Technische Hochschule Rosenheim
Technical University of Applied Sciences

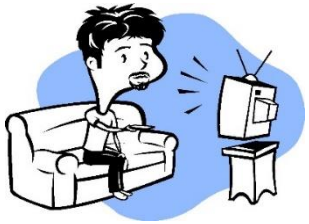
Embedded Systems
Kapitel 3: Interrupts, Speicher

Prof. Dr. Wolfgang Mühlbauer
Fakultät für Informatik
wolfgang.muehlbauer@th-rosenheim.de

Sommersemester 2020

"Multitasking"

- **Beispiel 1: „Zuhause“**
  - Aufgaben:
    - Fernsehen
    - Spaghetti kochen
  - Lösung: Kochtopf regelmäßig überwachen (**Polling**)



- **Beispiel 2**: „WG Party“
  - Aufgaben:
    - Küche aufräumen
    - Empfang erster Gäste
  - Lösung: Türklingel (**Interrupt**)

- **Beispiel 3**: „Mikrocontroller“
  - Mikrocontroller: Steuerung + Überwachung Lichtschranke
  - Mensch kommt in Nähe der Maschine, Lichtschranke wird unterbrochen
  - Mikrocontroller muss Programmausführung unterbrechen und auf Ereignis ***sofort*** reagieren (**Interrupt**)

Wie bekommt man ein Ereignis mit?

- **Busy Waiting**
  - Man wartet und blockiert bis ein Ereignis eingetreten ist.
  - Beispiel: `while (PINA & (1 << PA4))`

- **Polling:** *Periodisches / zeitgesteuertes System*
  - ***Programm*** überprüft Zustand regelmäßig und ruft ggfs. eine Bearbeitungsfunktion auf. Zwischendrin kann das Programm etwas anderes tun.
  - Ereignisbearbeitung erfolgt ***synchron*** zum Programmablauf.

- **Interrupt:** *Ereignisgesteuertes System*
  - Gerät "meldet" sich beim Prozessor, der daraufhin in eine Bearbeitungsfunktion verzweigt. Die Software wird vom Prozessor unterbrochen.
  - Ergebnisbearbeitung erfolgt ***asynchron*** zum Programmablauf.

Interrupts

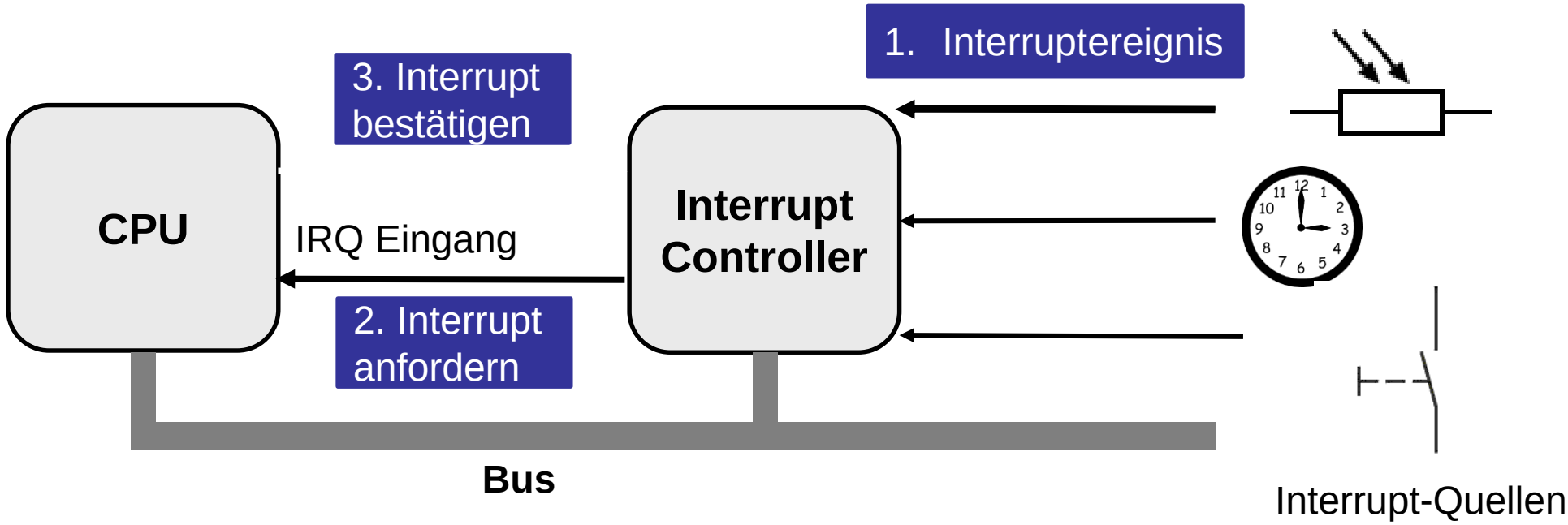
- ***Eigenschaften***
  - ***Asynchron:*** Unvorhersehbar wann genau Programm unterbrochen wird.
  - ***Nicht reproduzierbar*** bzw. vorhersehbar.

