MyOS - Kochbuch für ein Mini-Betriebssystem

C-Implementierung eines eigenen Kernels auf dem Cortex-Mx?

Dipl.-Ing. Univ.
Remo Markgraf

MicroConsult GmbH

While-Loop Ade

Warum ein Mini-Betriebssystem?

Pro

- Änderungsfreundlich, Task ändern und hinzufügen
- Tasks können priorisiert werden um
- Timing einfacher
- Architektur übersichtlicher, Aufgaben in Tasks
- Wiederverwendbarkeit, Copy-Paste leichter

Contra

- Overhead in Flash und RAM
- Auswahl und Einarbeitung RTOS
- Komplexe Parametrisierung für den Anwendungsfall
- Customizing kompliziert/teuer oder nicht möglich

Warum selber machen?

Pro

- Auf Cortex-Mx zugeschnitten und optimiert
- Keine Einarbeiten in ein RTOS
- Keine komplexe Konfiguration des RTOS
- Ressourcenbedarf besser optimierbar
- Exakt auf den Anwendungsfall zugeschnitten
- Einfache Fehlersuche, man weiß genau was wo passiert
- Know-how zu Cortex-Mx verbessert den ganzen Code

Contra

- Viele freie und kommerzielle RTOS auf dem Markt
- Portierbarkeit auf andere Controller Architekturen
- Know-how zu Cortex-Mx und Betriebssystemen
- Zeitaufwand zur Realisierung

MyOS Kochrezept

Man nehme

1 Stück Cortex-Mx und fülle ihn mit der minimalen Task Control Blockstruktur

1 Scheduler aus den vorbereiteten Assemblerteilen und dem eigenen C-Teil zusammenfügen

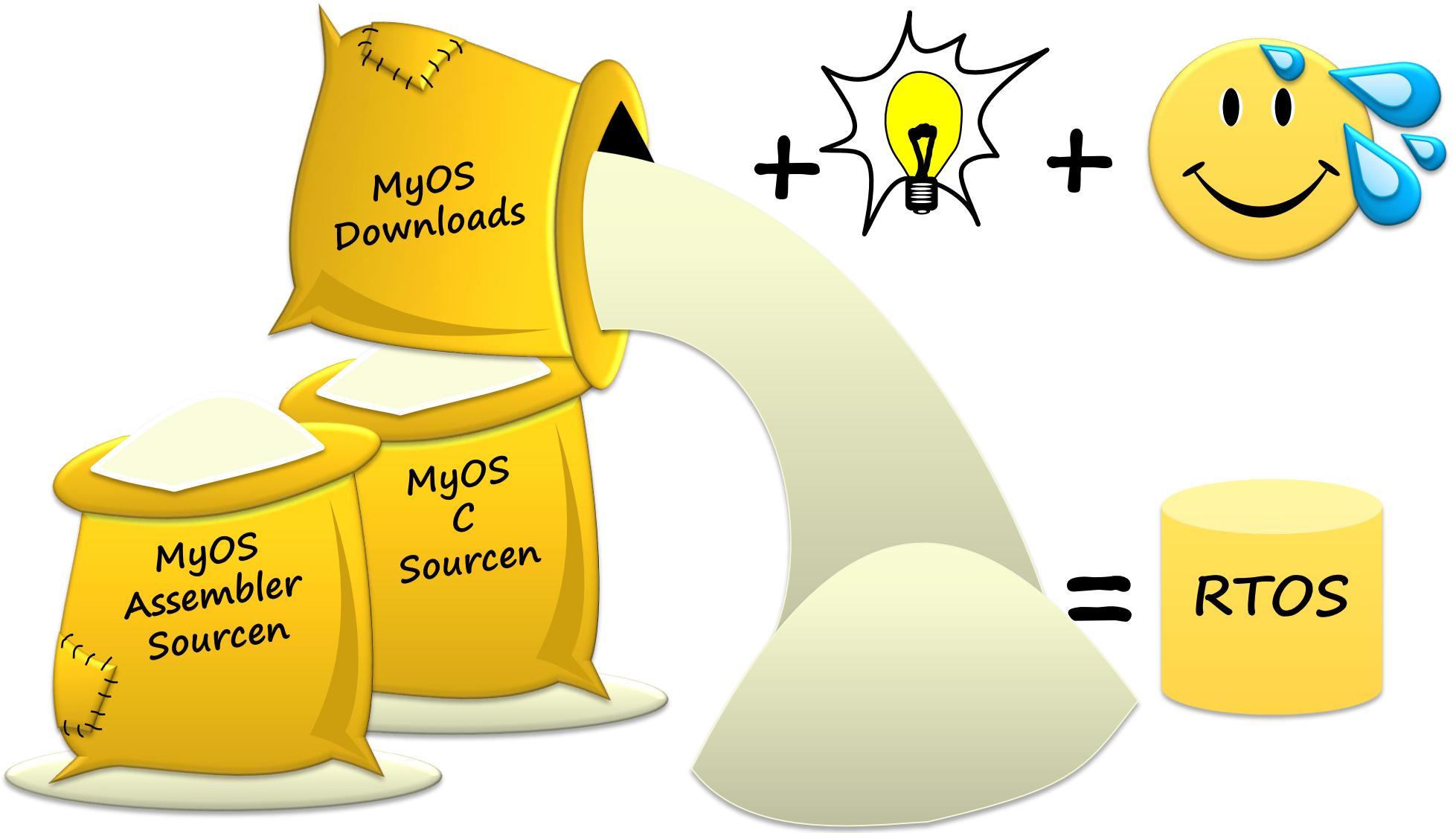
System Call Aufrufe nach Belieben unter ständigem Compilieren stückchenweise hinzugeben

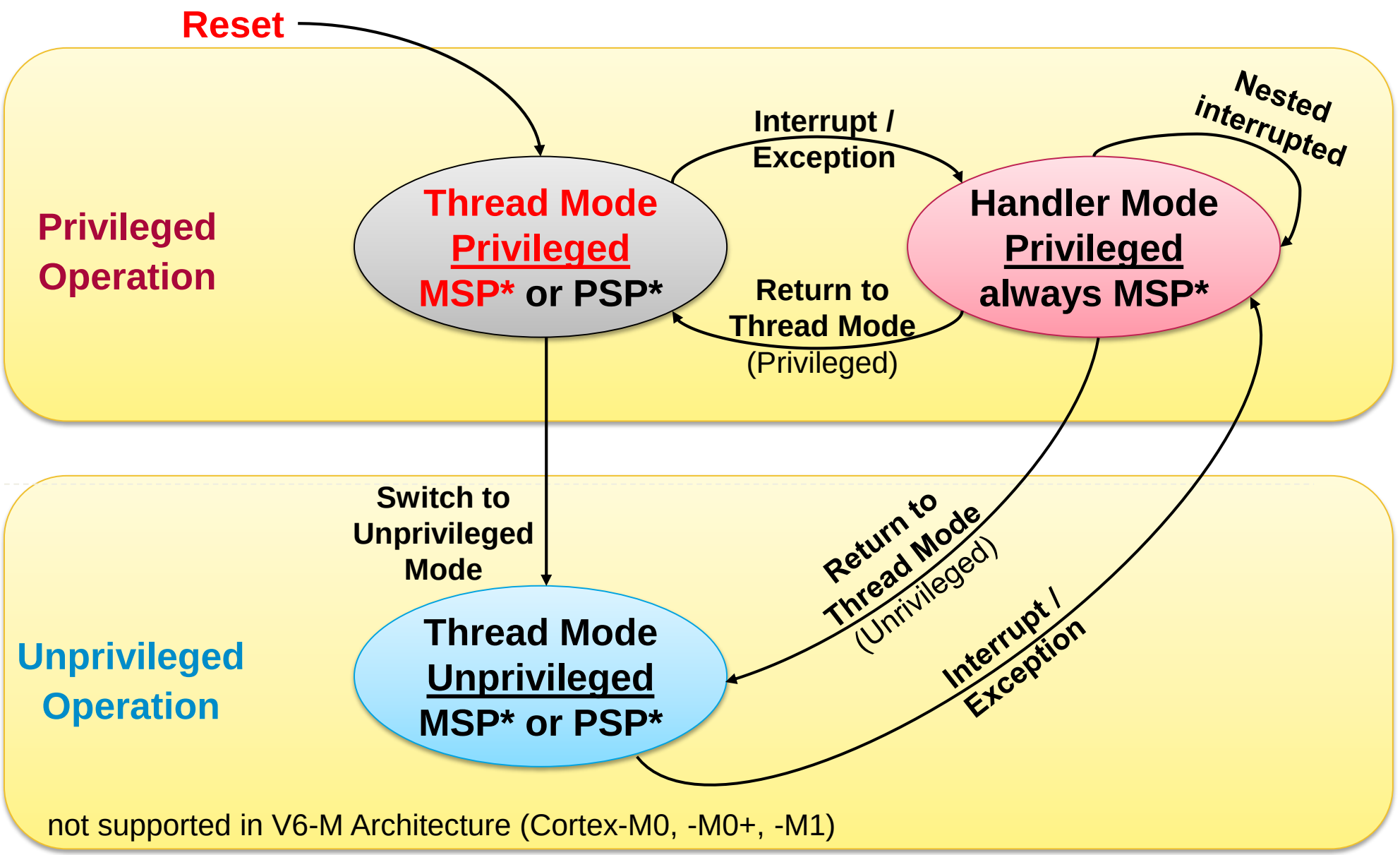
Vorsichtig debuggen und bei Erreichen der gewünschten Funktionalität auch testen.

Als Beilagen eignen sich Prioritäten, Memory Protection, Hardfault Handler und Wait States.

Das Gericht wird mit mehreren Tasks angereichert und reicht für mehrere Projekte.

