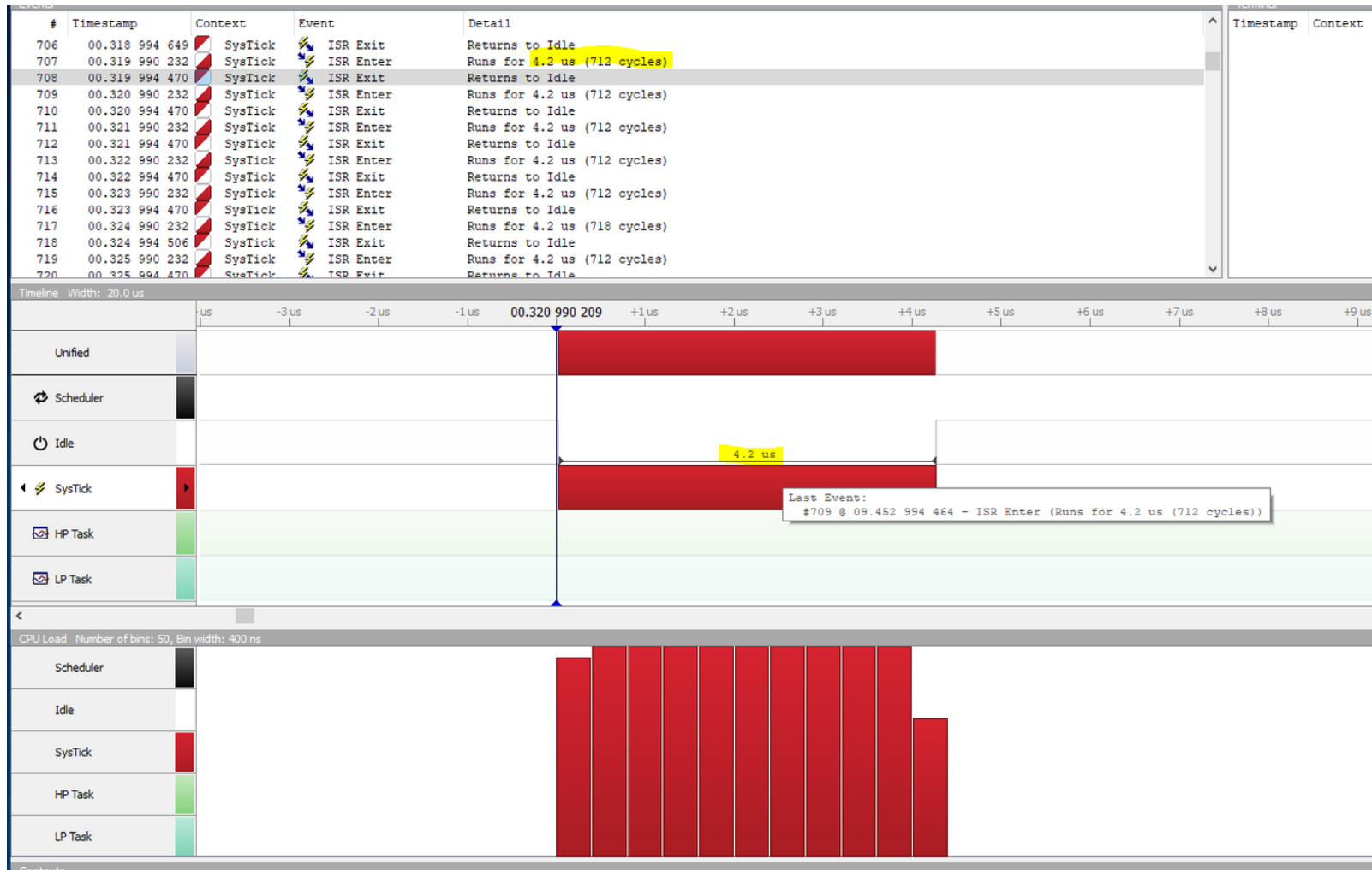
Beispiel Segger System View



SysTick Handler



ISR Enter Funktion



Überlegungen zur RTOS Performance

- (1) Mikroprozessor mit 168 MHz, 1 Zyklus ca. 6ns
 - ▶ Z.B. für High-Performance Anwendungen
- (2) Mikroprozessor mit 1 MHz, 1 Zyklus 1 μ s
 - ▶ Z.B. für Low-Power Anwendungen und Dynamic Frequency Scaling
 - ▶ Energieverbrauch ist linear zur Frequenz bei μ C: $E = kf$
- Dauer Switch ISR: 722 Zyklen, SystemTick = 1ms
 - ▶ Dauer bei (1) 4,2 μ s bei (2) 722 μ s
 - ▶ **Auslastung** bei (1) 0,42% bei (2) 72,2% (ohne ISR Code)

Hinweis: Dauer der 722 Zyklen begründet durch Aktivierung Debug-Interface RTOS

Überlegungen Performance RTOS

- Schlussfolgerung für die Verwendung von RTOS
 - ▶ Bei Systemen mit kleinen Taktraten sehr kritisch
 - Unbedingt Notwendigkeit von Debug-Ausgaben prüfen
 - Berechnung und Abschätzung sehr wichtig
 - Performance Evaluierung am Echtsystem (Task-Switch, Semaphore,..)
 - Evtl. Umstieg auf System ohne RTOS
 - ▶ Bei Systemen mit hohen Taktraten fast kein Performance-Verlust
 - Vorteile von RTOS sehr gut nutzbar



Kurze Wiederholung

- Harte & Weiche Echtzeit
- Synchrone und Asynchrone Programmierung
- Task Switch Ablauf
- Performance -> mehr Overhead durch Betriebssystem

