Gliederung der Vorlesung

1. Einleitung
2. Power Management
3. Parallele I/O
4. Serielle Schnittstelle
   - I^2C
   - USART
5. Timer
6. Interrupt Handling
7. Signalverarbeitung
8. Regelungen
9. Software Interrupt
10. Leitungscodes
11. Weiterführende Themen

Ereignisgesteuerte Programmierung

- Ereignisgesteuerte Programmierung liefert eine eigene wichtige Klasse von Programmen
- Wichtigstes Kennzeichen ist, dass die Steuerung des Programm-ablaufs durch **externe Ereignisse** und nicht nur durch die Daten des Programms selbst erfolgt
- Ereignisgesteuerte Programme sind immer dann nötig, wenn der Computer intensiv mit seiner Umwelt in Interaktion tritt
- Beispiele:
  - grafische Oberflächen
  - Kommunikation
  - Prozesssteuerung

Interrupt Handling - Ereignisse

Wichtige externe Ereignisquellen sind;

- Tastatur
- Maus
- serielle Schnittstellen
  - asynchrone Schnittstellen, Ethernet, USB, Firewire
  - I2C, SPI, CAN
- Prozessanbindungen
  - parallele Schnittstellen
  - AD- und DA-Schnittstellen
- Timer
  - Periodische Interrupts
  - Ereignisse wie abgeschlossene Messungen
  - Real Time Clock
  - Watchdog

Interrupt Handling - Mechanismen

Ereignisverarbeitung kann implementiert sein als:

- Busy Waiting
  - Die einfachste Form der Ereignisabfrage
  - Endlosschleife bis Ereignis eintritt
- Polling
  - Zeitgesteuerte Abfrage des Ereignisses. Einfache Form der Parallelverarbeitung, wenn alle Ereignisquellen nacheinander abgefragt werden.
  - Problematisch wenn das „Message Arrival Pattern“ nicht bekannt ist
- Interrupt
  - „Echte“ Parallelverarbeitung (Nebenläufigkeit) der Ereignisse
  - Gefahren durch Multithreading, wenn kritische Bereiche nicht geschützt sind
  - Gefahr bei zu langen Interruptroutinen oder zu vielen Interrupts

Interrupt Handling - Mechanismen

Vergleich am Beispiel des Wartens auf einen Gast, wobei eine Klingel (Interrupt) vorhanden ist oder nicht:

- Busy Waiting (ohne Klingel)

  Man wartet an der Tür, bis jemand kommt. Dabei kommt man aber zu sonst nichts und ist immer beschäftigt. Der Gast wird dafür aber direkt empfangen, wenn er ankommt.

- Polling (ohne Klingel)

  Man läuft in festen oder variablen zeitlichen Abständen immer wieder an die Tür und prüft, ob jemand da ist. Man kann dazwischen etwas anderes (zu Ende) bearbeiten, der Gast wartet aber (vielleicht unzumutbar) lange auf Einlass.

- Interrupt (mit Klingel)

  Während man seine Aufgaben erledigt, kann man jederzeit durch die Klingel unterbrochen werden, um die Tür zu öffnen (Interrupt). Wie lange es dauert, bis man die Tür öffnet, kommt darauf an,

  - Interrupt Response Time: wie weit ist man von der Tür gerade entfernt
  - Kontextwechsel: was muss man alles tun, um an die Tür gehen zu können (Buch aus der Hand legen, Lesezeichen reinlegen, Brille absetzen...)
  - Priorität: Klingelt es normal (einmal), dann macht man alles in Ruhe und geordnet (Interrupt Request). Klingelt es Sturm, rennt man auf dem kürzesten Weg zur Tür (hoch priorisierter oder spezialisierter „Fast“ Interrupt Request)

