Kapitel 3: Interruptbehandlung

Why interrupts, anyway?

... I just remebered when we found out there was no interrupt facility built in to the 8563. I remember how patient the designer was when he sat me down to explain to me that you don't need an interrupt from the 8563 indicating that an operation is complete because you can check the status ANY TIME mearly by stopping what you're doing (over and over) and looking at the appropriate register, (even if this means banking in I/O) or better yet sit in a loop watching watching the register that indicates when an operation is done (what else could be going on in the system besides talking to the 8563 ???) Our running gag became not needing a ringer on the phone because you can pick it up ANY TIME and check to see if someone's on it, or better yet, sit at your desk all day picking the phone up. Even in the hottest discussions someone would suddenly stop, excuse himself, and pick up the nearest phone just to see if there was someone on it. This utterly failed to get the point across but provided hours of amusement. The owners at the local bar wondered what fixation the guys from Commodore had with the pay phone.

Bil Herd (typos sic), Commodore C128 HW design/develpoment team leader, 22-Jan-93

Problemkontext

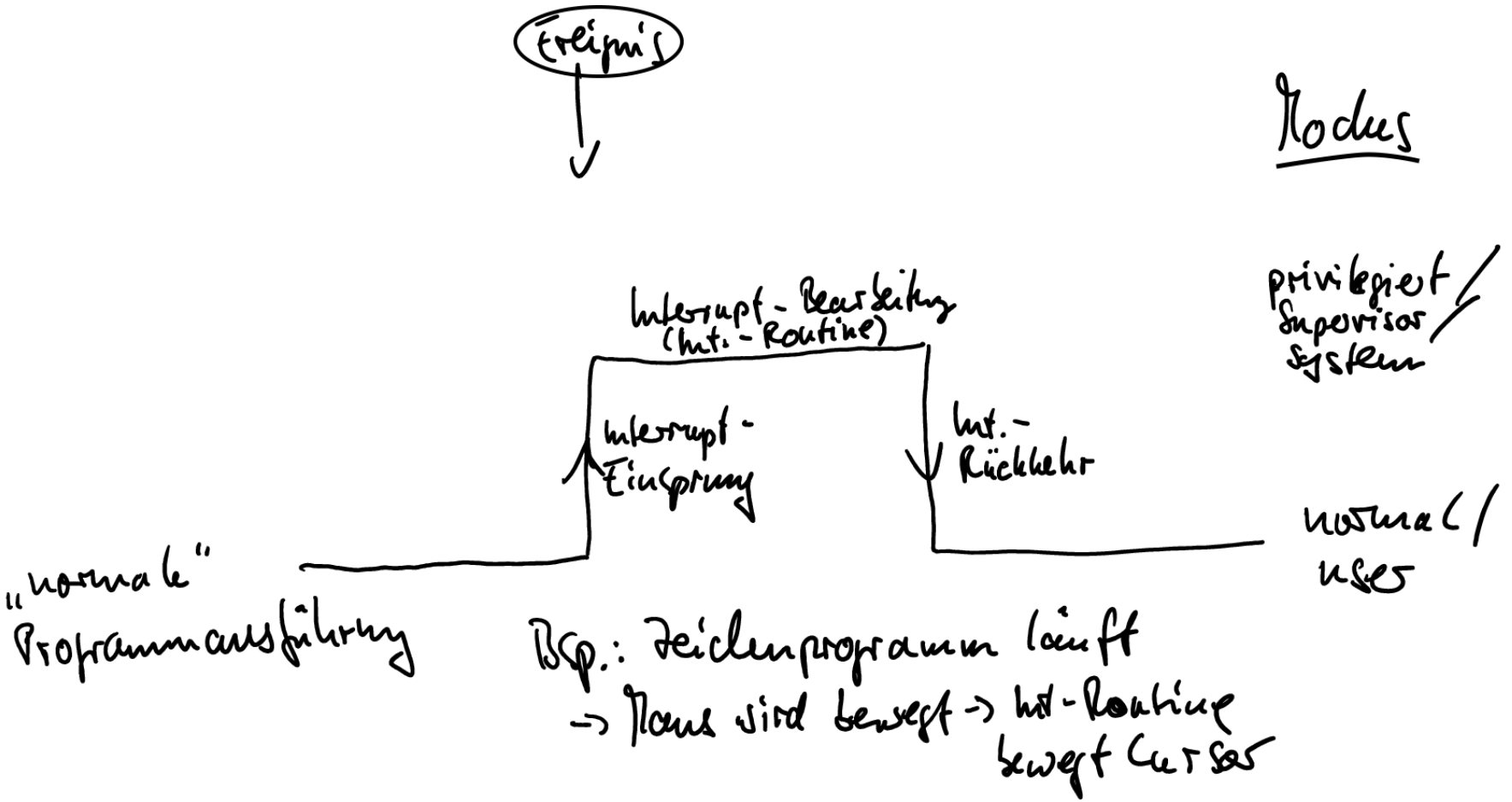
Analogie Telefon: ohne Klingel ⇒ Nutzer muss regelmäßig auf Anrufer prüfen