Prioritätsbasiertes preemptives Scheduling

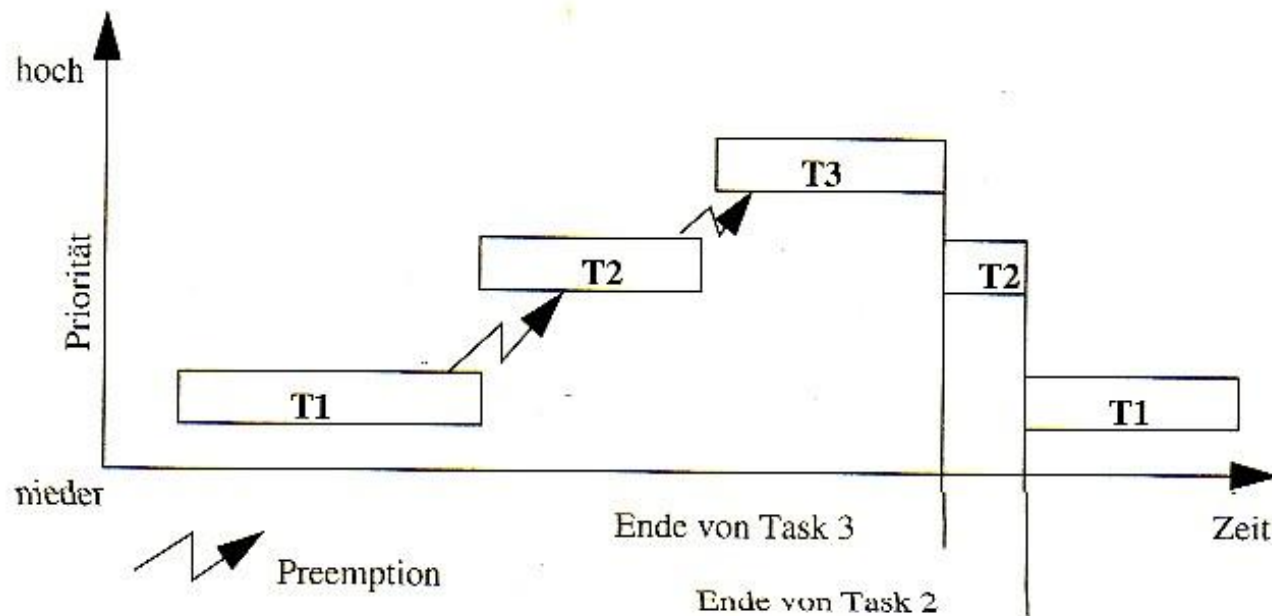


Abb. 3.4 Prioritätsbasiertes preemptives Scheduling

Round Robin Scheduling

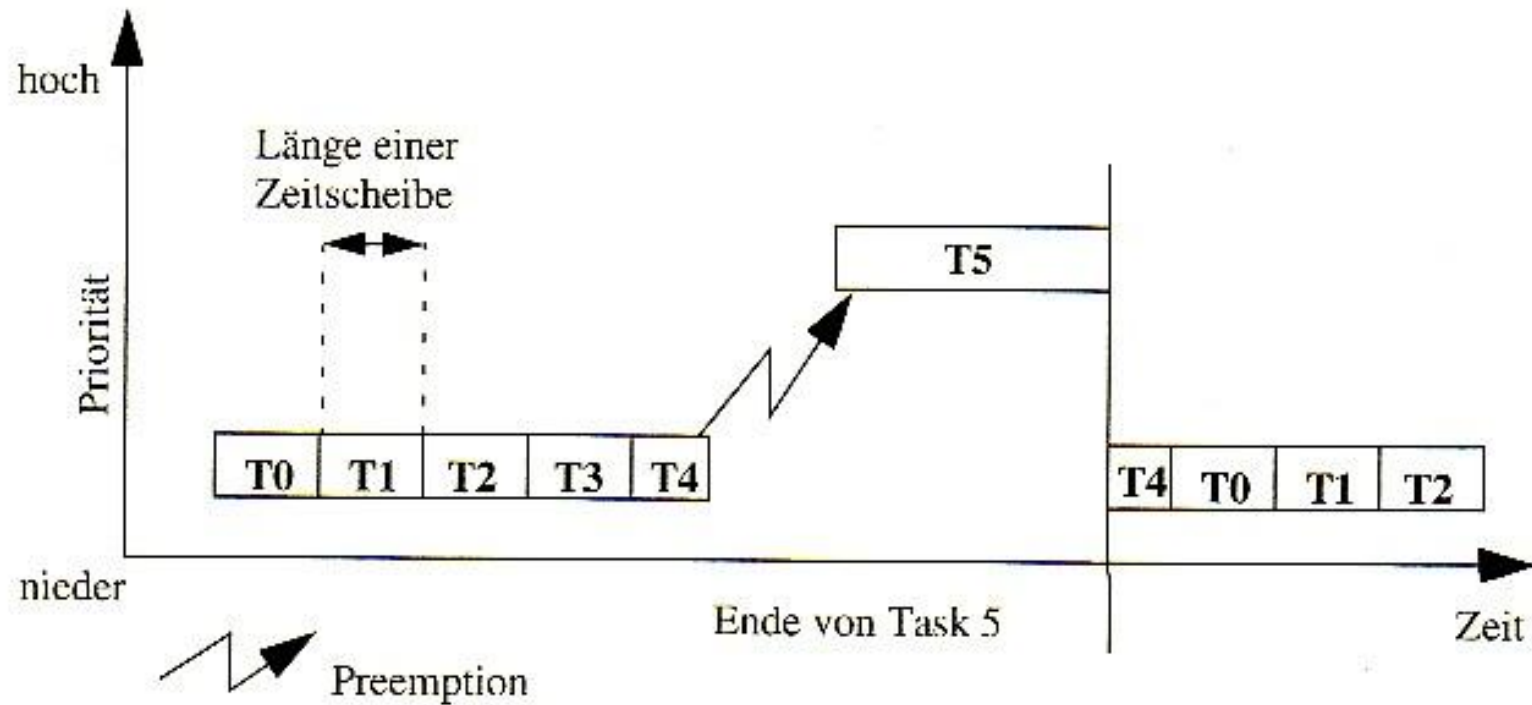


Abb. 3.5 Round Robin Scheduling

Mischsystem Prioritäten und Round Robin

- Gruppen von Tasks haben die gleiche Priorität
- Diese Gruppe von Tasks wird im Round Robin Verfahren abgearbeitet
- Unterbrechung von Tasks mit höherer Priorität möglich