Interrupt Handling - Polling

- Synchrone Ereignisbehandlung durch **Event Loop**
  - Programme fragt alle Schnittstellen in einer Schleife ab und reagiert auf auftretende Ereignisse
  - Randbedingung: Reaktionsroutinen dürfen selbst nicht warten, sondern müssen sofort zurückkehren
  - Problem: Die Wahrscheinlichkeit Ereignisse zu verpassen ist sehr hoch
- Variante:
  - Programm wird durch Timer getriggert und fragt nur in definierten Zeitabständen die Ereignisse ab.
  - Vorteil: Die verbleibende Zeit kann für andere Programme genutzt werden

Interrupt Handling – Beispiel Polling

```
// Pollingbeispiel (round robin)

void main(void)
   {
   for(;;)   // Endlosschleife
     {
     if( ereignis1)
           job1();
     if( ereignis2)
           job2();
     if( ereignis3)
           job3();
     }
   }
```

Interrupt Handling – Polling mit Priorität

```
// Pollingbeispiel (prioritätsgesteuert)

void main(void)
   {
   for(;;) // Endlosschleife
     {
     if( ereignis1)
           { job1(); continue; }
     if( ereignis2)
           { job2(); continue; }
     if( ereignis3)
           { job3(); continue; }
     }
   }
```

Interrupt Handling – Asynchrone Ereignisbehandlung

Interrupts ermöglichen die asynchrone Behandlung von Ereignissen (asynchron in Bezug auf den Befehlsstrom des laufenden Programms). Neben der Hardwareunterstützung für Interrupts erfordert dies ein spezielles asynchrones Programmierkonzept, das aufwändiger ist als ein synchrones (vergleiche Polling).

Anwendungsfälle sind in vielen Bereichen geläufig :

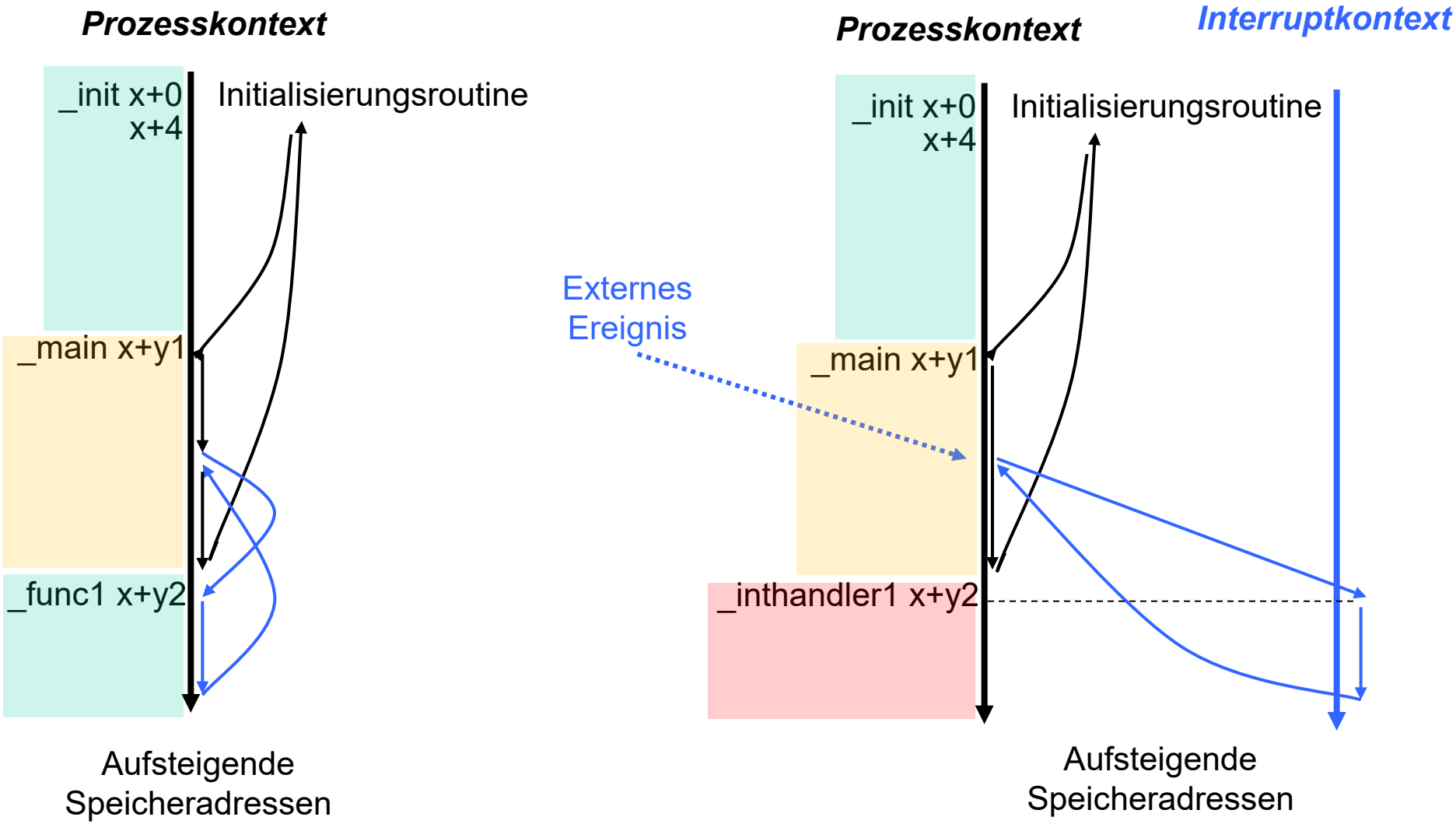
„Als Beispiel für eine Anwendung von Interrupts kann man sich einen Prozessor vorstellen, der, nachdem er einer Hardwarekomponente einen Auftrag gegeben hat, nicht aktiv auf deren Antwort wartet (Polling), sondern so lange andere Aufgaben erledigt, bis ihn jene Hardwarekomponente von sich aus durch einen Interrupt wieder auf sich aufmerksam macht. Ohne Interrupts wären beispielsweise präemptive (=Verdrängen von laufenden Programmen) Multitasking-Betriebssysteme unmöglich, da Programme ohne sie nicht mehr unterbrochen, vom Betriebssystem umgeschaltet (Timesharing) und Ein-/Ausgabegeräte nicht mehr bedient werden könnten.“ (Wikipedia)