MyOS Fertigbackmischung



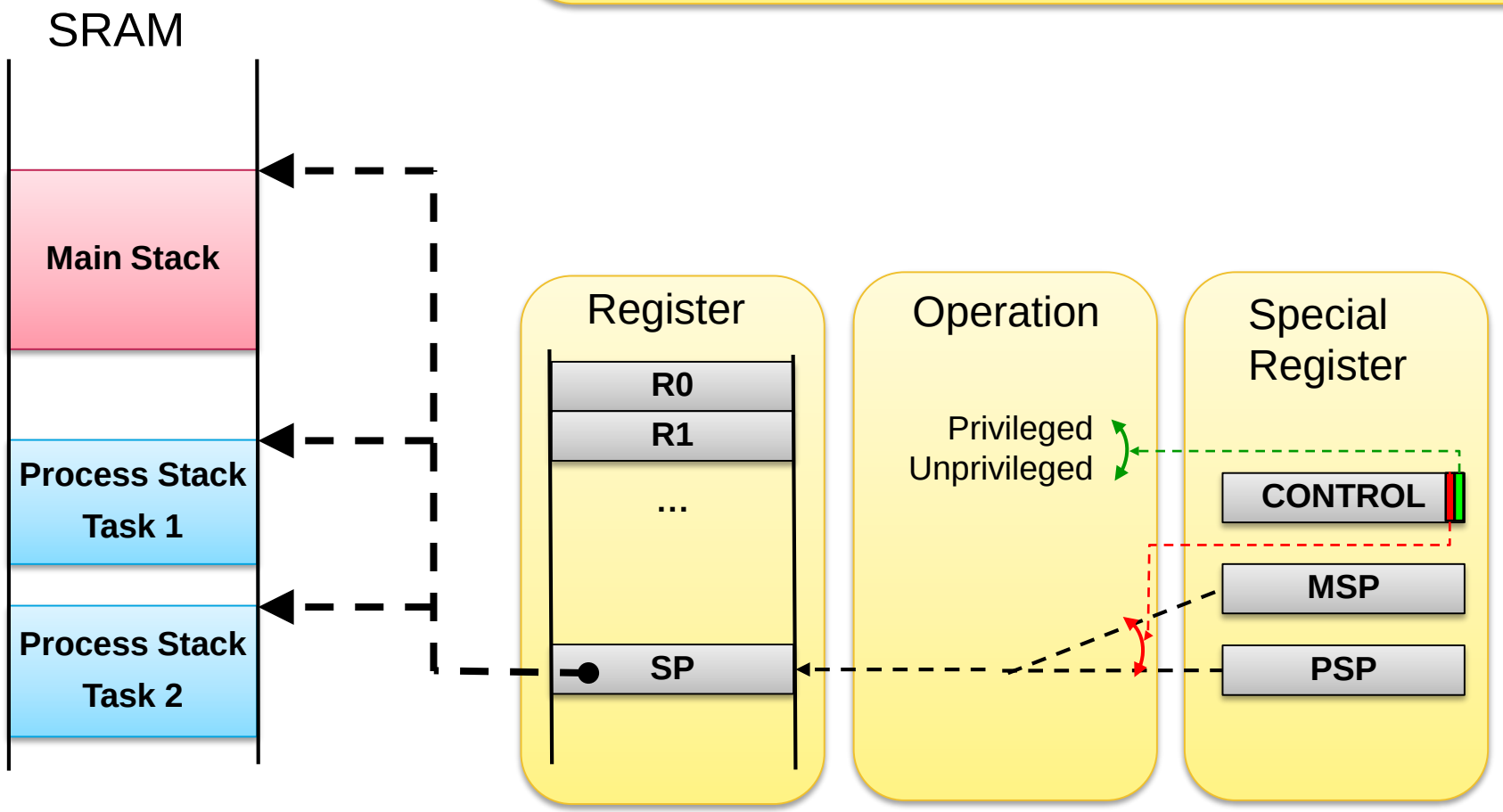


*) MSP = Main Stack, PSP = Process Stack

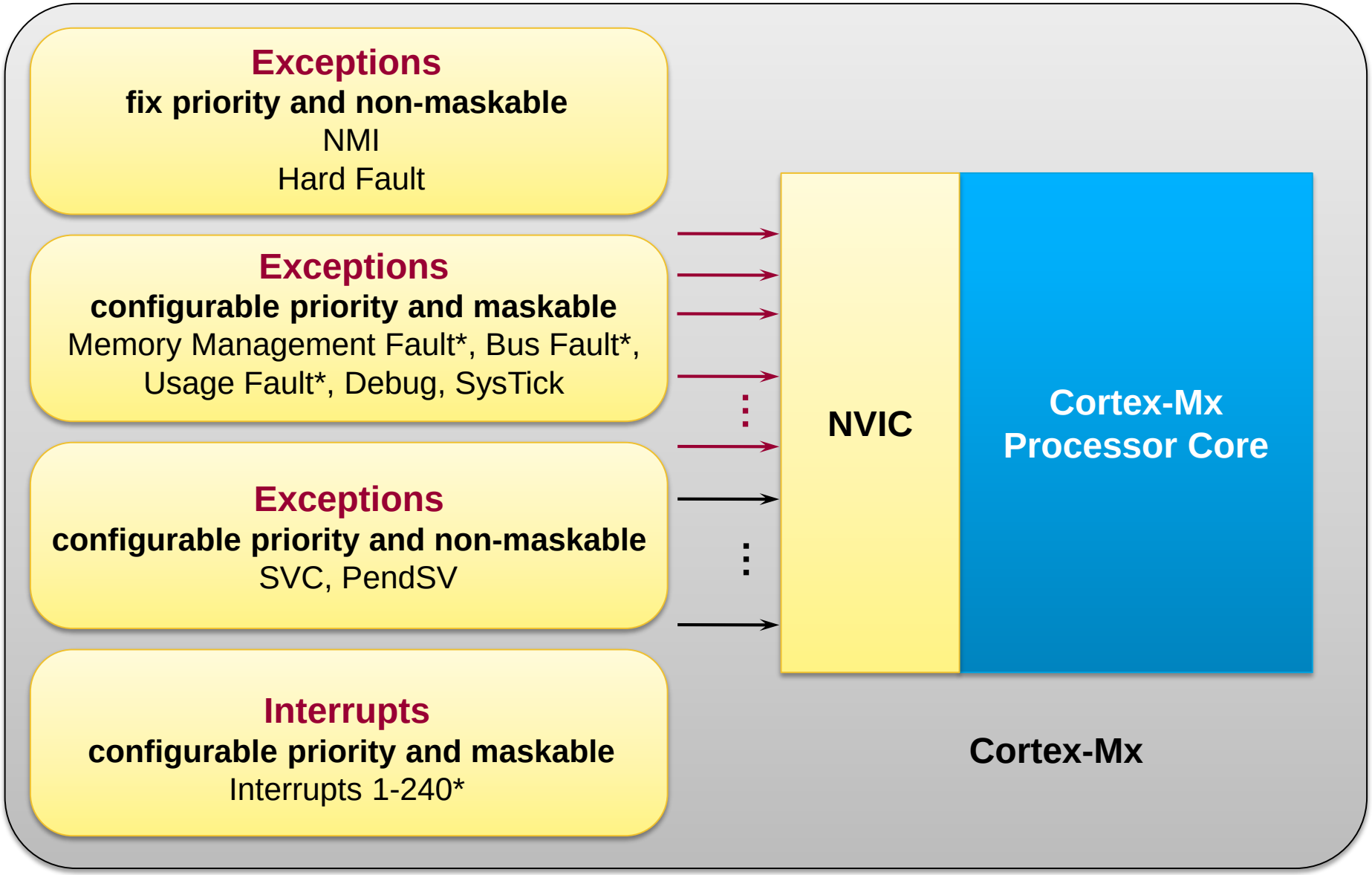
Main Stack / Process Stack

2 umschaltbare Stack Pointer

- MSP, Main Stack Pointer für Interrupt Service Routinen und das OS
- PSP, Process Stack Pointer für Tasks
- Jede Task hat einen eigenen Stack-Bereich



Exceptiones/Interrupts



*) ARMv6-M (Cortex-M0, M0+, M1): no Local Faults (Memory Management Fault, Bus Fault, Usage Fault), 1-32 Interrupts

NVIC

- Nested Vectored Interrupt Controller
- Tabelle mit Handler Adressen
- Interrupts haben Prioritäten
- Niederpriore Interrupts können von höherprioren unterbrochen werden