Prioritätsinversion

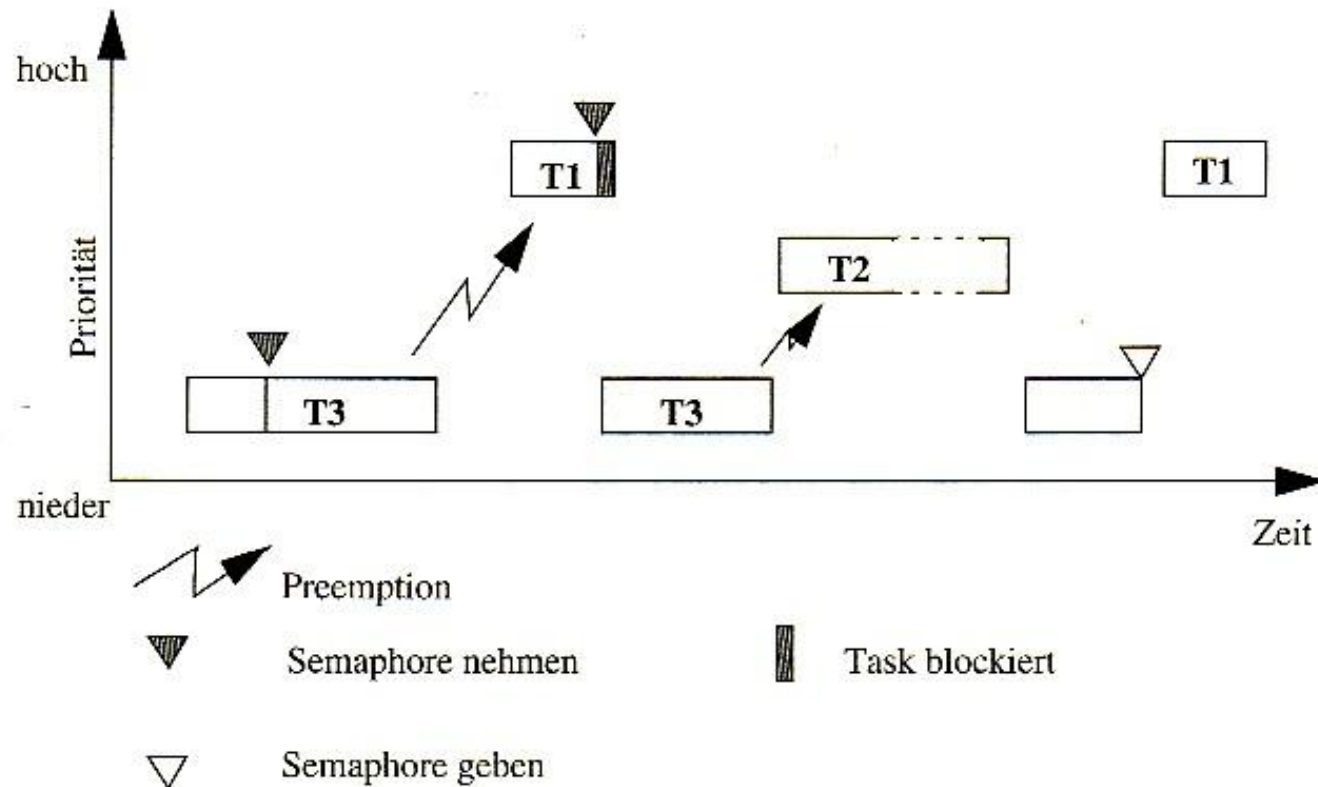
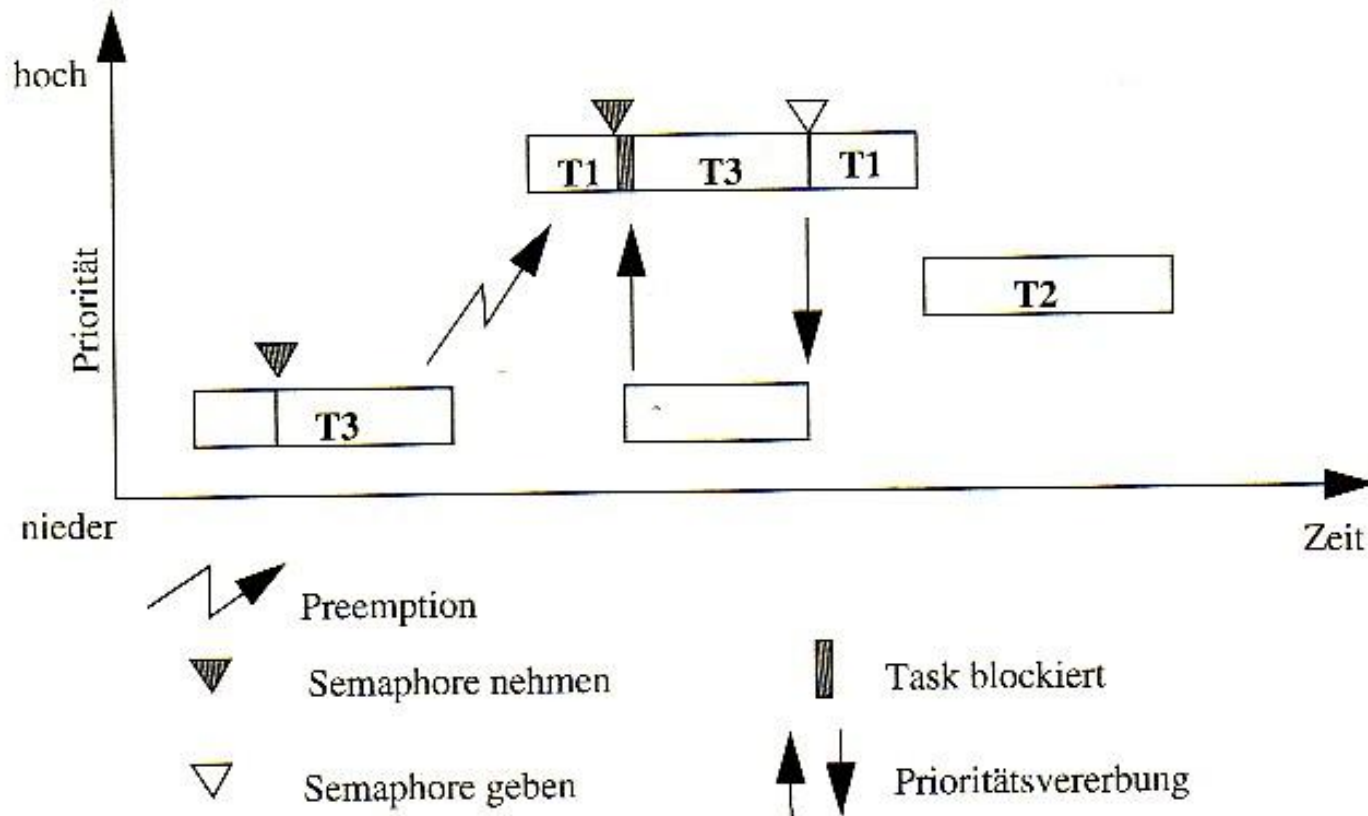