Interrupt Handling - Interrupt

- Asynchrone Ereignisbehandlung durch Interrupts
- laufendes Programm wird unterbrochen und eine Reaktionsroutine (Interrupt Service Routine ISR) für das Ereignis gestartet
- Der **Kontext** der laufenden Programms (Inhalt von Registern) muss gerettet werden (vergleiche APCS Funktionsaufruf)
- Der Interrupthandler benötigt einen eigenen Kontext, der unabhängig von dem ursprünglich laufenden Programm ist, um eine **Nebenläufigkeit** zu erreichen also keine Beeinflussung des Zustands außerhalb der Handlers
- Interrupts besitzen eine **Priorität**, um bei Unterbrechungen durch mehrere Quellen für diejenigen mit hoher Priorität Reaktionszeiten garantieren zu können

Interrupt Handling - Kontextwechsel

Funktionsaufrufe durch Sprünge als Ablaufsteuerung

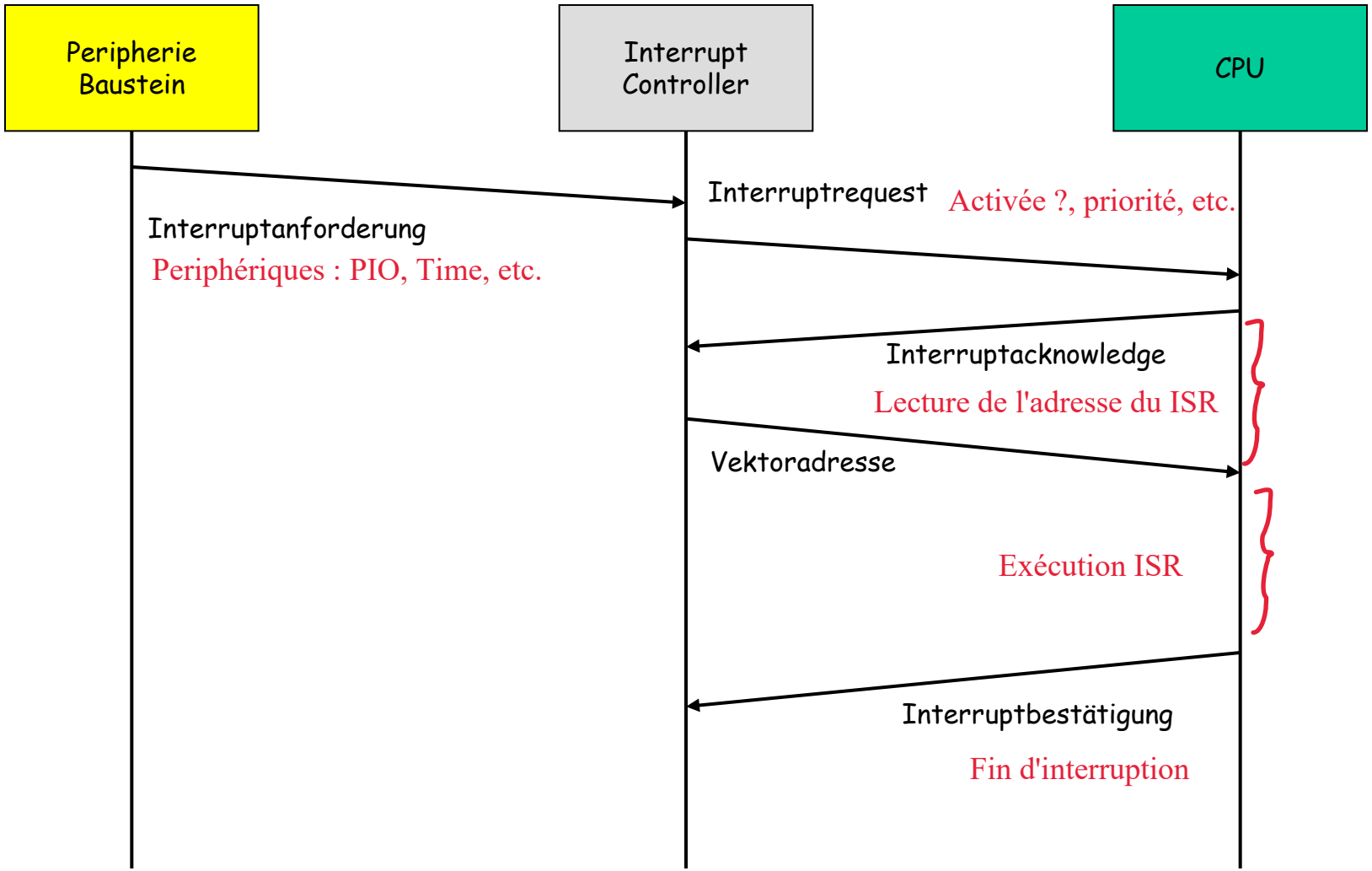


Interrupt Handling – Interrupt Controller

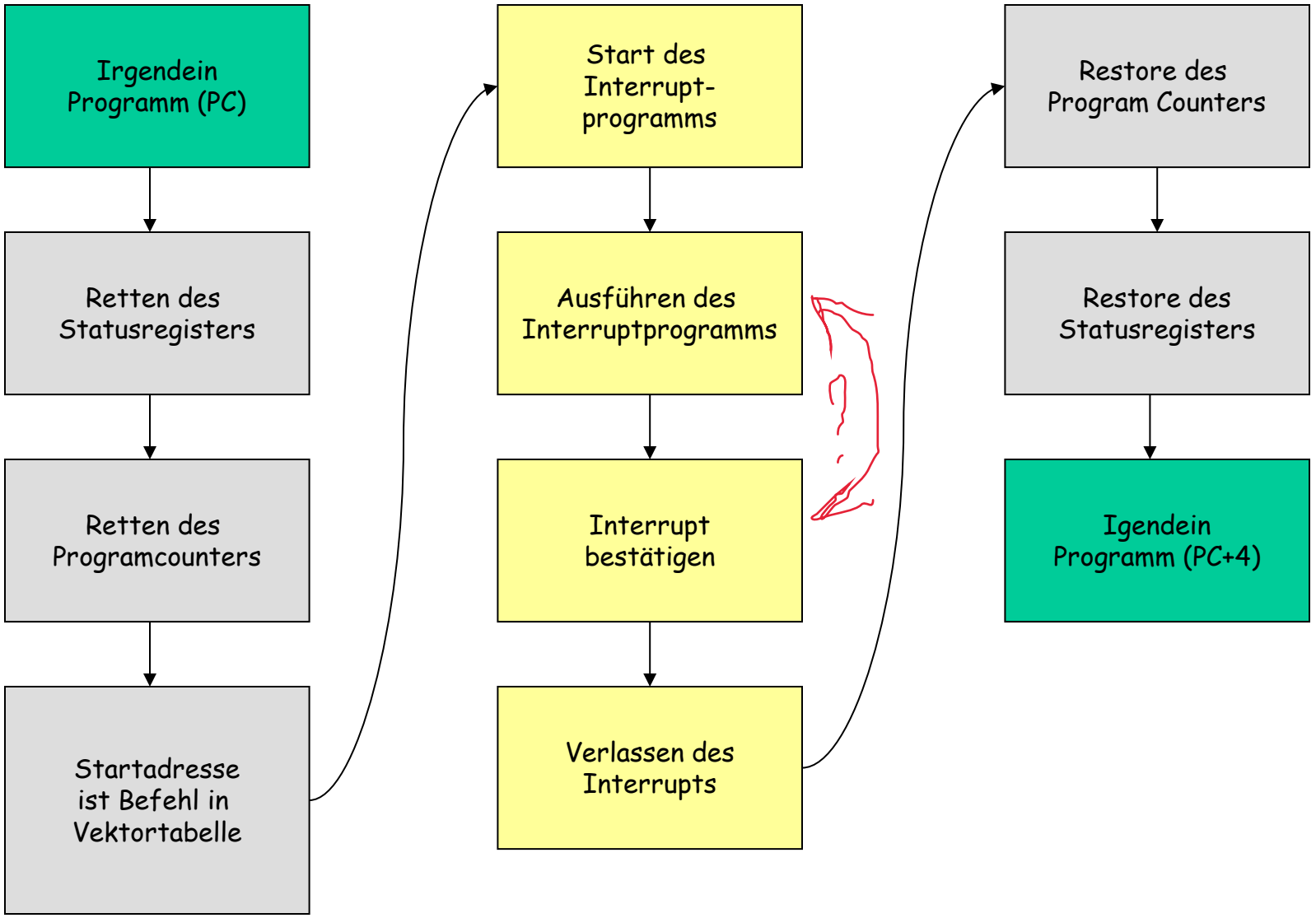
- In der Regel gibt es mehrere unabhängige Interrupt Quellen, die von der CPU einen Interrupt anfordern können.
- Die Verwaltung der Anforderungen benötigte selbst CPU-Zeit, die nicht zur Abarbeitung der eigentlichen Programme verwendet werden kann.
- Um die CPU von der Verwaltung der Interrupt Request zu entlasten, gibt es Interruptcontroller, die die Anforderungen verwalten und synchronisieren.

Mikrocontroller	Fähigkeiten
AT91	Advanced Interrupt Controller (AIC) mit 8 Prioritäten und zusätzlichen Fast Interrupt; erfordert explizite Programmierung der Kontextsicherung zur Unterbrechbarkeit niedrigpriorisierter Interrupts
SAM3X	Die Cortex-M Architektur verwendet den Nested Vector Interrupt Controller (NVIC) mit 30 maskierbaren Interrupts und 16 Prioritäten; transparente Kontextsicherung zur Unterbrechbarkeit niedrigpriosierter Interrupts; unterstützt Interrupt Preemption und Tail Chaining sowie Late Arriving für schnelle Kontextwechsel
RP2040	Die Cortex-M Architektur verwendet den NVIC (siehe SAM3X)