Warum problematisch? Auslöser (Anruf) asynchron zum Workflow des Nutzers

asynchron: { nicht zeitlich gekoppelt / jederzeit

⇒ Eintreffen eines Ereignisses ist nicht vorhersagbar aus Sicht des Nutzers

Prinzipieller Ablauf



Abgrenzung von Subroutinen

	Interrupt	Subroutine
Verursacher	(ext.) Ereignis asynchron	Anwendungscode synchron
Maßnahmen	Prozessorstatus sichern (Status, PC, ..) ggf. Register in Int.-Routine sichern Restaurierung d. Status durch spezielle Return-Instruktion	Parameter übergeben, ggf. Anwendungsregister sichern
Sprungziel	Über Sprungtabelle mit Ereignis- typ als Index (Vektortabelle)	Ziel wird mit Call-Instruktion angegeben

Kategorien von Auslösern

- Hardware-Interrupts
  - interne, Peripherie im µController, z.B. Timer, serielle Schnittstelle, fertiggestellte A/D-Wandlung
  - externe, signalisiert über Bus-Anschluss (z.B. USB) oder Pins, die Interrupts triggern können (dedizierte oder GPIO - general purpose I/O)

Kategorien von Auslösern (2)

- Fehler-Interrupts, je nach Hersteller: Exception, Trap, Fault, Error Condition/Interrupt...

  System entdeckt Fehlerzustand bei HW-Zugriff oder Instruktionsausführung

  (synchron)

- Software-Interrupts: Exception, Trap

  Absichtlich durch spezielle Instruktionen ausgelöst.

  z.B. ARM: SVC "Supervisor Call"

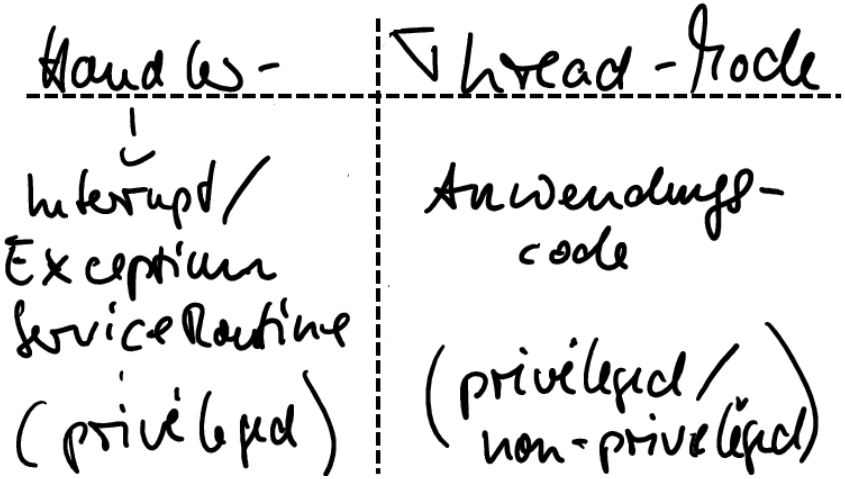
  (synchron)