Vector Table Offset Register **

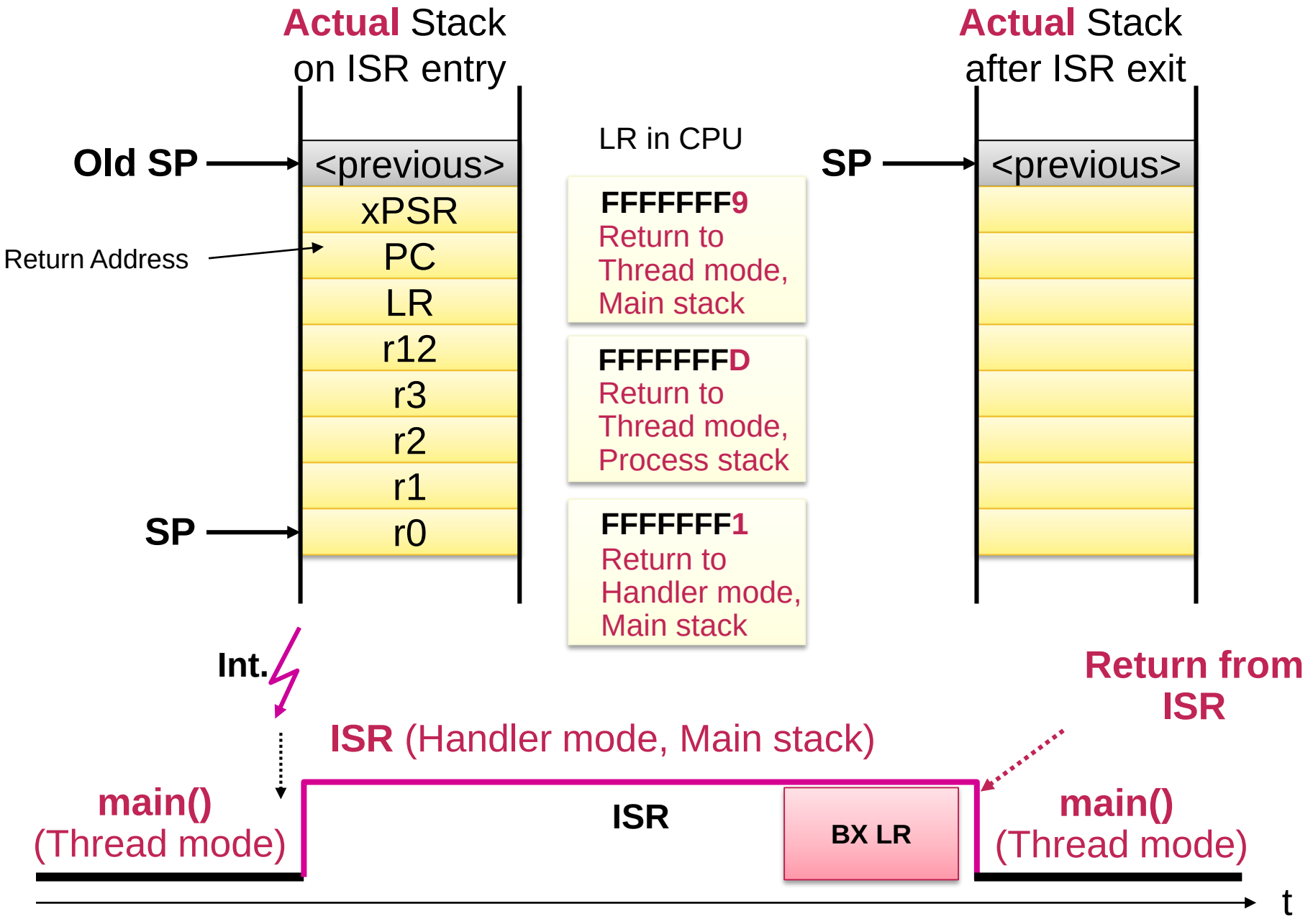
Address: 0xE000ED08

Reset Value: 0x0000'0000

Vector Table

Address		Vector #
0x40 + 4*N	External N	16 + N
…	…	…
0x40	External 0	16
0x3C	SysTick	15
0x38	PendSV	14
0x34	Reserved	13
0x30	Debug Monitor *	12
0x2C	SVCall	11
0x1C to 0x28	Reserved	7-10
0x18	Usage Fault *	6
0x14	Bus Fault *	5
0x10	Mem Manage Fault *	4
0x0C	Hard Fault	3
0x08	NMI	2
0x04	Reset	1
0x00	Initial Main SP	N/A

*) ARMv6-M (Cortex-M0, M0+, M1): no Local Faults (Memory Management Fault, Bus Fault, Usage Fault)
**) ARMv6-M (Cortex-M0, M0+, M1): not present, Cortex-M3 r2p0 and earlier: bit [31:30] are reserved, read as 0


Actual Stack on ISR entry
Old SP
<previous>
xPSR
Return Address
PC
LR
r12
r3
r2
r1
SP
r0
LR in CPU
FFFFFFF9
Return to Thread mode, Main stack
FFFFFFFD
Return to Thread mode, Process stack
FFFFFFF1
Return to Handler mode, Main stack
Actual Stack after ISR exit
SP
<previous>
Int.
Return from ISR
ISR (Handler mode, Main stack)
main()
(Thread mode)
ISR
BX LR
main()
(Thread mode)
t

AAPCS
Advanced ARM
Procedure Call Standard

Register

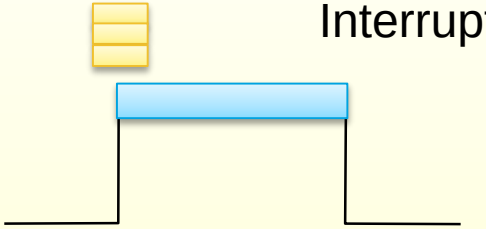
r0
r1
r2
r3

Funktionsparameter
dürfen von der aufgerufenen Funktion verändert werden

Lokale Variablen,
Register müssen von der aufgerufenen Funktion gesichert werden

r4
r5
r6
r7
r8
r9
r10
r11

Kontext Sicherung bei Interrupt



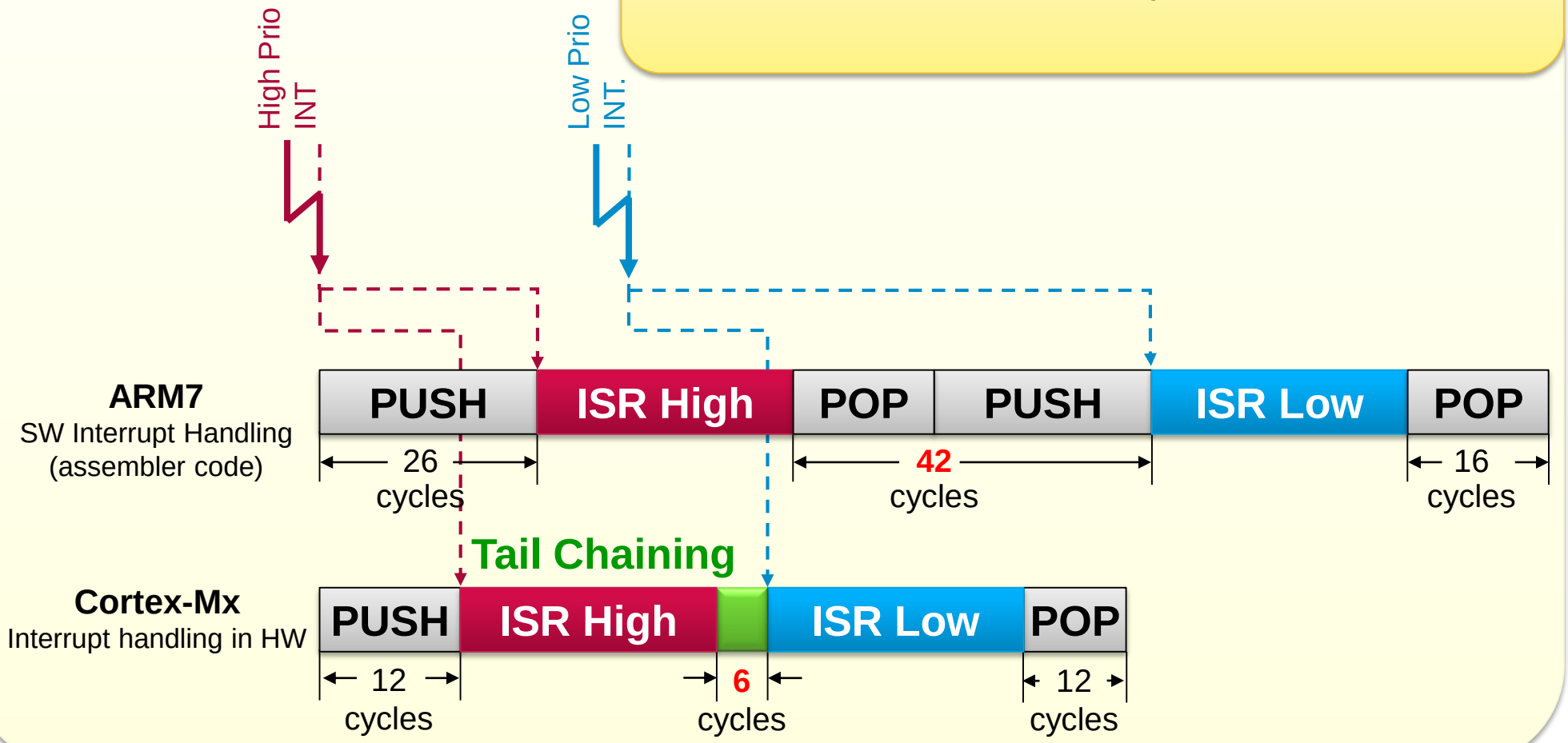
r12

Scratch Register
darf von der aufgerufenen Funktion verändert werden

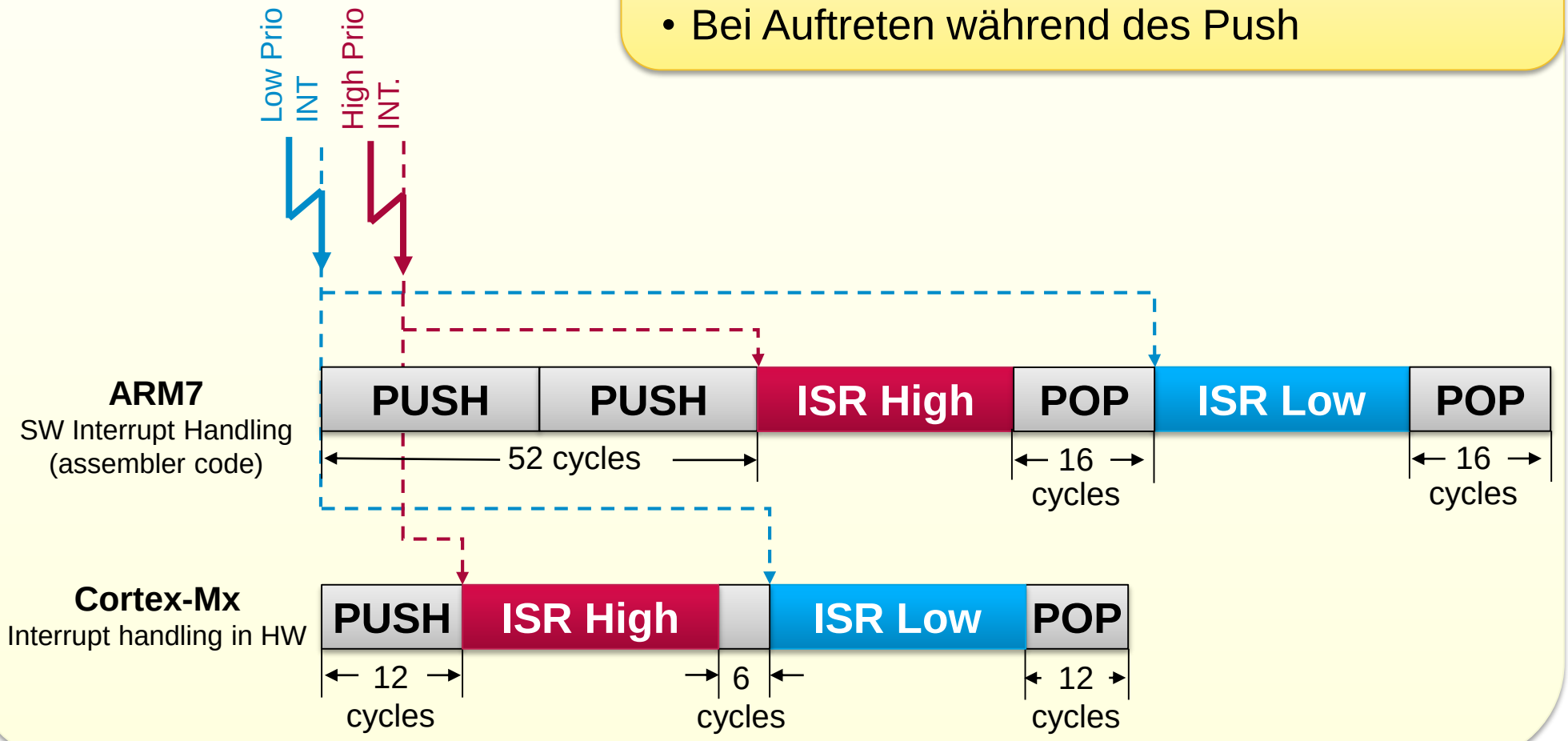
r13/SP — Stack Pointer
r14/LR — Link Register
r15/PC — Program Counter

t

Tail Chaining:
- Anhängen von Low Prio Interrupt Handling
- Ohne erneutes Push-Pop