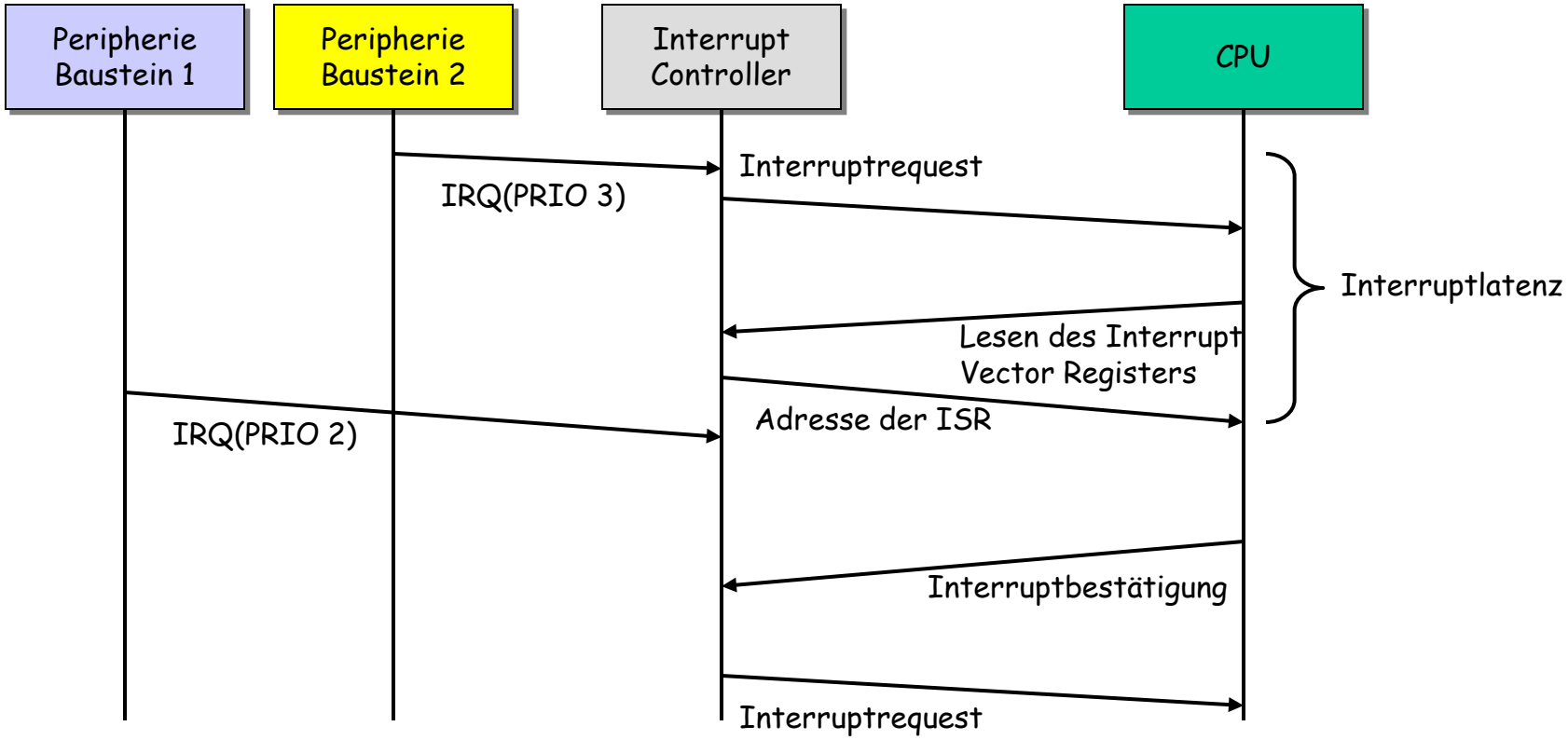
Interrupt Handling – Interruptanforderung