Abb. 3.6 Prioritätsinversion bei Synchronisation durch Semaphoren

Prioritätsvererbung



Zuweisung Tasks

- Jeder Task hat eine Priorität
- Die Auswahl, welche Task in den Zustand Running gesetzt wird, erfolgt nach der einfachen Regel:
 - ▶ Der Scheduler aktiviert den Task mit der höchsten Priorität von allen Tasks, die im Zustand „READY“ sind
 - ▶ D.h. immer wenn ein Task mit höherer Priorität als bisher „READY“ wird, wird er in den Zustand „RUNNING“ gesetzt
- Falls Tasks die gleiche Priorität haben, werden diese im Round Robin Verfahren gescheduled

Priority Boosts Windows

Quelle: Help von Visual Studio

Priority Boosts

- Each thread has a *dynamic priority*. This is the priority the scheduler uses to determine which thread to execute. Initially, a thread's dynamic priority is the same as its base priority. The system **can boost and lower the dynamic priority**, to ensure that it is responsive and that **no threads are starved** for processor time. The system does not boost the priority of threads with a base priority level between 16 and 31. Only threads with a base priority between 0 and 15 receive dynamic priority boosts.

Priority Inversion bei Windows

Quelle: Help von Visual Studio

Priority Inversion

- Priority inversion occurs when two or more threads with different priorities are in contention to be scheduled. Consider a simple case with three threads: thread 1, thread 2, and thread 3. Thread 1 is high priority and becomes ready to be scheduled. Thread 2, a low-priority thread, is executing code in a critical section. Thread 1, the high-priority thread, begins waiting for a shared resource from thread 2. Thread 3 has medium priority. **Thread 3 receives all the processor time, because the high-priority thread (thread 1) is waiting for shared resources from the low-priority thread (thread 2). Thread 2 won't leave the critical section, because does not have the highest priority and won't be scheduled.**

Priority Inversion bei Windows

Quelle: Help von Visual Studio

Priority Inversion

- **Windows:** The scheduler **solves this problem by randomly boosting the priority of the ready threads** (in this case, the low priority lock-holders). The low priority threads run long enough to exit the critical section, and the high-priority thread can enter the critical section. **If the low-priority thread doesn't get enough CPU time to exit the critical section the first time, it will get another chance during the next round of scheduling.**

Windows Embedded Priority Inversion

Quelle: Help von Visual Studio

Priority inversion occurs when a high priority thread is blocked from running because a lower priority thread owns a kernel object, such as a mutex or critical section, that the high priority thread needs. For example, a Windows CE application contains threads 1, 2, and 3. Thread 1 is high priority currently scheduled thread and it tries to acquire a critical section owned by Thread 2. Thread 2 could be in one of two states:

- (1) It could be ready to run
- (2) It could be already blocked because it is waiting on a critical section owned by Thread 3

In case (1), Thread 1 would block and Thread 2 would be boosted to the priority of Thread 1 in hopes that it will release the critical section. In case (2) Thread 2 would be boosted along with Thread 3 and any other thread that was connected in the chain.

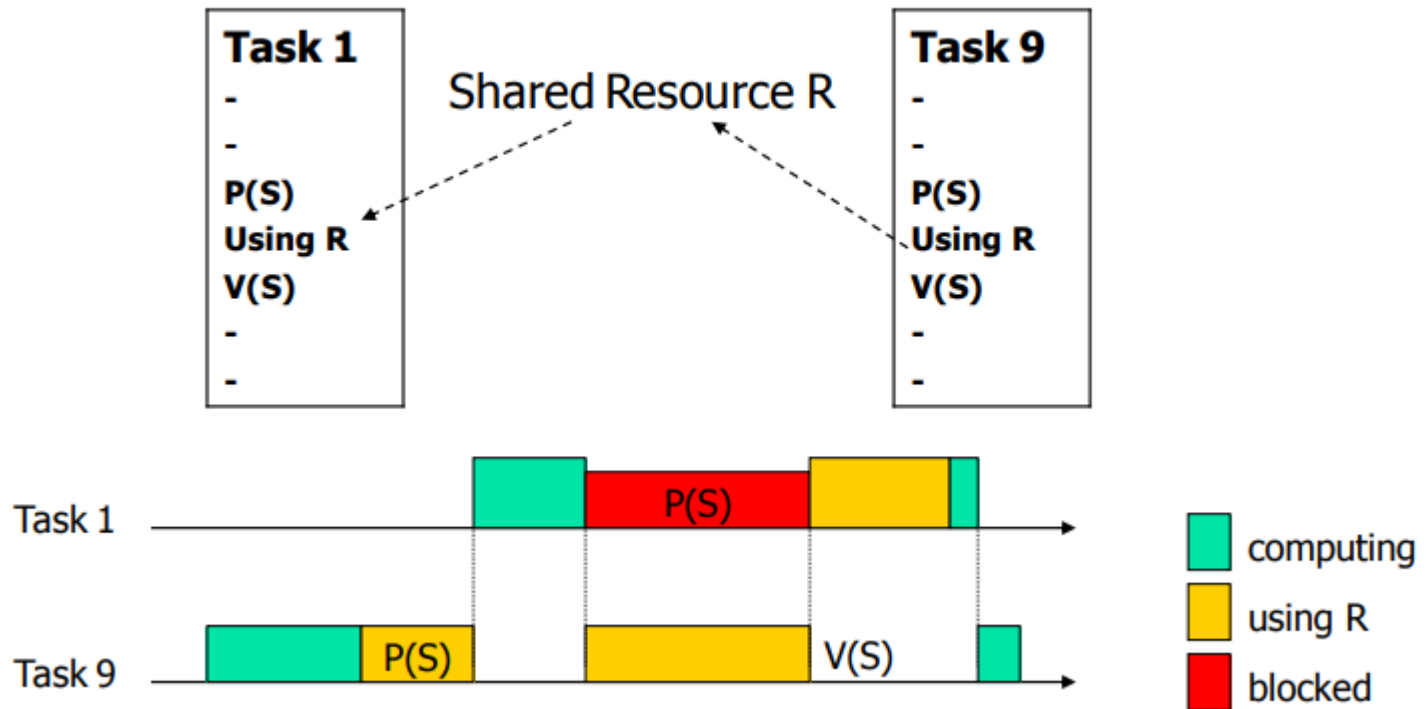
In contrast, Windows CE 3.0 guarantees the handling of priority inversion only to a depth of one level for case (2). Therefore, Thread 1 will block until Thread 2 gives up the critical section. Thread 2 and 3 will be scheduled based on a normal scheduling algorithm. In a real-time system priority inversion cases are to be avoided for the basic reason that you lose control on the scheduling of your threads. **Controlling the exact timing of a thread in a real-time environment is very important.**