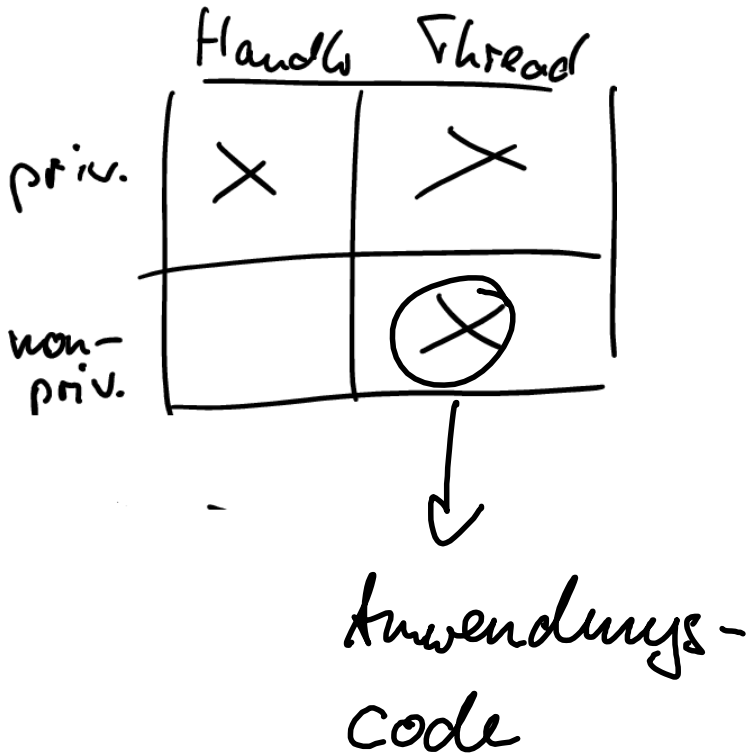
Abgrenzung SW-Interrupt / Subroutine

Abgrenzung Software-Interrupt / Subroutine

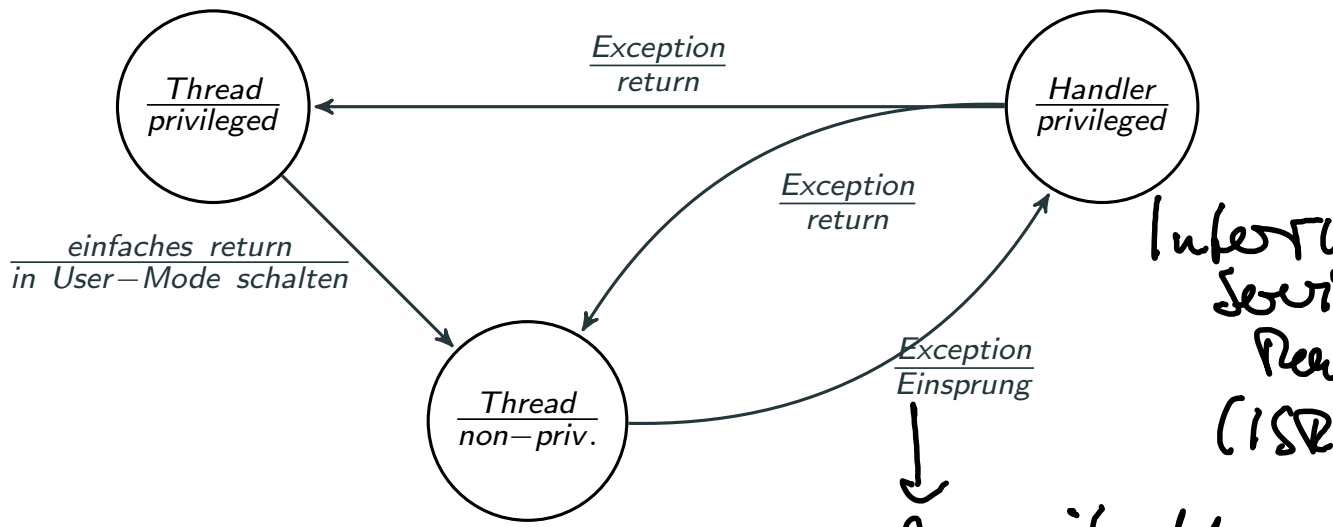
- ggf. autom. Sichern des Status bei SW-Int.
- Sprungziel bei SW-Int. nicht durch die Instruktion selbst angegeben ⇒ Sprung an sichere Adresse
  ↳ durch System in Vektortabelle vorgegeben
- Beim Sprung erfolgt Wechsel in höhere Privilegstufe für die Dauer der Ausführung der Interrupt-Routine

Modus und Privilegstufe: ARM Cortex M-3





Zustandsübergänge zwischen den Ausführungsmodi

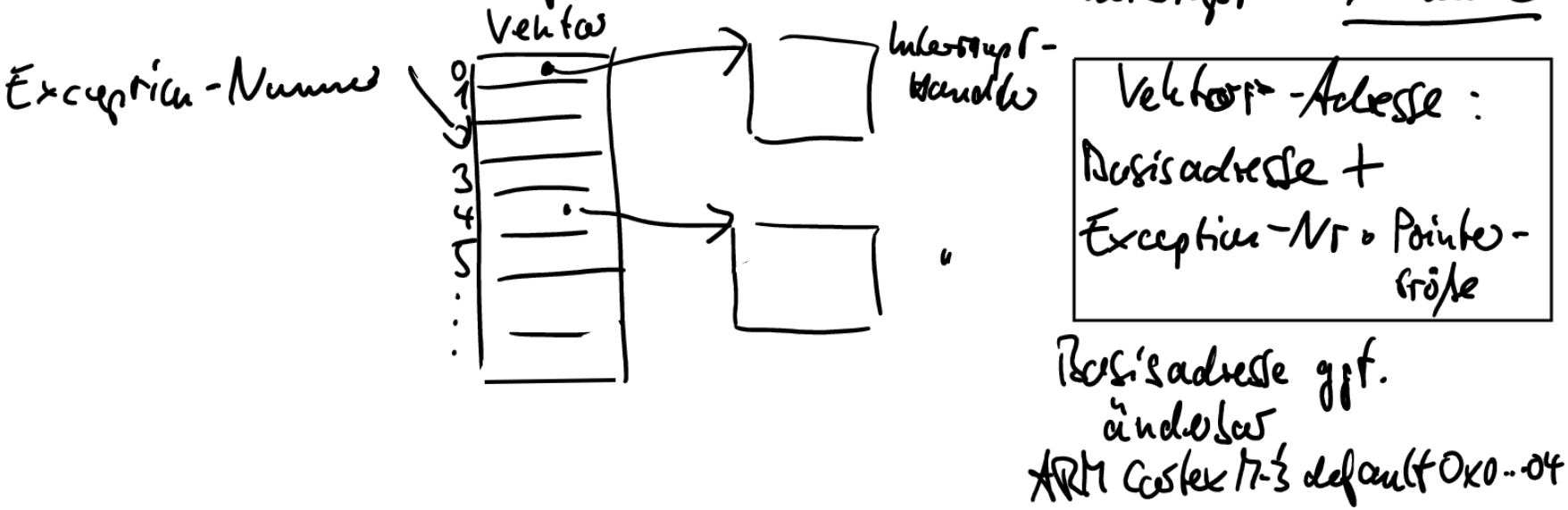


Interrupt-Vektoren

"Vektor": Array, hier meist: 1 Eintrag in Sprungtabelle

Mechanismus des Einsprungs in Interrupt-Handler:

Interrupt/Exception ausgelöst: Ereignis bestimmt Exception/Interrupt-Nummer



Beispiel ARM Cortex M-3

Vektortabelle:

```
void (* const pISRVectors[])(void) =
{
  /* stack pointer */
  (void (*)(void))((unsigned long)simpleMainStack +
sizeof(simpleMainStack)),
  /* reset handler */
  ResetHandler,
  /* NMI handler */
  NMIHandler,
  /* fault handlers: Hard, MPU, Bus, Usage */
  ISRDefaultHandler,
  ISRDefaultHandler,
  ISRDefaultHandler,
  ISRDefaultHandler,
  /* reserved values */
  0,
  0,
  0,
  0,
  /* System Service call handler */
  ISRDefaultHandler,
  /* Debug Monitor handler */
  ISRDefaultHandler,
  /* reserved */
  0,
  /* Pendable request for system service handler */
  ISRDefaultHandler,
  /* System tick timer handler */
  ISRDefaultHandler
};
```

initiale Stack-Pointer → an Adresse 0x0...0 abgelegt

Vektor-Eintrag 0: Reset-Handler → Adresse 0x0...4

1 Routine f. verschiedene Quellen: Dummy oder Dispatch
↳ muss zunächst Quelle bestimmen

nach Systick-Handler folgen Vektoren für Hardware-Interrupts (IRQs)
↳ Interrupt Request

Priorisierung

Abgrenzung von Vektor-Nummer:

Vektor-Nr -> Quelle / Ereignis

Priorität -> Wichtigkeit: Welcher Interrupt wird ausgelöst, wenn mehrere Ereignisse eintreffen
-> kann zur Verdrängung eines niederpriorisierten Interrupts führen

ARM Cortex-M3: 3 feste Prioritäten: -3 Reset, -2 NMI, -1 Hard Fault
kleiner Wert = hohe Priorität
Alle weiteren Vektoren besitzen programmierbare (wählbare) Priorität (max. 128)
Prioritäten können gruppiert werden
⇒ Gruppenpriorität bestimmt ggf. Verdrängung
Einzelpriorität bestimmt Reihenfolge in Gruppe

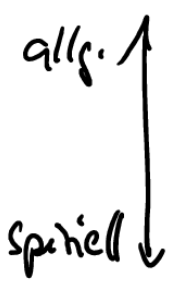
Maskierung

Maskierung ≈ Verhindern v. Interrupts (selektiv)