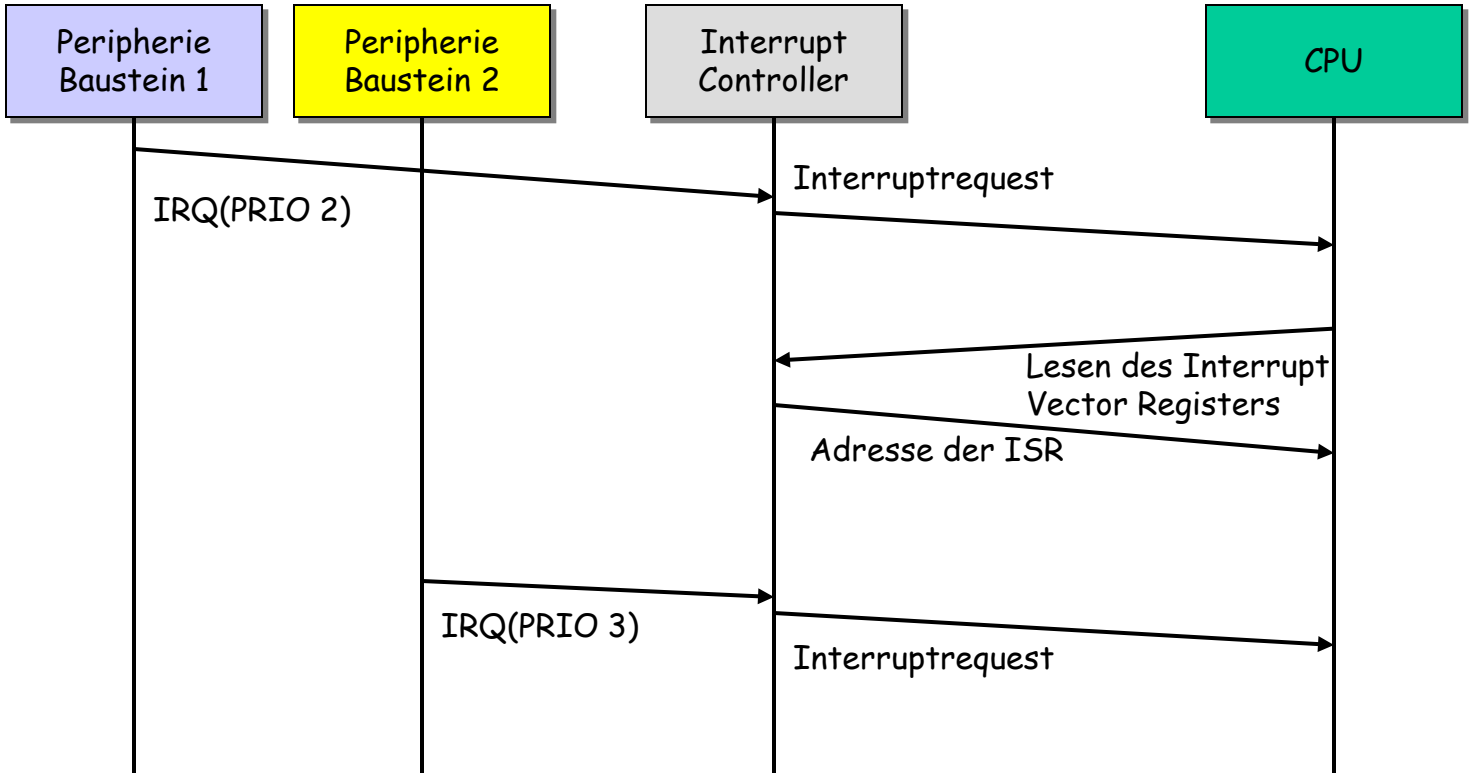
Interrupt Handling – beispielhafter Ablauf



Interrupt Handling – Priorität I



Interrupt Handling – Priorität II



Interrupt Handling - DMA

Asynchrone Ereignisbehandlung mit Direct Memory Access (DMA)