Windows als RTOS?

- Keine Prioritätsvererbung
- Zu wenig Prioritäten für Prozesse und Threads
- Powermanagement von Prozessor kann System unterbrechen (abhängig von BIOS)
- Im Gegensatz WinCE: Prioritätsvererbung – 256 Prioritätslevel
- Zufälligkeit anstelle von Determinismus bei Scheduling
- Daher kein RTOS!

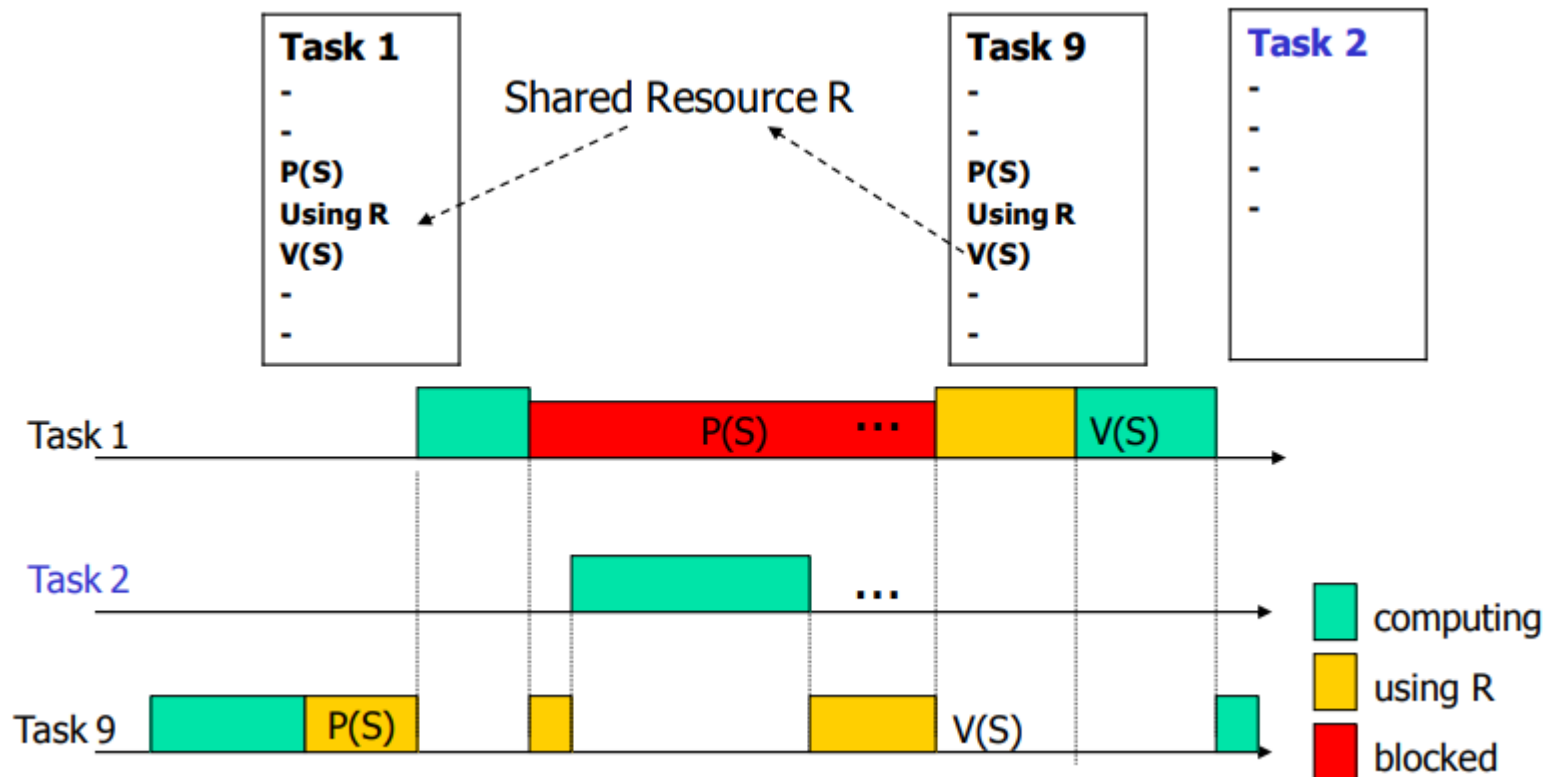
Beispiel: Prioritätsinversion mit 2 Tasks

- ▶ P ... Semaphore (probieren, „pass“)
- ▶ V ... Semaphore erhöhen (verhogen / increase)
- ▶ R ... Ressource