↳ Bitmasken in Steuerregistern, die einzelne Interrupts erlauben/verhindern

NMI = Non Maskable Interrupt, meist auch als Pin herausgeführt

Maskierung meist mehrschichtig



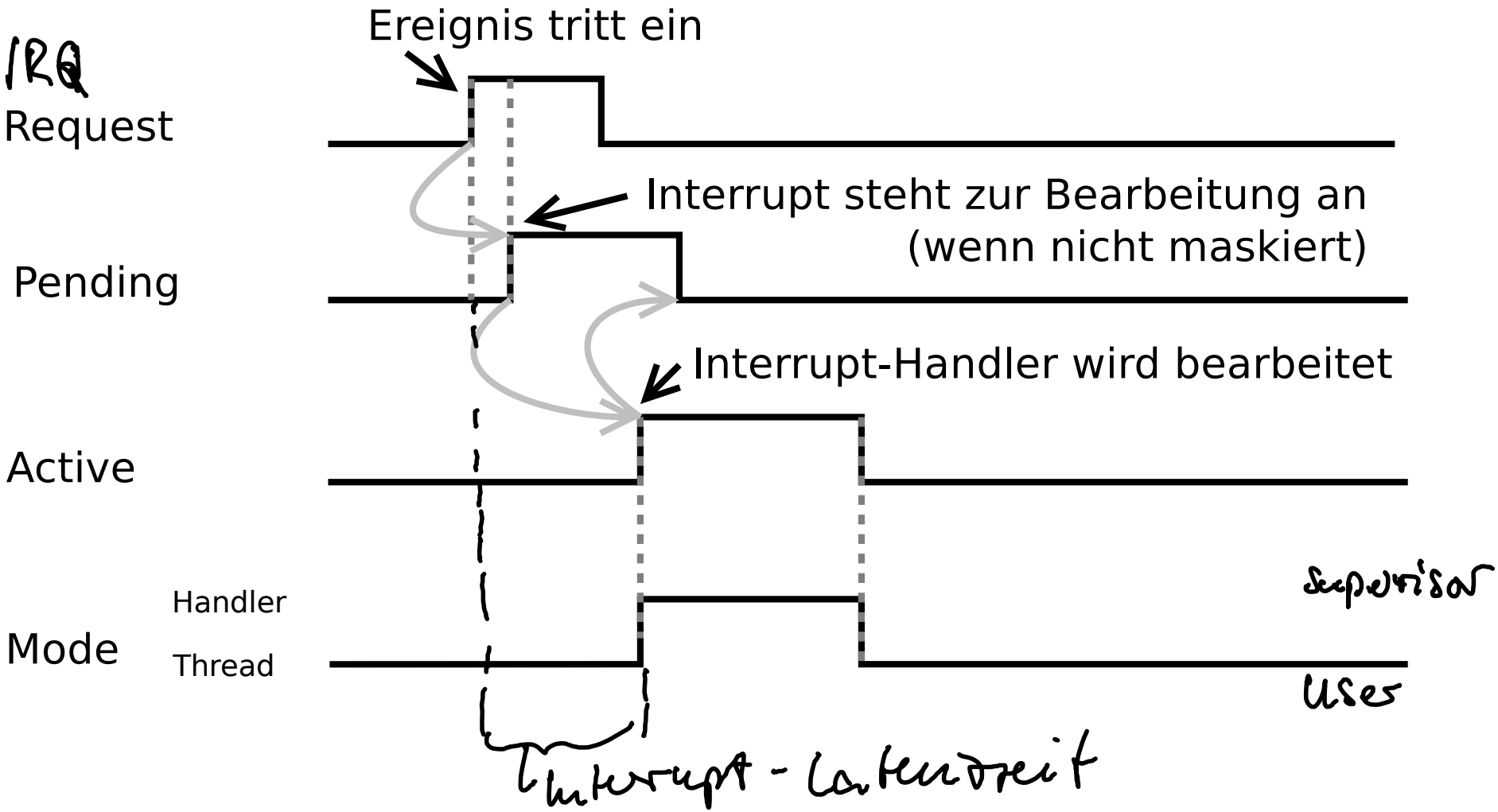
global

Gruppe (Priorität) → Nur Interrupts > Masken-Priorität

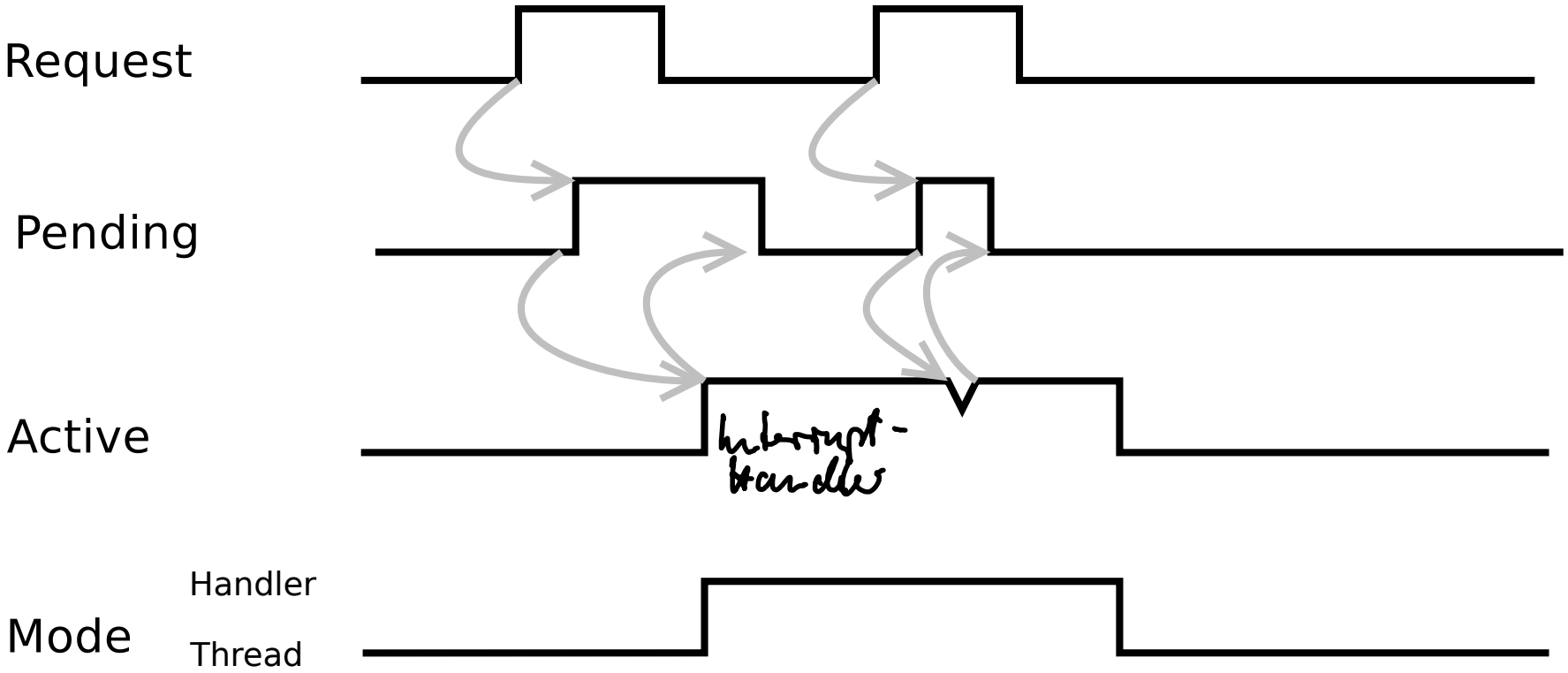
IRQ/Exception No.