Late Arrival:
- Vorziehen eines High Prio Interrupt Handlings
- Ohne erneutes Push-Pop
- Bei Auftreten während des Push



SysTick

PendSV

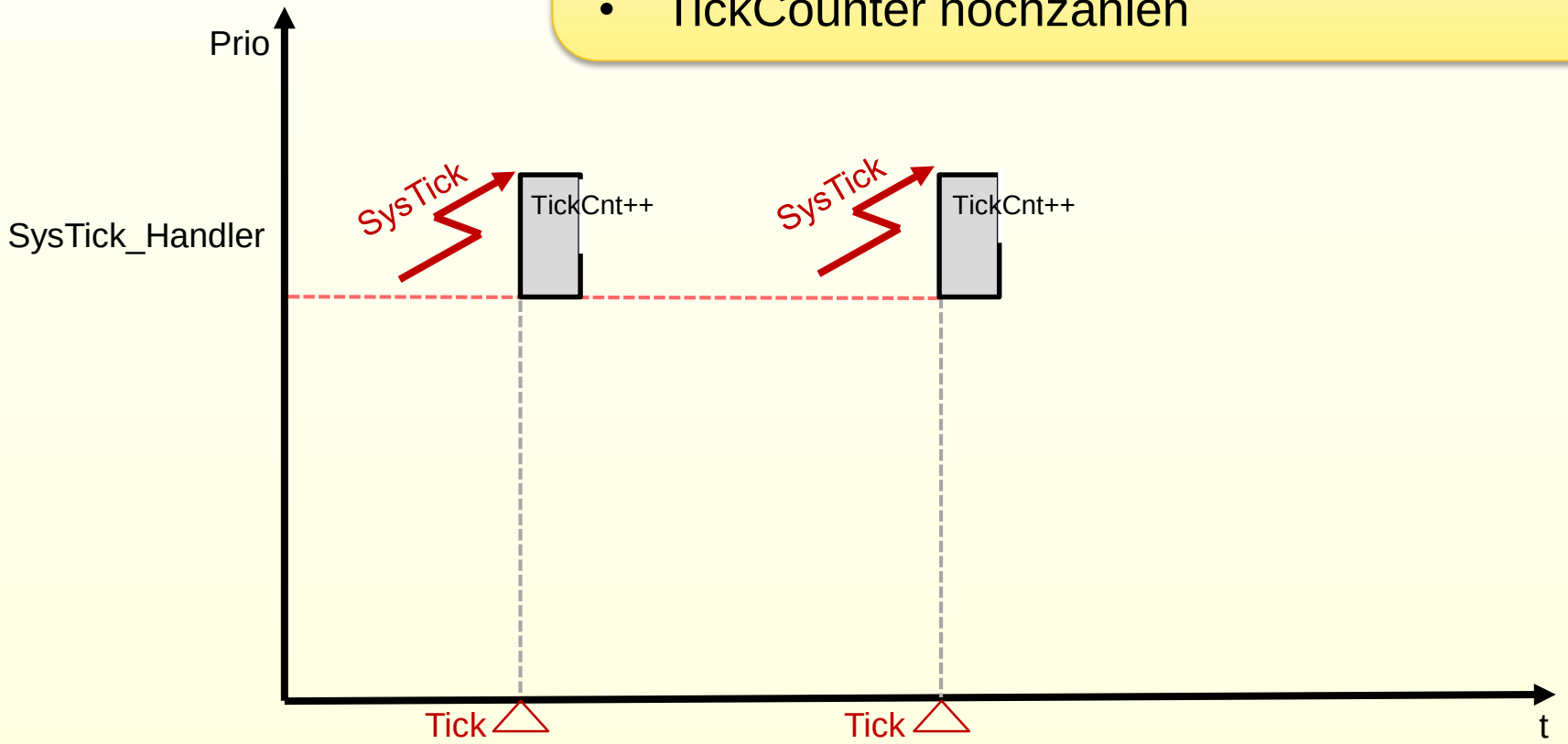
SVC

MyOS

Exceptiones & Interrupts

OS-Events

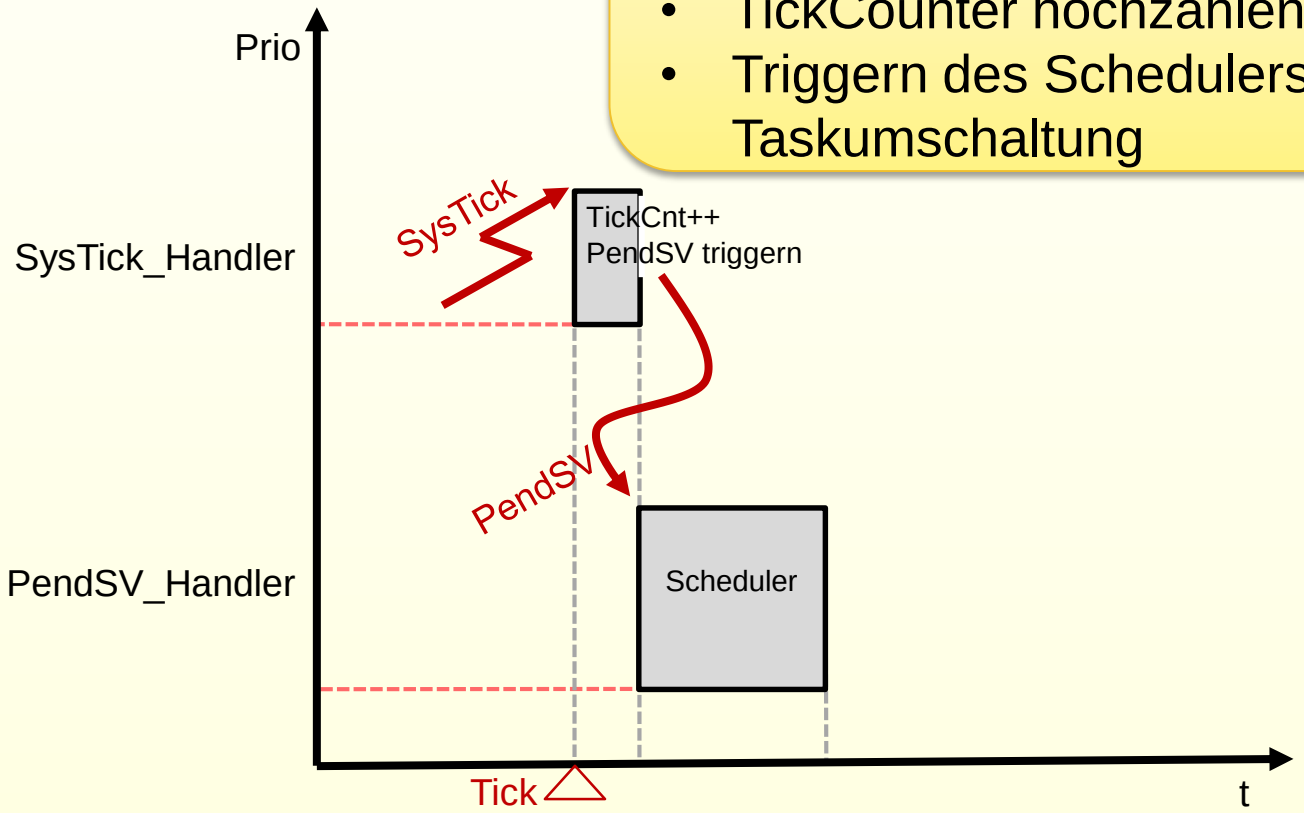
SysTick:
- Grundlegendes Timing des Systems
- Wiederkehrende Interruptfolge von z.B. 5ms
- TickCounter hochzählen



OS-Events

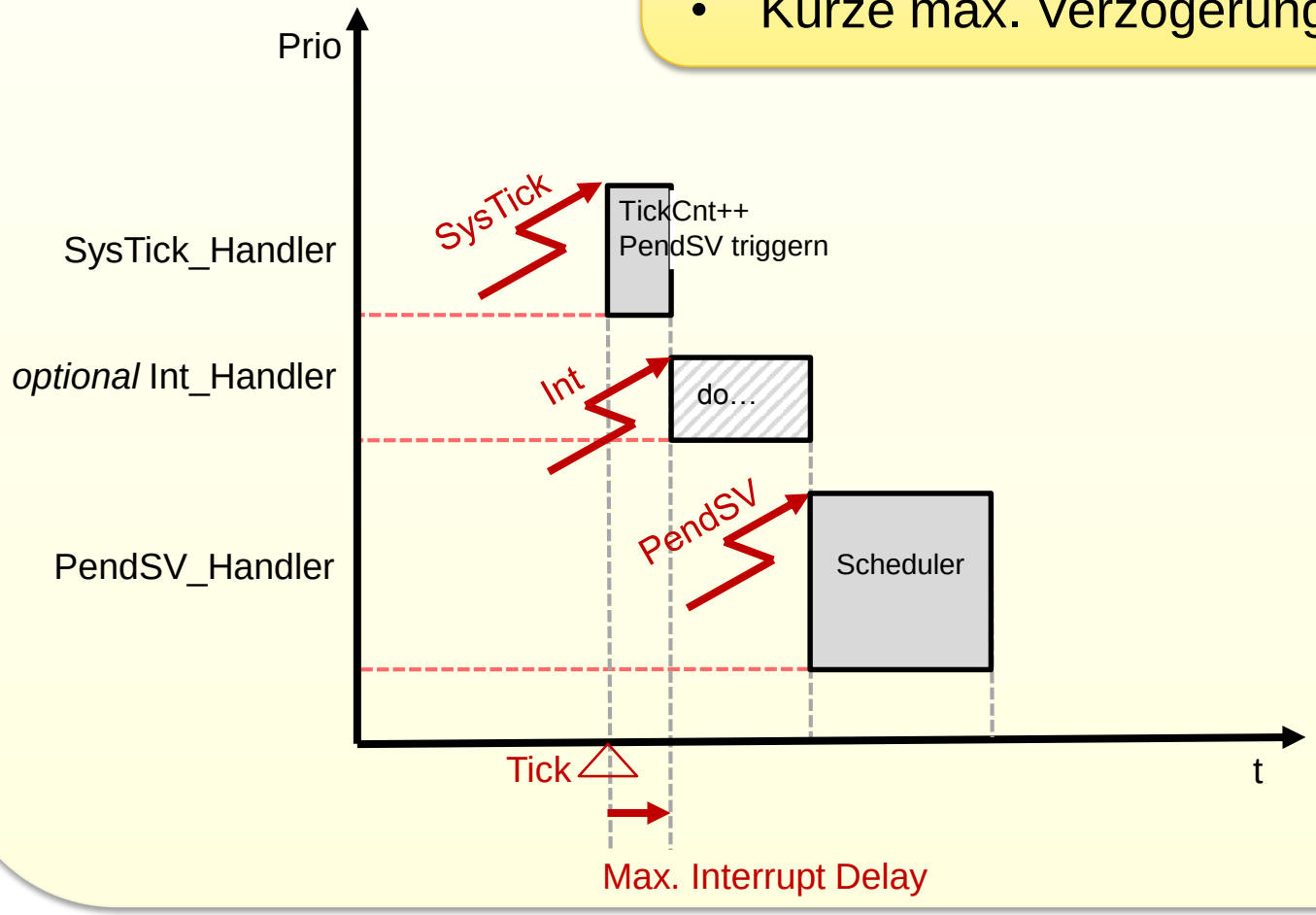
SysTick:

- Grundlegendes Timing des Systems
- Wiederkehrende Interruptfolge von z.B. 5ms
- TickCounter hochzählen
- Triggern des Schedulers zur Taskumschaltung



OS-Events

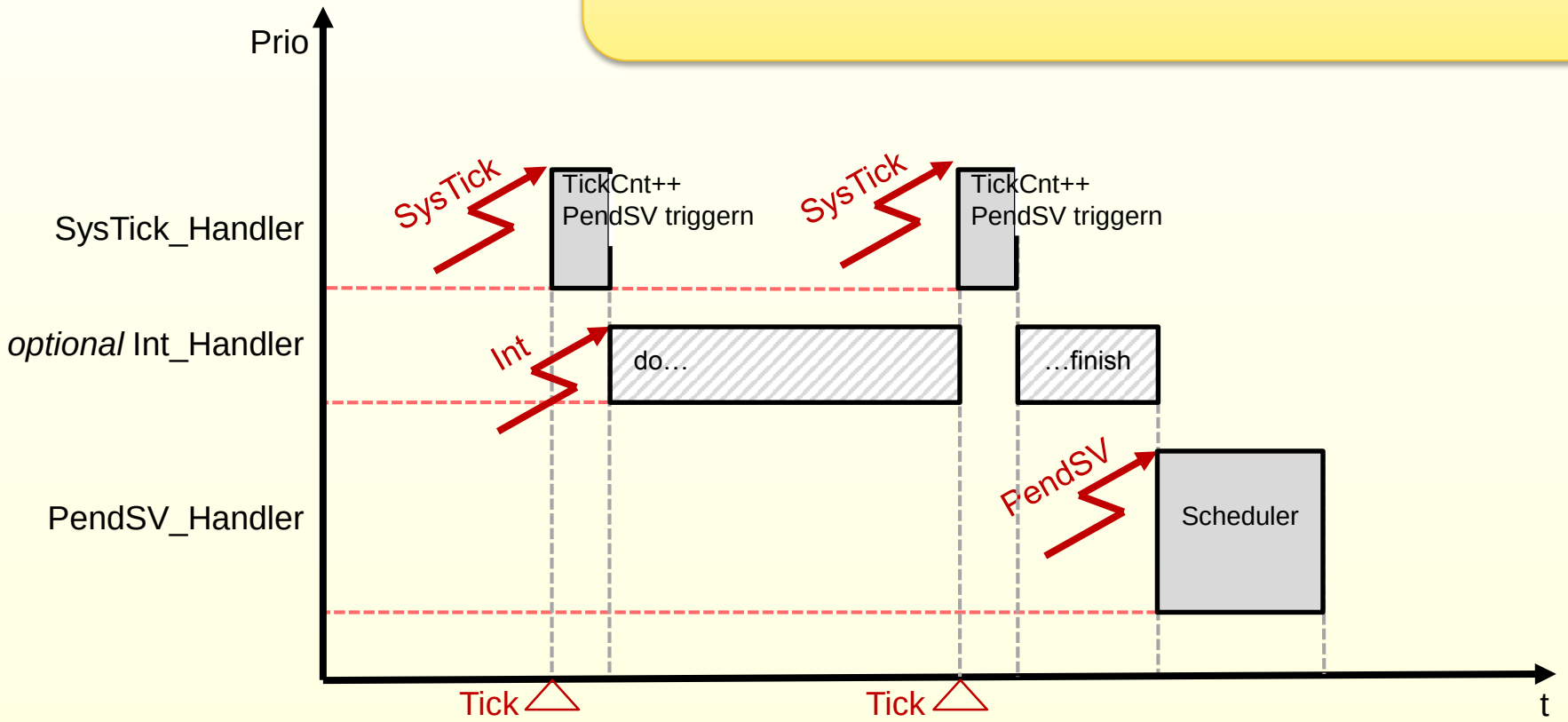
PendSV:
- Scheduling zur Taskumschaltung
- Getrennt vom Triggern im Systick
- Kurze max. Verzögerung für Interrupts



OS-Events

Lange Interrupts:

- Timing des Systems bleibt stabil (TickCnt)
- Scheduling verzögert

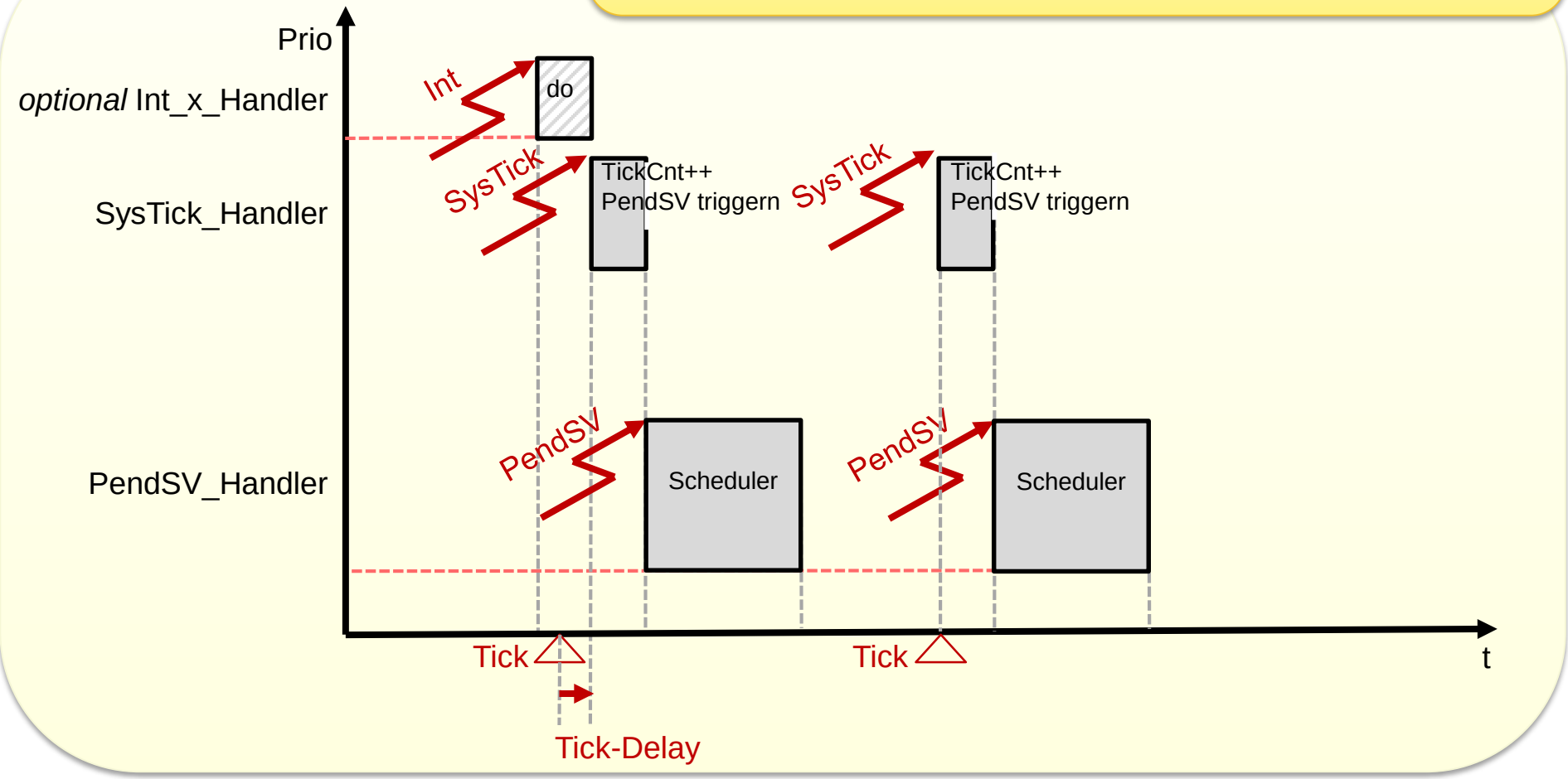


MICRO CONSULT

OS-Events

Hochpriore Interrupts:

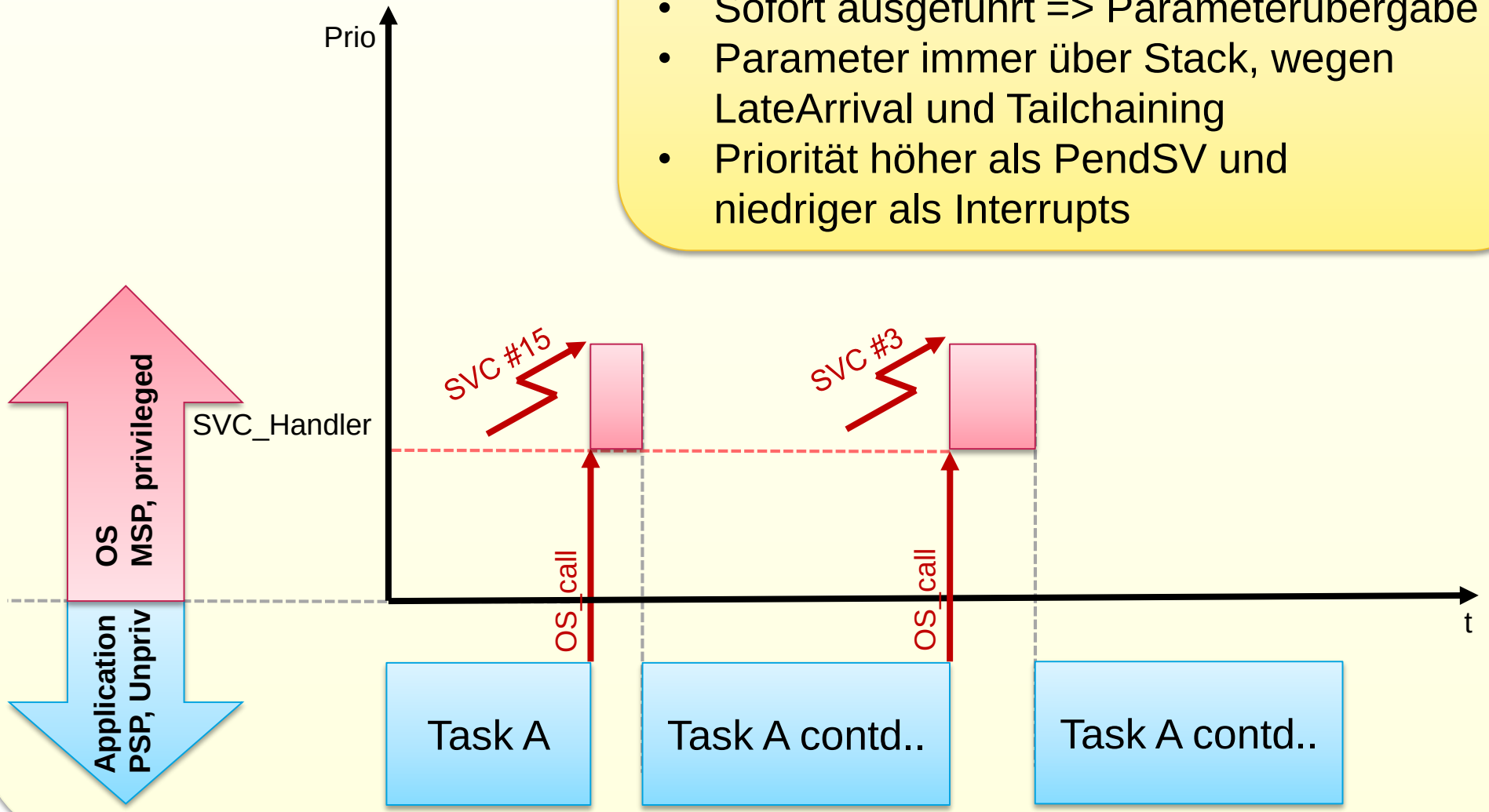
- Für Interrupts ohne SysTick-Delay
- Timing des Systems bleibt stabil (TickCnt)



OS-Events

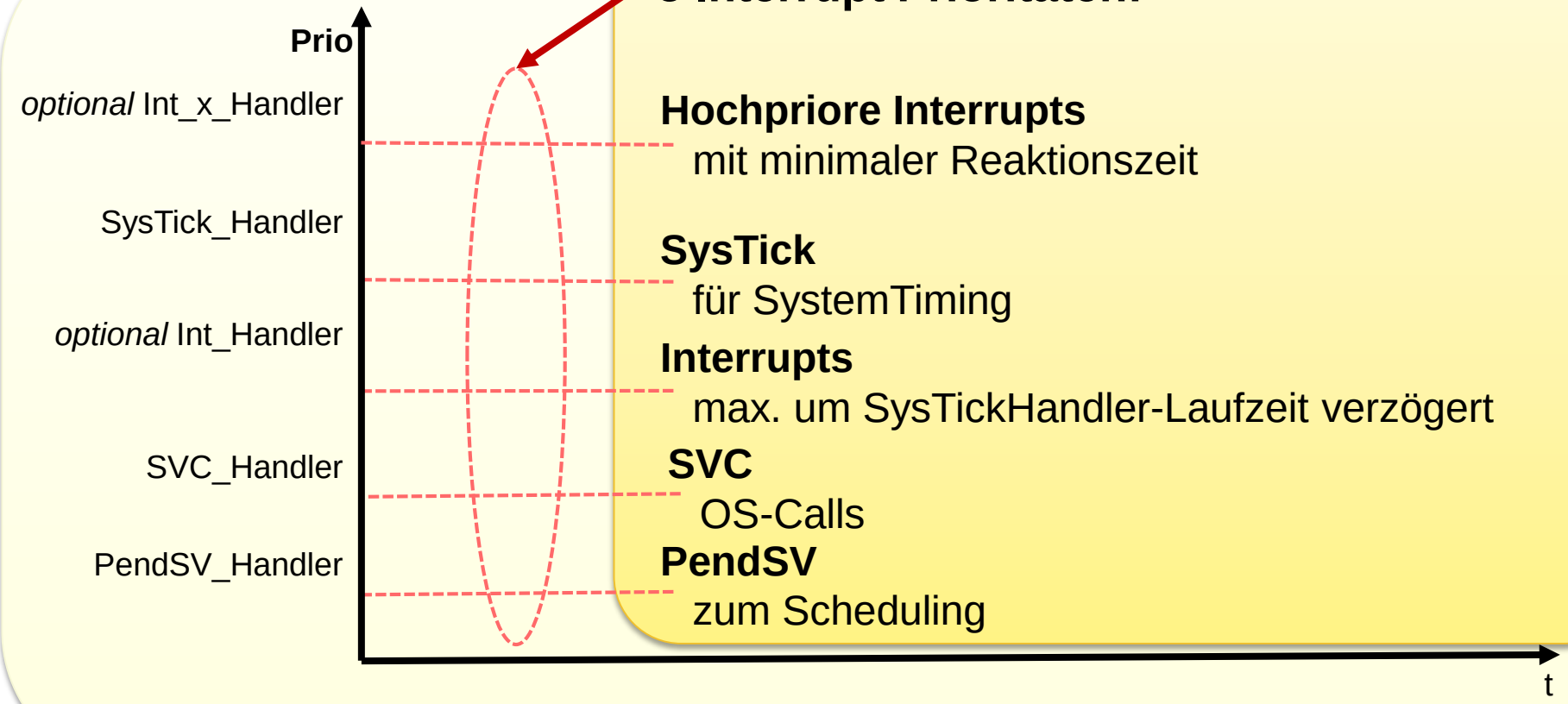
SVC:

- „Supervisor Call“, Betriebssystemaufrufe
- Extra Mschinenbefehl, SVC #Num
- Sofort ausgeführt => Parameterübergabe
- Parameter immer über Stack, wegen LateArrival und Tailchaining
- Priorität höher als PendSV und niedriger als Interrupts



MICRO CONSULT

OS-Events



5 Interrupt Prioritäten:

Hochpriore Interrupts
mit minimaler Reaktionszeit

SysTick
für SystemTiming

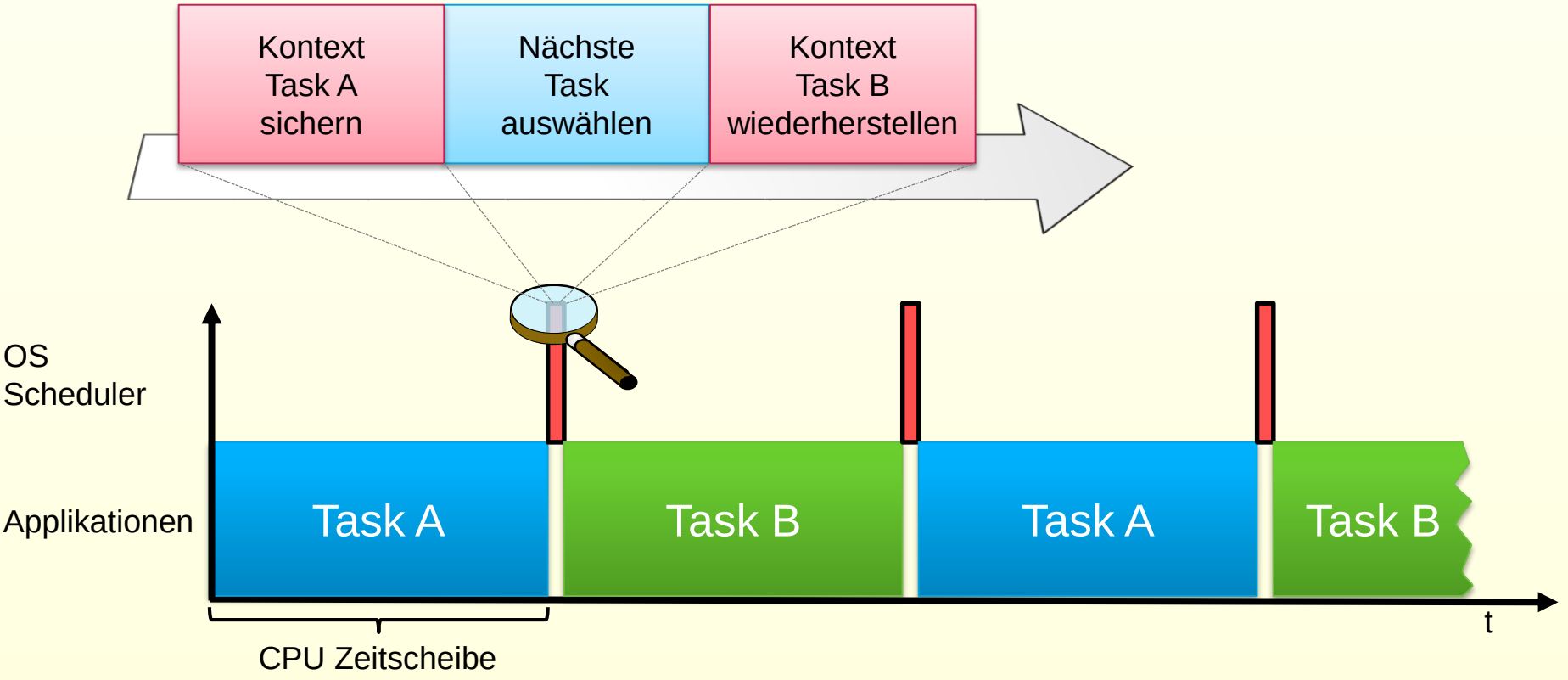
Interrupts
max. um SysTickHandler-Laufzeit verzögert

SVC
OS-Calls

PendSV
zum Scheduling

Der Scheduler

- Ist wesentlicher Bestandteil eines OS Kernels
- Teilt die Controller Ressourcen in Zeitscheiben den Tasks zu
  (hier pre-emptive, Round Robbin)
- Sichert den Kontext der Tasks, wenn sie unterbrochen werden
- Stellt den Kontext für die Tasks wieder her

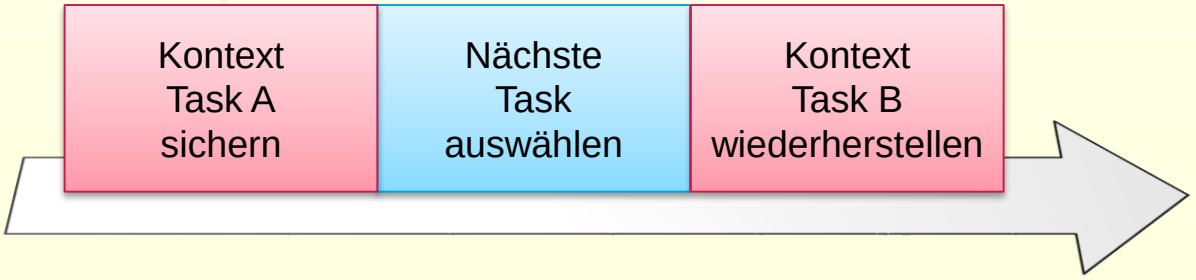


C oder Assembler?