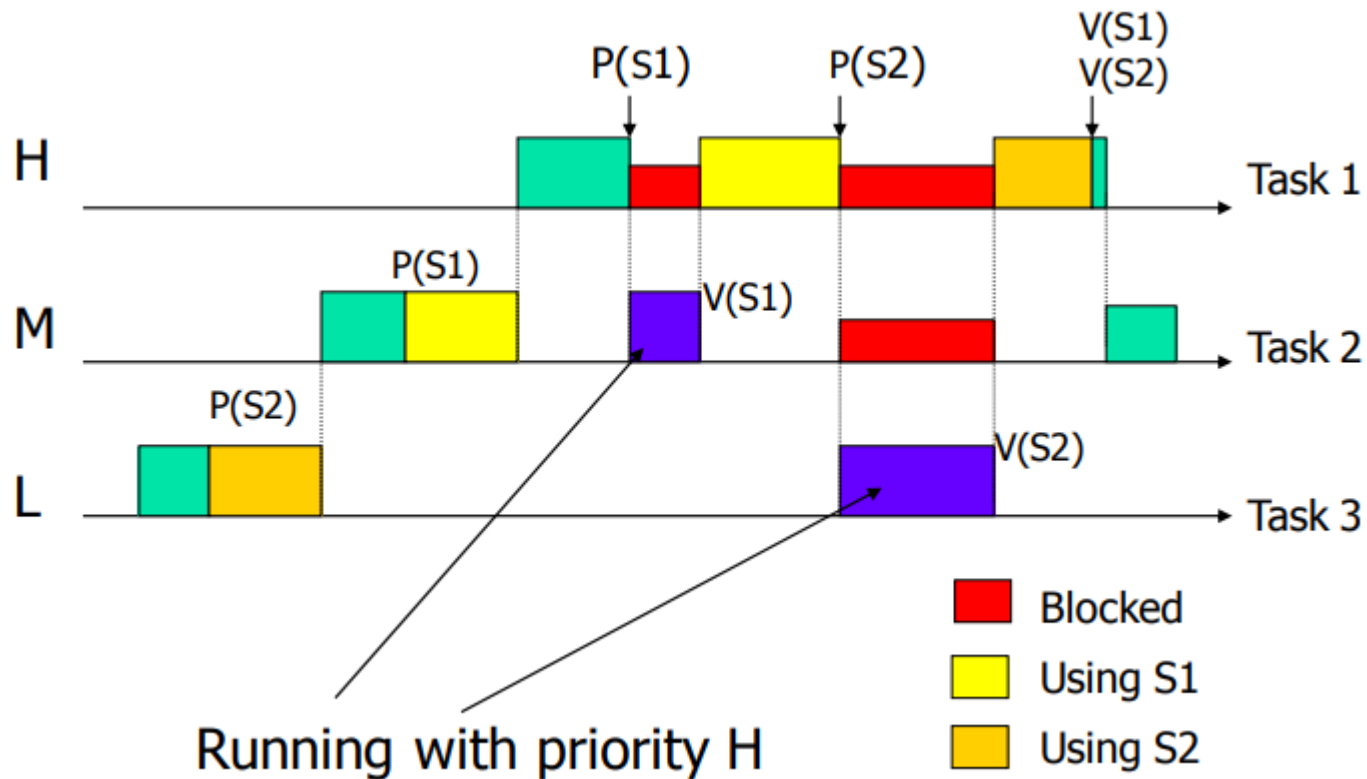
Quelle: <https://www.it.uu.se/edu/course/homepage/realtid/ht10/schedule/synchronization-resource-sharing.pdf>

Prioritätsinversion mit 3 Tasks



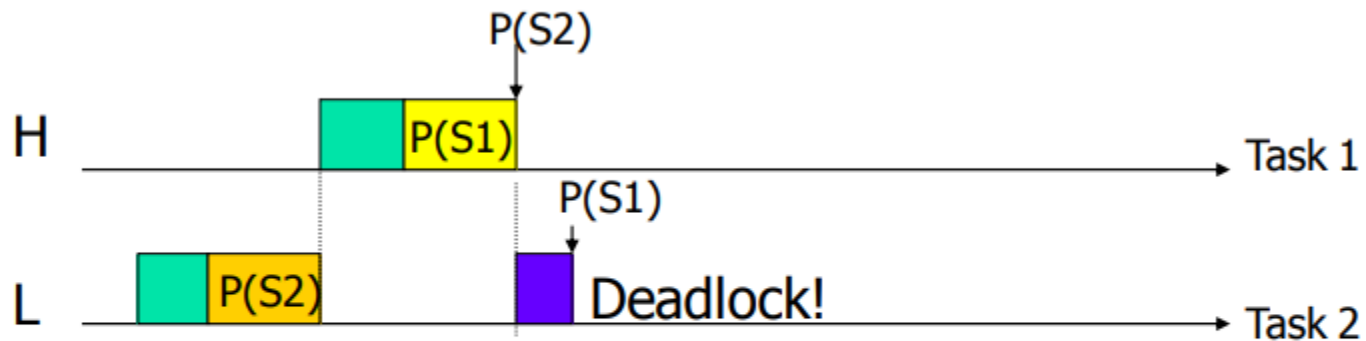
Quelle: <https://www.it.uu.se/edu/course/homepage/realtid/ht10/schedule/synchronization-resource-sharing.pdf>

Basic Inheritance Protocol (BIP)



Quelle: <https://www.it.uu.se/edu/course/homepage/realtid/ht10/schedule/synchronization-resource-sharing.pdf>