Subfunktion (z.B. nur RX am UART)

Detaillierter Ablauf



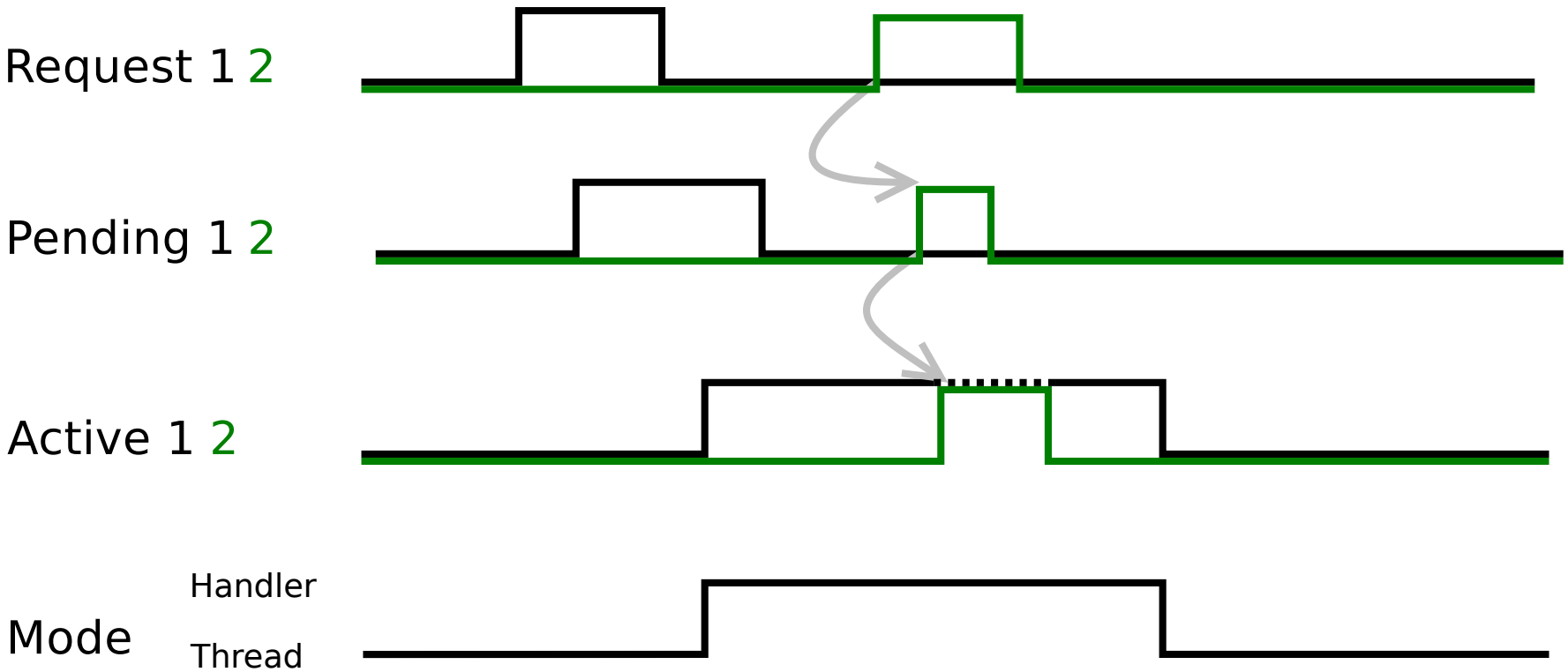
Ablauf mit erneutem Request derselben Interruptquelle



Keine vollständige Rückkehr in den User Mode, auch nachdem der 1. Handler bereits beendet wurde $\Rightarrow$ Register müssen nicht erneut restauriert / gesichert werden

Ablauf mit zweitem, höherpriorem Request

Request 2 hat höhere Priorität



- Interrupt höherer Priorität wird “pending” $\Rightarrow$ wichtiger als aktuell bearbeiteter Interrupt
- Handler 2 verdrängt vorübergehend Handler 1 vom Prozessor

Retten der Register

Beim Eintsprung in den Handler werden einige Register auf den Stack gesichert ("Stacking")

bei Rückkehr restauriert ("Unstacking")

Bei ARM Cortex-M3: PSR
PC
LR
R12
R3
R2
R1
R0

kein Unstacking bei verschachtelten Handler-Aufrufen (nesting / Verdrängung)
⇒ Handler sollten trotzdem alle verwendeten Register sichern/restaurieren

ARM-Faults

Fehler-Exceptions, Kategorien bei Cortex-M3:

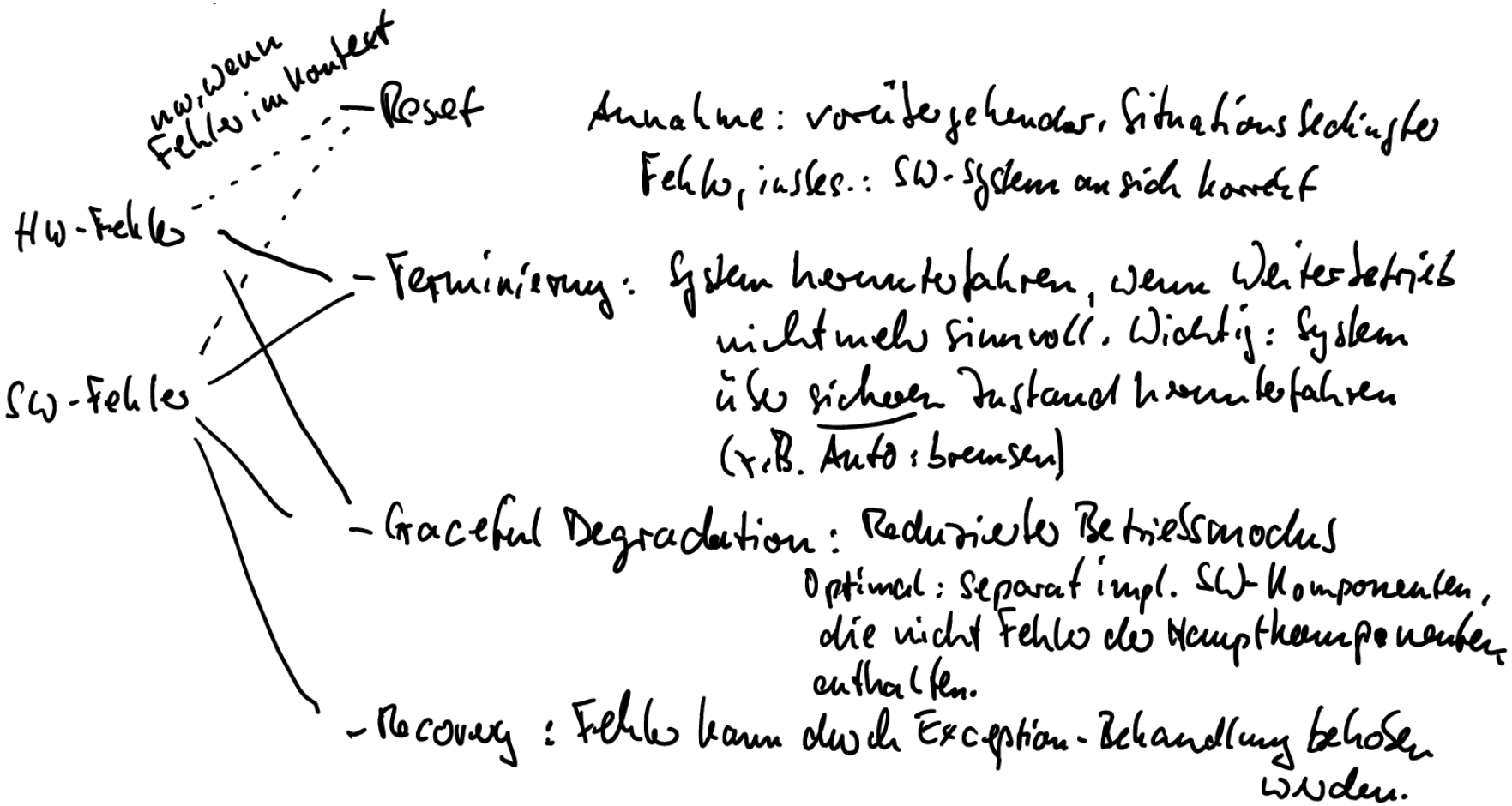
Bus Fault z.B.
- kein Speicher an angesprochener Adresse vorhanden
- Zugriff mit falscher Größe für Speichertyp (z.B. 32 Bit statt 16 Bit)

Memory Management Fault, z.B.
- Zugriffsverletzung (Speicherschutz)

Usage Fault, z.B.
- Division durch 0, illegale Instruktion

Hard Fault
- i.W. aus anderen Faults eskalierte Fehler, da keine Fortsetzung des Ablaufs mehr erlaubt.