- Fast alles ist in C möglich
- Im MyOS-Kontext sinnvolle ASM Bereiche:
  - Sichern und Wiederherstellen des Kontexts im Scheduler
  - Durch viele Inline Assembler Befehle und Intrinsic Funktionsaufrufe würde C hier schnell unhandlich werden.

ASM:
Wegen vieler direkter Register-Zugriffe und Stack-Manipulation

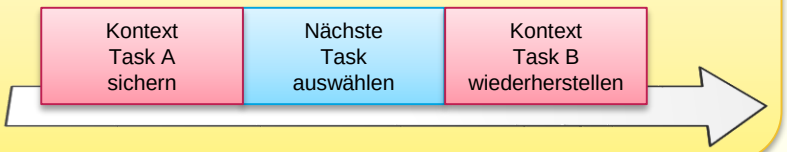
C:
Die Auswahl der nächsten Task kann vorteilhaft in C implementiert werden.



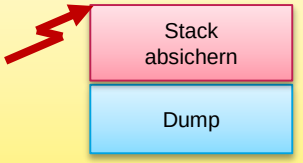
C oder Assembler?

Im MyOS-Kontext gibt es drei sinnvolle ASM Bereiche:

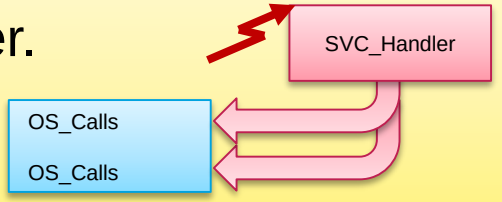
Sichern und Wiederherstellen des Kontexts im Scheduler in ASM.
Die Auswahl der nächsten Task kann
vorteilhaft in C implementiert werden.



Absicherung eines funktionierenden Stacks im Hardfault-Handler.
Das eventuelle Dumpen der Fehlersituation kann
mit dem abgesicherten Stack in C realisiert werden.

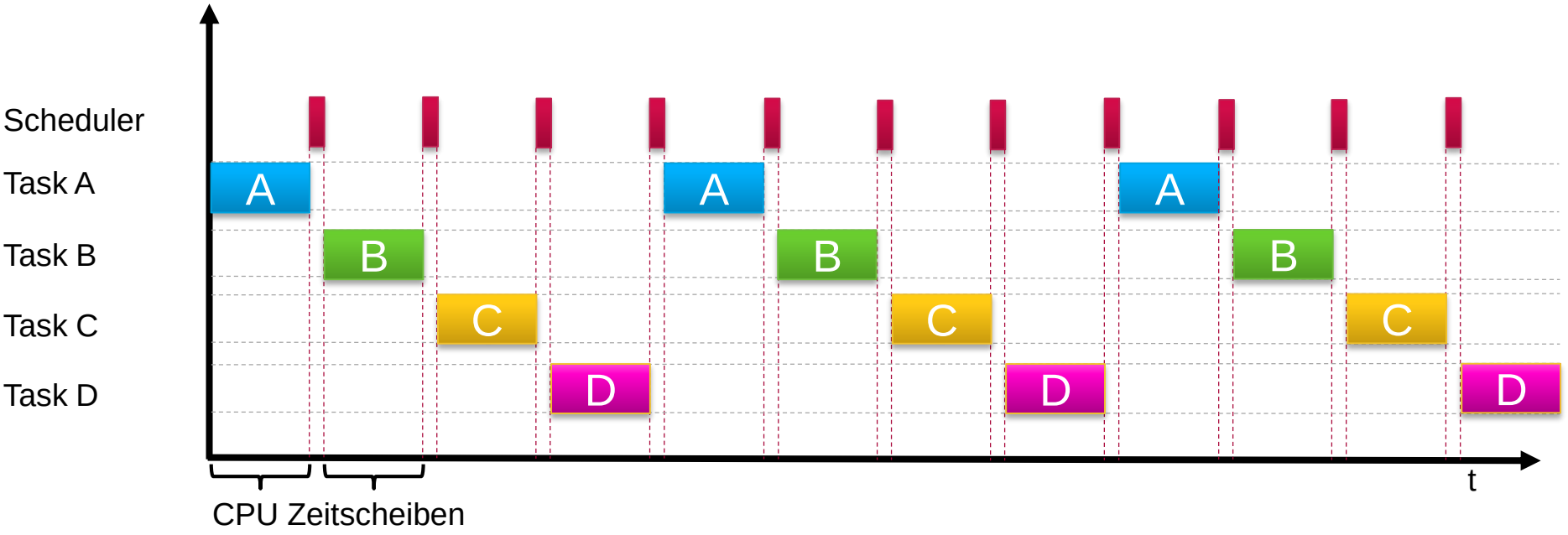


Dispatchen der unterschiedlichen Betriebssystemaufrufe und
Stackbehandlung der Parameter im SVC-Handler.
Die eigentlichen Betriebssystemroutinen
können in C geschrieben werden.



Einfachstes Round Robin

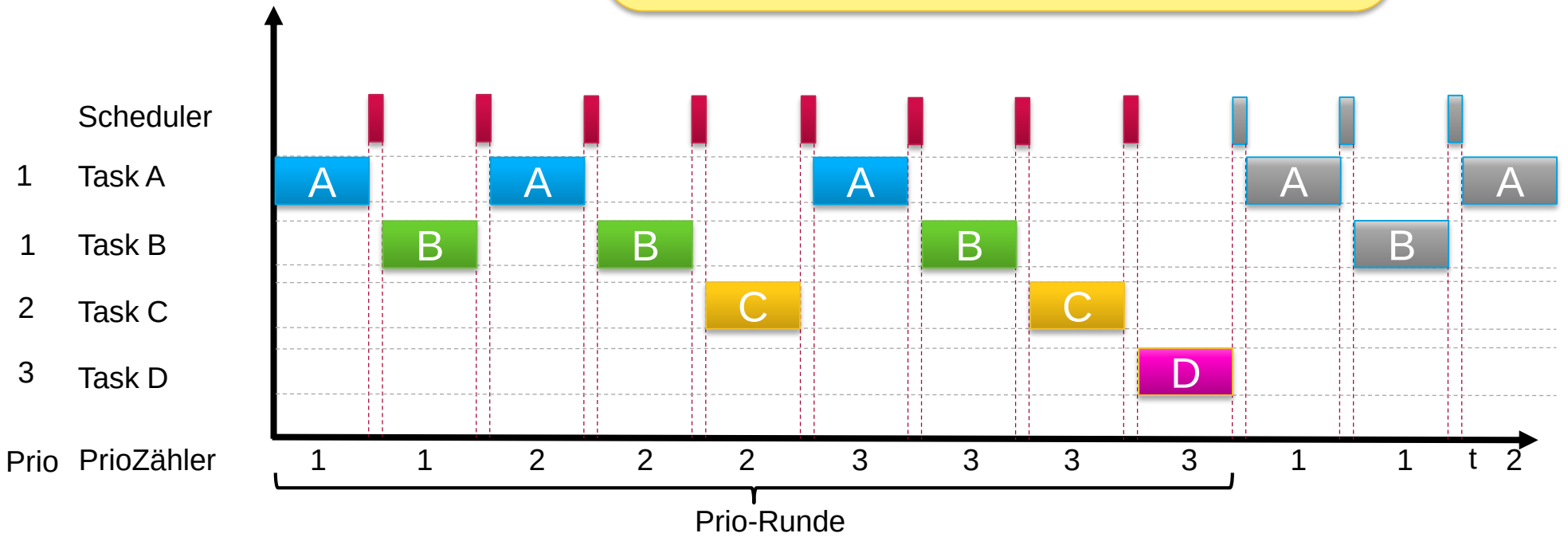
- Zuteilung gleich langer Zeitscheiben
- Jede Task kommt der Reihe nach dran