BIP – potential deadlock

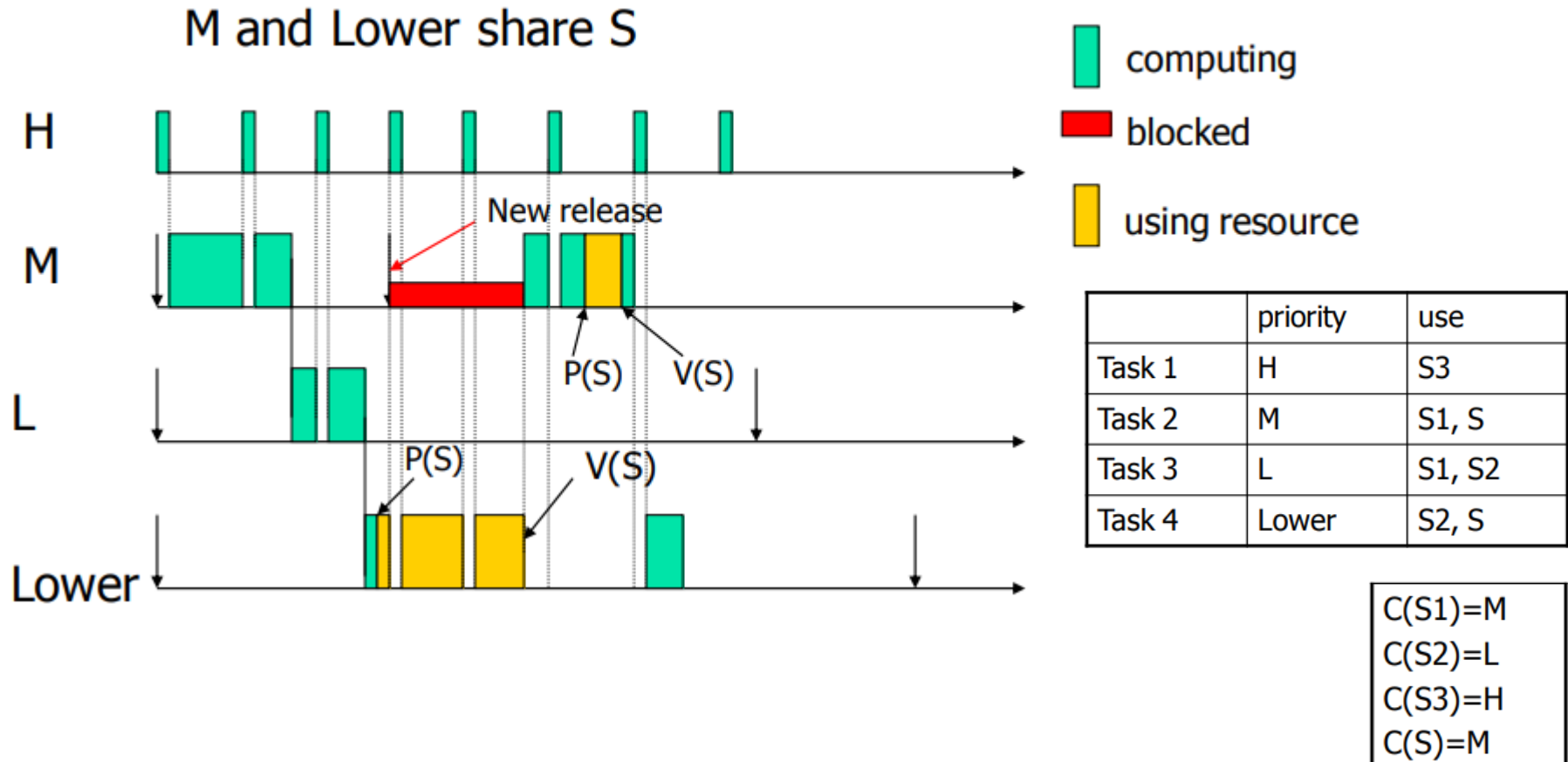


Task 2: ... P(S2) ... P(S1)...

Task 1: ... P(S1) ... P(S2)...

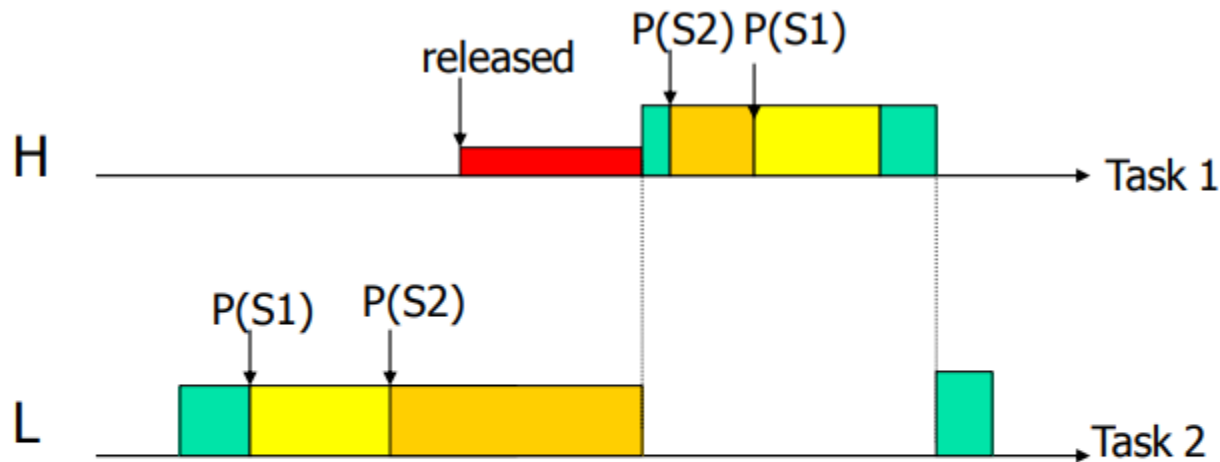
Quelle: <https://www.it.uu.se/edu/course/homepage/realtid/ht10/schedule/synchronization-resource-sharing.pdf>

Highest Locker's Priority (HLP) Protocol



Quelle: <https://www.it.uu.se/edu/course/homepage/realtid/ht10/schedule/synchronization-resource-sharing.pdf>