Fehlerbehandlung

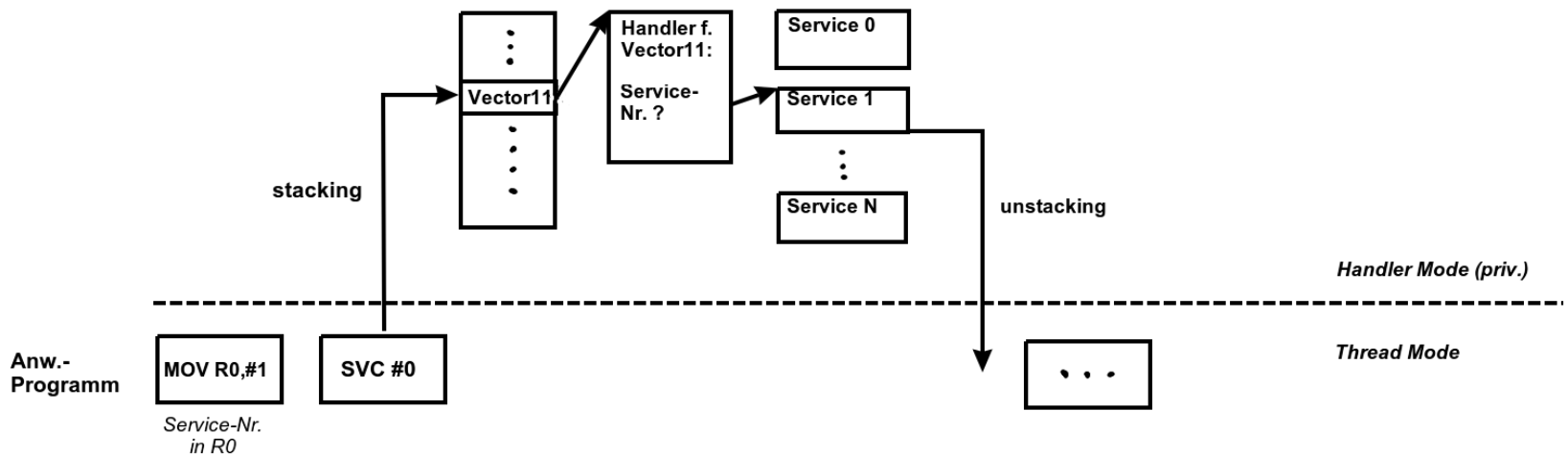


Service Calls

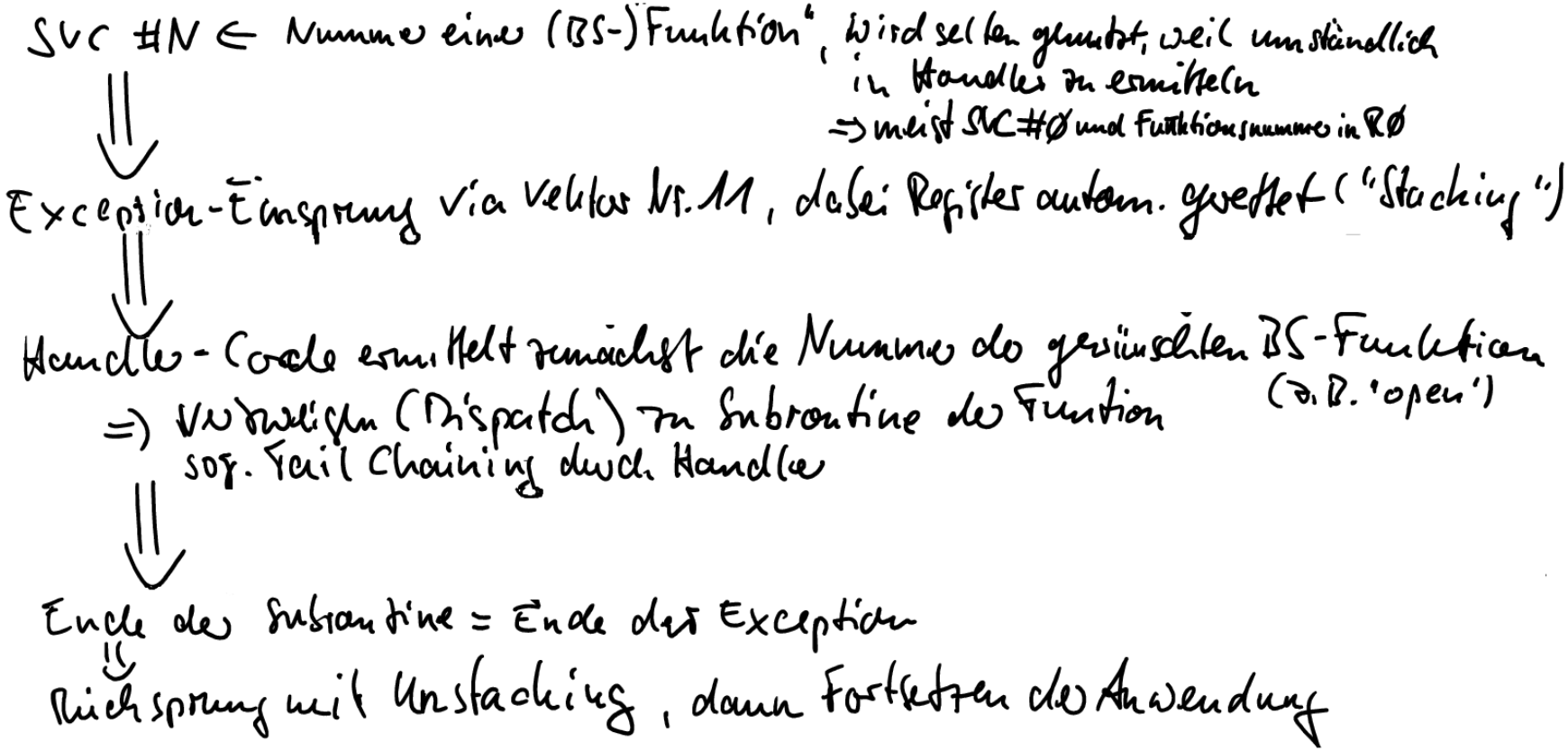
über SW-Interrupt realisierte Betriebssystem-Funktionen (meist)

Cortex-M3: Operation SVC (Supervisor Call)
Aufruf von SVC bewirkt
- Sprung an feste Adresse via Tabelle
- Umschalten in priv. Modus
- Retten v. Register

} sichere Ausführungsumgebung



Service Calls: Ablauf



Betriebssystem-Unterstützung

Hier betrachtet: Grundlagen/Komponenten, die später zu Konstruktion v. Betriebssystemen dienen

- Service Calls (ARM: SVC) → Betriebssystem-API + geschützte Ausführung (statisch)
- SysTick(Handler) → Regelmäßiges Aufrufen von Verwaltungsfunktionen des BS (z.B. Task-Wechsel) (dynamisch)
- User/Supervisor Mode (ARM: Thread/Handler) → Schutz durch Einschränkung der Möglichkeiten von Anwendungscode → BS und damit System wird geschützt
- Speicherschutz auf HW-Ebene: MMU/MPU (= Memory Management/Protection Unit) → Schutz des BS vor Speicherzugriff durch Anwendung und der Anwendungsprozesse voreinander

Debugging

Aktive Unterstützung von Debugging durch die Systemarchitektur

— Debug Handler : Exception, die u.a. bei Breakpoints ausgelöst wird

Auslösen v. Breakpoints :

- über Debug-Schnittstelle (z.B. JTAG)
- durch BKPT-Instruktion im Code
  (ARM, ähnl. für andere Architekturen)
- alternativ über IRQ, NMI
  falls keine andere Möglichkeit