Round Robin mit Prioritäten

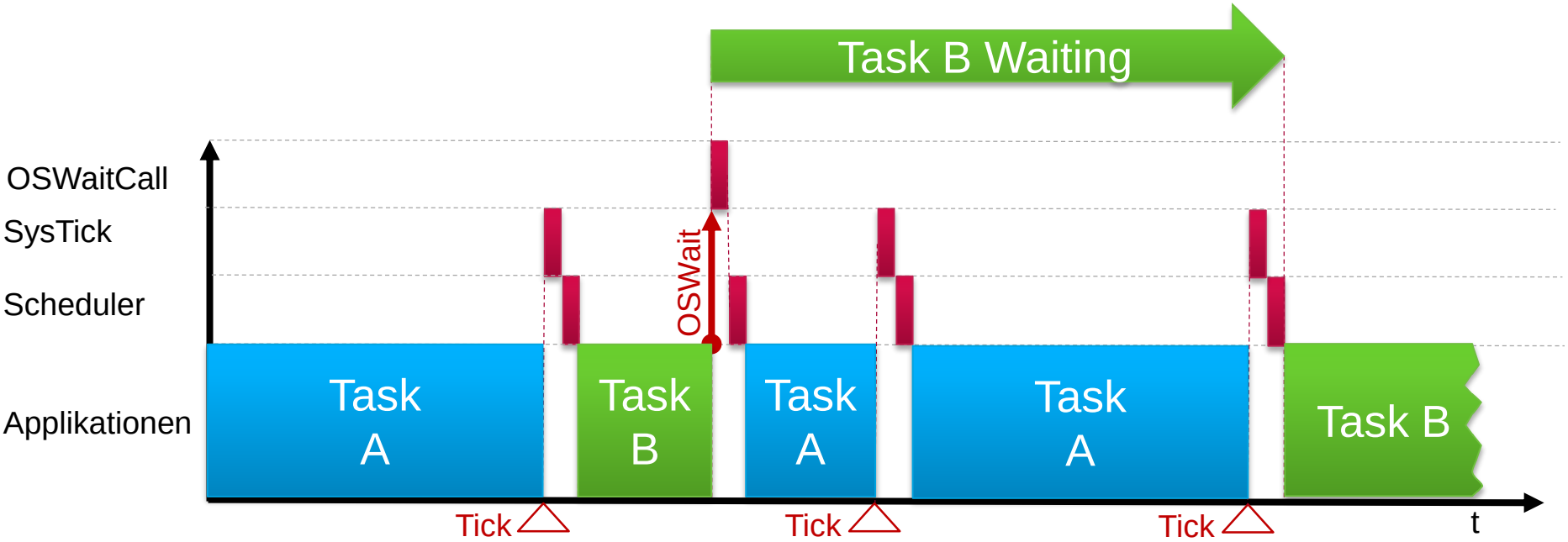
- Zuteilung gleich langer Zeitscheiben nach Priorität
- Inkrementelle Schleife bis zur höchsten Prio 0..1, 0..2, 0..3, …
- Innerhalb der Schleife bekommen diejenigen Tasks, deren Prio <= dem aktuellen Prio-Zähler ist eine Zeitscheibe zugewiesen

A Task A, **Prio 1**, 3 Zeitscheiben, 3/9, **~33%** CPU
B Task B, **Prio 1**, 3 Zeitscheiben, 3/9, **~33%** CPU
C Task C**, Prio 2**, 2 Zeitscheiben, 2/9, **~22%** CPU
D Task D, **Prio 3**, 1 Zeitscheiben, 1/9, **~11%** CPU



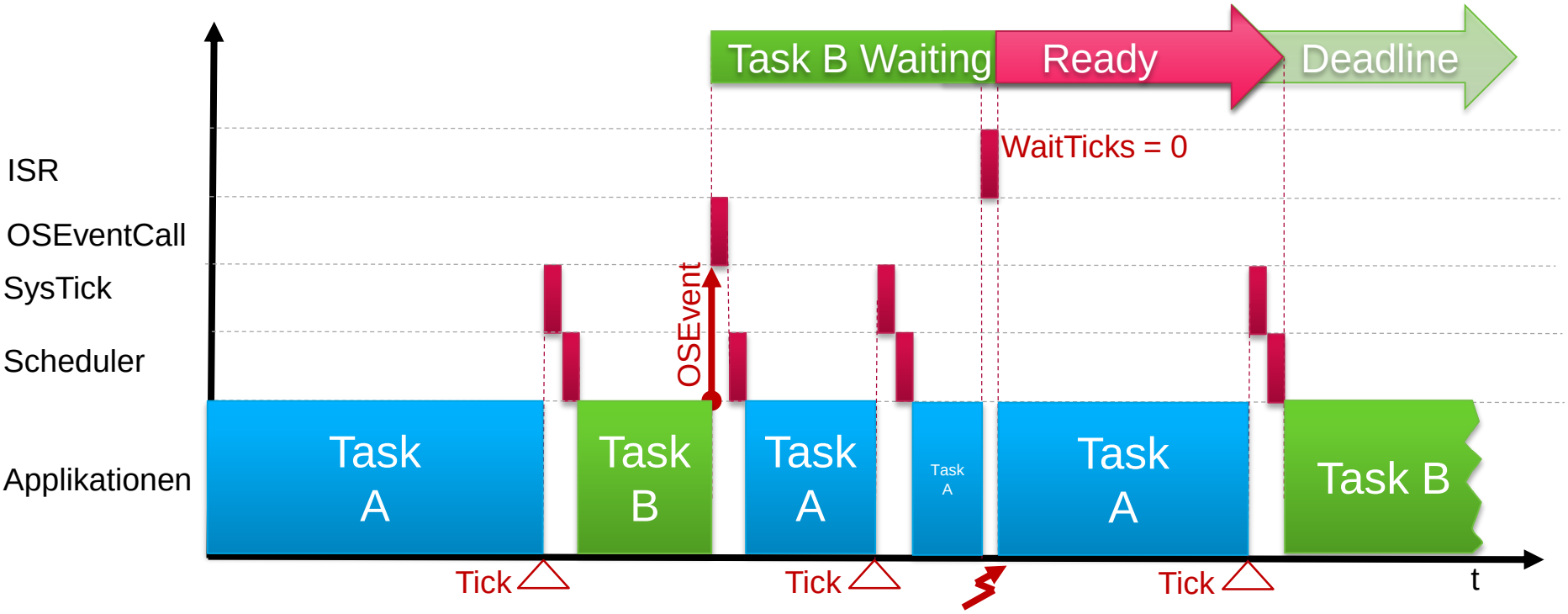
Round Robin mit Wait-States

- Aufruf aus der Task über SVC Call mit Wait-Ticks als Param.
- SVC triggert PendSV zum Rescheduling
- Scheduler teilt keine Zeitscheibe zu solange die Wait-Ticks nicht verstrichen sind
- Idle Task falls alle Tasks im Wait Zustand sind



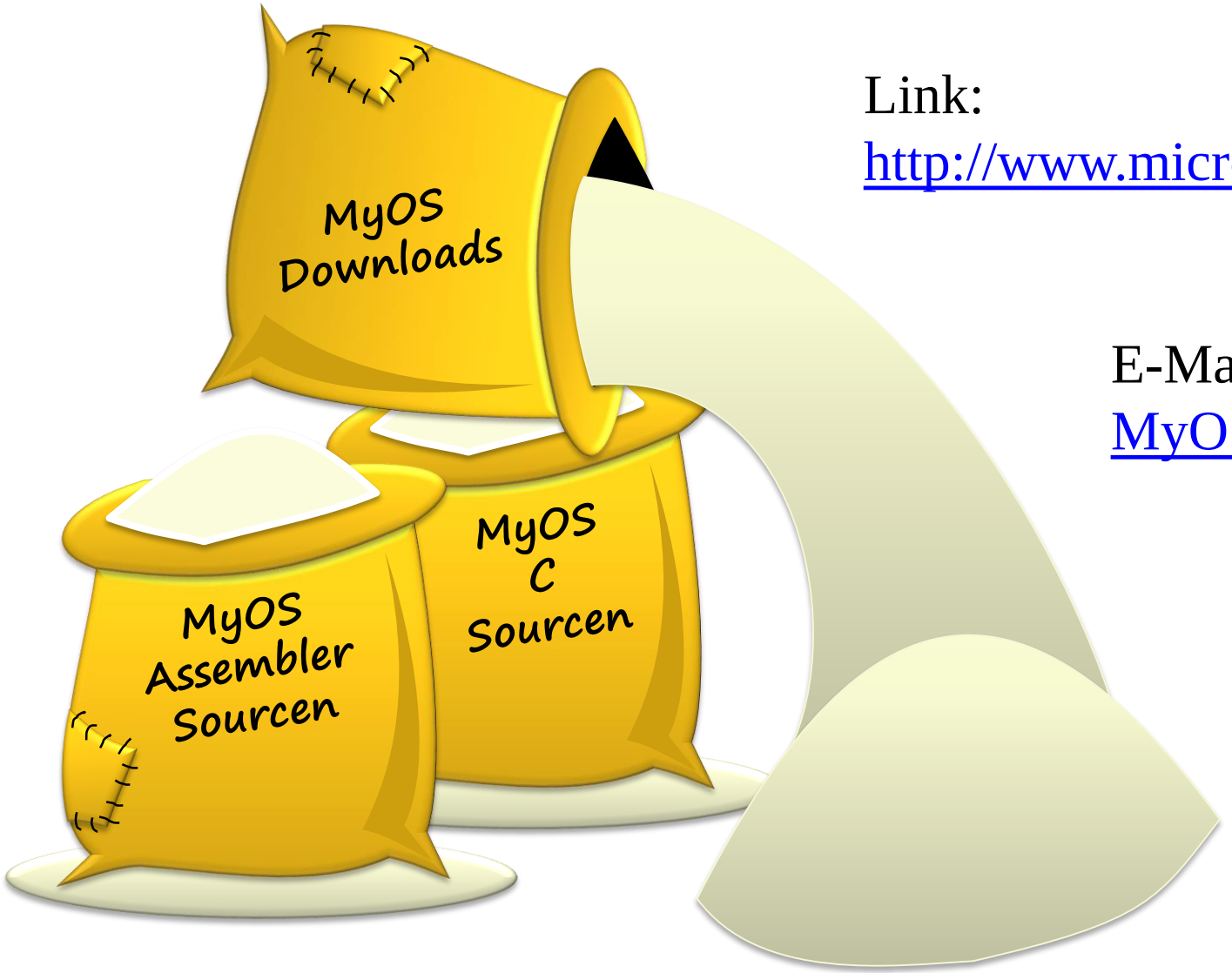
Warten auf Events mit Deadline

- Deadline setzten wie warten mit Wait-Ticks
- Die Interrupt Service Routine setzt Wait-Ticks auf 0 zurück
- Die Task ist damit „Ready“ für die nächste Zeitscheibe
- Sollte der Event nicht auftreten läuft die Deadline ab



MICRO CONSULT

MyOS Downloads



Link:
http://www.microconsult.de/MyOS

E-Mail (Update-Service):
MyOS@microconsult.de

Zusammenfassung

Wann setze ich ein eigenes MyOS ein?

Eigenes MyOS

- Immer dann wenn ich eine Loop verwenden würde
- und kein komplettes RTOS verwenden will
- Als Basis zur Portierung eines vorhandenen Betriebssystems z.B. von 8/16-bit auf Cortex-Mx

RTOS (Kommerziell oder Freeware)

- Komplexe Peripherieverwaltung
- ETH TCP/IP

Danke

www.microconsult.de Remo Markgraf, Tel. 089 450617-36, r.markgraf@microconsult.de