HLP – deadlock free

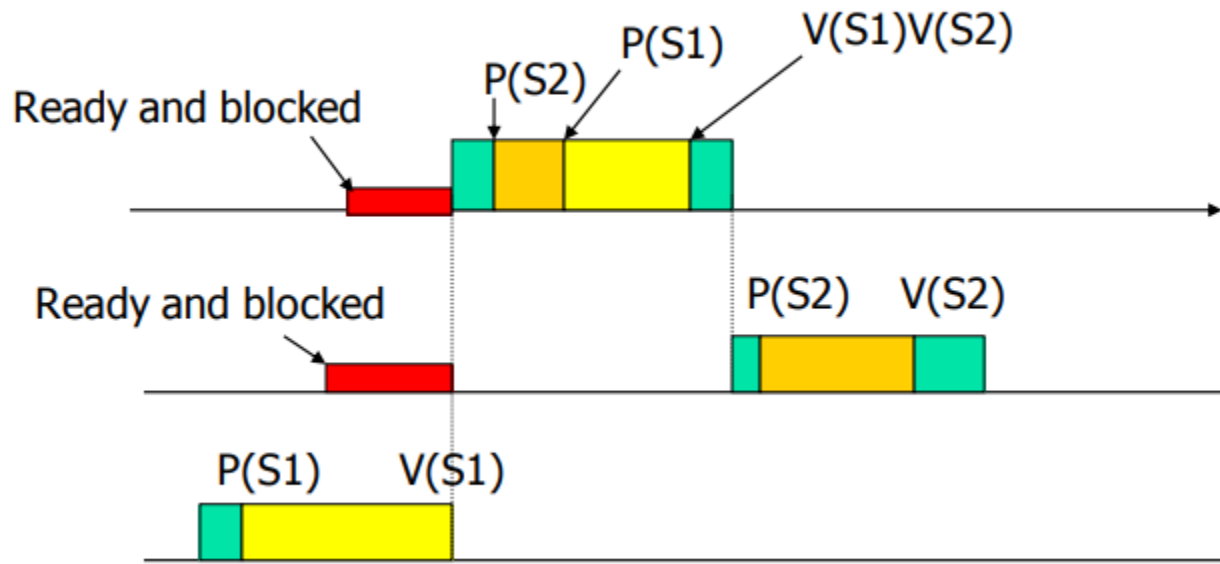


Once task 2 gets S1, it runs with pri H, task 1 will be blocked (no chance to get S2 before task 2)

Quelle: <https://www.it.uu.se/edu/course/homepage/realtid/ht10/schedule/synchronization-resource-sharing.pdf>

HLP – große Blocking-Zeiten

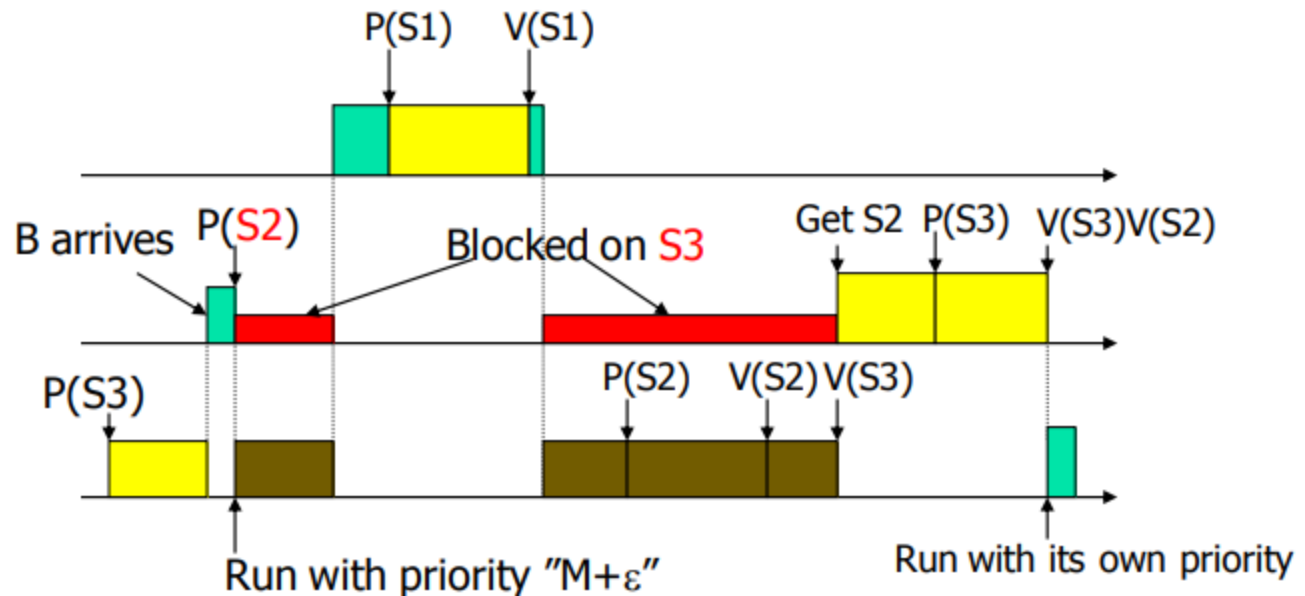
Tasks will be blocked at most once



Quelle: <https://www.it.uu.se/edu/course/homepage/realtid/ht10/schedule/synchronization-resource-sharing.pdf>

Priority Ceiling Protocol

A: ...P(S1)...V(S1)... Prio(A)=H
 B: ...P(S2)...P(S3)...V(S3)...V(S2)... Prio(B)=M C(S1)=H
 C: ...P(S3)...P(S2)...V(S2)...V(S3) Prio(C)=L C(S2)=C(S3)=M



Quelle: <https://www.it.uu.se/edu/course/homepage/realtid/ht10/schedule/synchronization-resource-sharing.pdf>

Vergabe der Prioritäten

Dringlichkeit

- Tasks mit kürzeren Deadlines sollen höhere Priorität bekommen

Gefährlichkeit

- Tasks, die bei Überschreiten einer Deadline irreversible Fehler verursachen können, sollen eine höhere Priorität bekommen

Häufige Implementierungen

- Ad-hoc-Techniken
- Heuristische Vorgehensweise
- Auswirkung:
 - ▶ Hohes Maß an Unvorhersehbarkeit
 - ▶ System kann lange arbeiten,
dann zufällige Fehler
 - ▶ Z.B. Overload Condition (Space Shuttle)
 - ▶ 100% Testabdeckung nicht möglich

Murphy's laws on Real Time

- Murphy's General Law:
 - ▶ If something can go wrong, it will go wrong
- Murphay's Constant Law:
 - ▶ Damage to an object is propotional to its value
- Troutman Postulates:
 - ▶ Any software bug will tend to maximize its damages
 - ▶ The worst software bug will be discovered six months after the field test.

Other laws on Real Time

- Johnson's First Law
 - ▶ If a system stops working, it will do it at the worst possible time.
- Sodd's Second Law
 - ▶ Sooner or later, the worst possible combination of circumstances will happen
 - ▶ A system must always be designed to resist the worst possible combination of circumstances



Josef Langer

Koordinator Masterstudiengang Embedded Systems

Leiter Research Group Embedded Systems

University of Applied Sciences Upper Austria

[josef.langer \(at\) fh-hagenberg.at](mailto:josef.langer@fh-hagenberg.at)