- Für sehr schnelle Reaktionen sind Interrupt-Softwareprogramme oft zu langsam. Besonders die schnelle Kommunikation (Gigabit Ethernet, USB, ...) erfordert eine Hardwareunterstützung um den notwendigen Datenstrom zu gewährleisten.
- Diese Hardwareunterstützung wird durch Direct Memory Access (DMA) Controller zur Verfügung gestellt. Dies erlaubt Peripherie-komponenten, direkt auf den Arbeitsspeicher zuzugreifen und damit sehr viel schneller Daten zu transferieren.
- DMA Controller können auf Interrupts reagieren und die Daten in den Speicher oder zum Peripheriegerät transferieren.
- Interrupts dienen so zur Ablaufkontrolle größerer Datentransfers, statt die Daten im Interrupt zu transportieren.

Interrupt Handling – Interrupt und Folgen

- Vorteil: schnellere Reaktion für den Interrupt höchster Priorität. Trennung verschiedener Ereignisquellen auf Programmebene möglich

- Problem: Nur höchster Interrupt hat garantierte Reaktionszeit, und auch nur dann, wenn der Interrupt nicht durch die laufenden Programme verboten wird (einige Betriebssysteme tun dies)

- Problem: Der Datentransport zwischen Interruptroutine und Anwendung ist sehr kritisch (Shared Data Problem)
  - Die Interrupt-Serviceroutine und das Hauptprogramm nutzen dieselben Daten.
  - Was passiert, wenn das Hauptprogramm gerade wichtige Berechnungen mit einem Datenelement durchführt … eine Unterbrechung auftritt, die dieses Datenelement verändert … und das Hauptprogramm dann seine Berechnung abschließt?
  - Mögliche Lösung: Im Hauptprogramm während dem Zugriff auf diese Datenelemente für kurze Zeit den Interrupt verbieten (critical section)

Interrupt Handling – Bewertung I

Vorteile von Polling (bzw. Busy Waiting) gegenüber Interrupts

- Einfacher zu implementieren, da Abfrage des Geräts im Hauptprogramm erfolgen kann.
- Busy Waiting erlaubt eine schnellere Reaktion auf das externe Ereignis. Es kann nach Erkennung des Ereignisses sofort reagiert werden. Bei Interrupts wird in eine **Interrupt Service Routine** (ISR) verzweigt: Kontextwechsel mit Sicherung der benutzten Register auf dem Stack.
- Geringerer Hardwareaufwand bei den Geräten notwendig, da diese nicht in der Lage sein müssen, auf das Auftreten eines Ereignisses mit einer Interrupt-Anfrage zu reagieren.
- Beispiel: ein Temperatursensor für Polling braucht im einfachsten Fall nur einen A/D-Wandler. Bei der Verwendung von Interrupts muss er jedoch neben der Möglichkeit, die Temperatur zu messen, zusätzlich auf die Änderung der Temperatur mit einer Interruptanfrage reagieren können.

Interrupt Handling – Bewertung II

Vorteile von Interrupts gegenüber Polling (Busy Waiting)

- Hauptprogramm wird deutlich einfacher und besser verständlich, weil es sich auf das Wesentliche konzentriert, ohne ständig oder von unterschiedlichen Codestellen aus Ereignisse abzufragen zu müssen.
- Das Auftreten eines externen Ereignisses wird **immer** überwacht. Beim Polling geschieht dies nur zu den Zeiten, zu denen das Hauptprogramm danach fragt.
- Das Hauptprogramm kann andere Aufgaben übernehmen, als das externe Gerät zu überwachen. Beim Polling wird häufig sämtliche Rechenleistung für die Abfrage der Geräte aufgewendet.
- Die Kommunikationsleitungen, Datenbusse und externen Geräte werden entlastet. Im Gegensatz zum Polling wird nur mit dem Gerät kommuniziert, wenn tatsächlich ein externes Ereignis stattfindet.
- Es ist oft stromsparender: Die CPU kann bis zum nächsten Auftreten eines Interrupts in einen stromsparenden Sleep-Mode versetzt werden (beim nächsten Interrupt automatisch wieder aktiviert).