- ***Mögliche Quellen***
  - Externe Hardwareereignisse
    - Spannung an Eingang ändert sich, z.B. durch Tastendruck.
  - E/A oder DMA Operation beendet.
    - Tastatur, Maus, Drucker, Festplatte, Flash

Inhalt

- Einführung
- **Funktionsweise von Interrupts**
- Interrupt-Programmierung
- Speicher

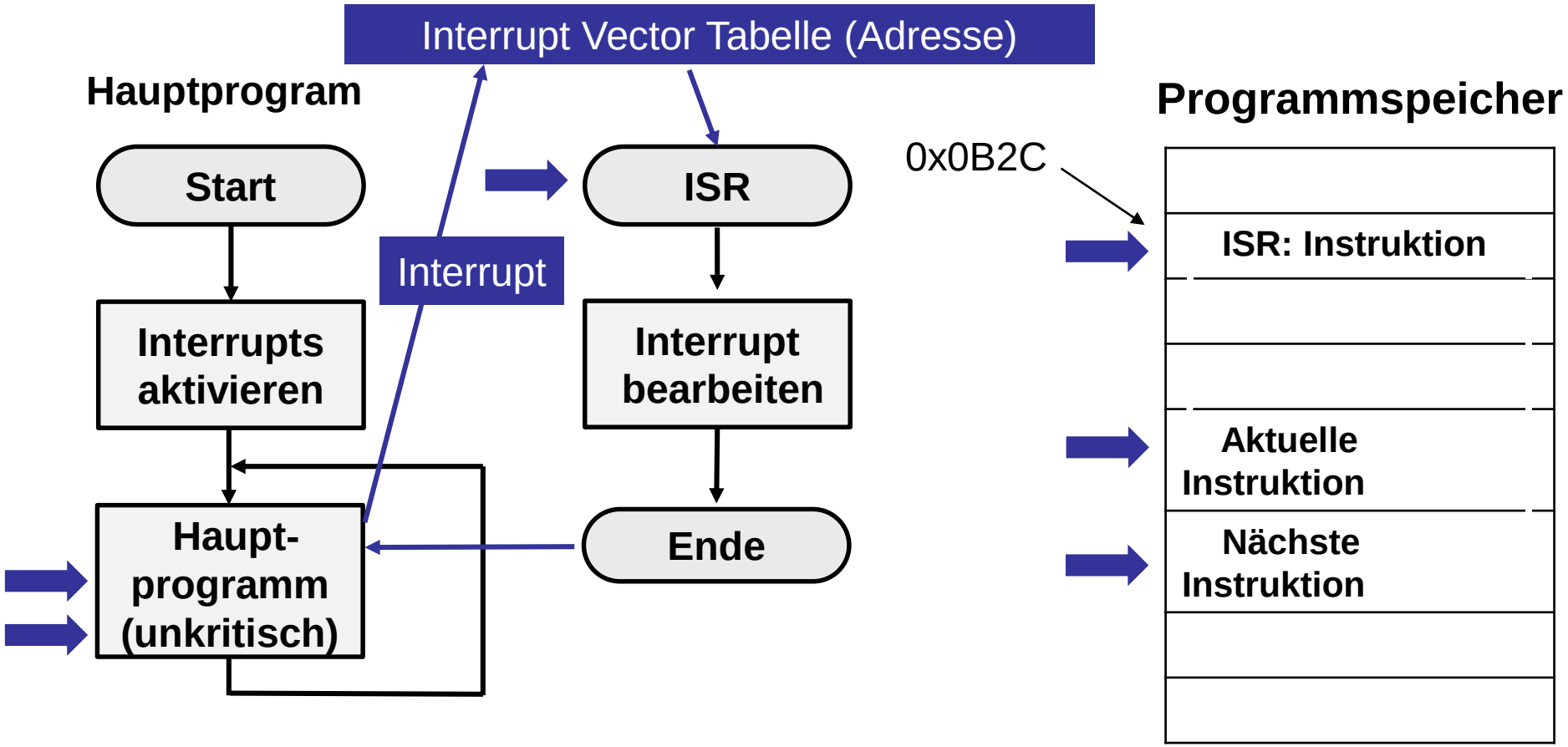
Interrupt Request



- ***Interrupt Controller***: Erkennen von Ereignissen + Priorisierung
  - IRQ Eingang: Unterbrechungsanforderung an CPU
  - Bus: Nummer (ID) des unterbrechenden Geräts bzw. Eingangspin

- CPU: Unterbricht Programm und startet ***Unterbrechungsroutine (=Interrupt Service Routine)*** an bekannter Adresse

Interrupt Service Routine (ISR)



ähnlich zu [3]

- Tritt ein Interrupt auf, unterbricht der ***Interrupt Controller*** die Verarbeitung des Hauptprogramms und verzweigt zu einer ***Interruptroutine (ISR).***
- Die ***ISR*** wird ausgeführt.
- Danach wird Hauptprogramm an Unterbrechungsstelle fortgesetzt.
- In der Regel: Während ISR-Ausführung sind weitere Interrupts gesperrt.

Interrupt Vector Table

- Nachschlagen: Welche ISR gehört zu welchem Interruptereignis?

- **„Fest verdrahtet“**
  - Jedes Ereignis hat eine Nummer („Vector No.“)
  - Jeder Nummer („Vector No.“) ist eine Programadresse („Program Address“) zugeordnet, zu der bei Eintreten des Ereignisses gesprungen wird. Dort liegt die ISR.

Table 14-1. Reset and Interrupt Vectors

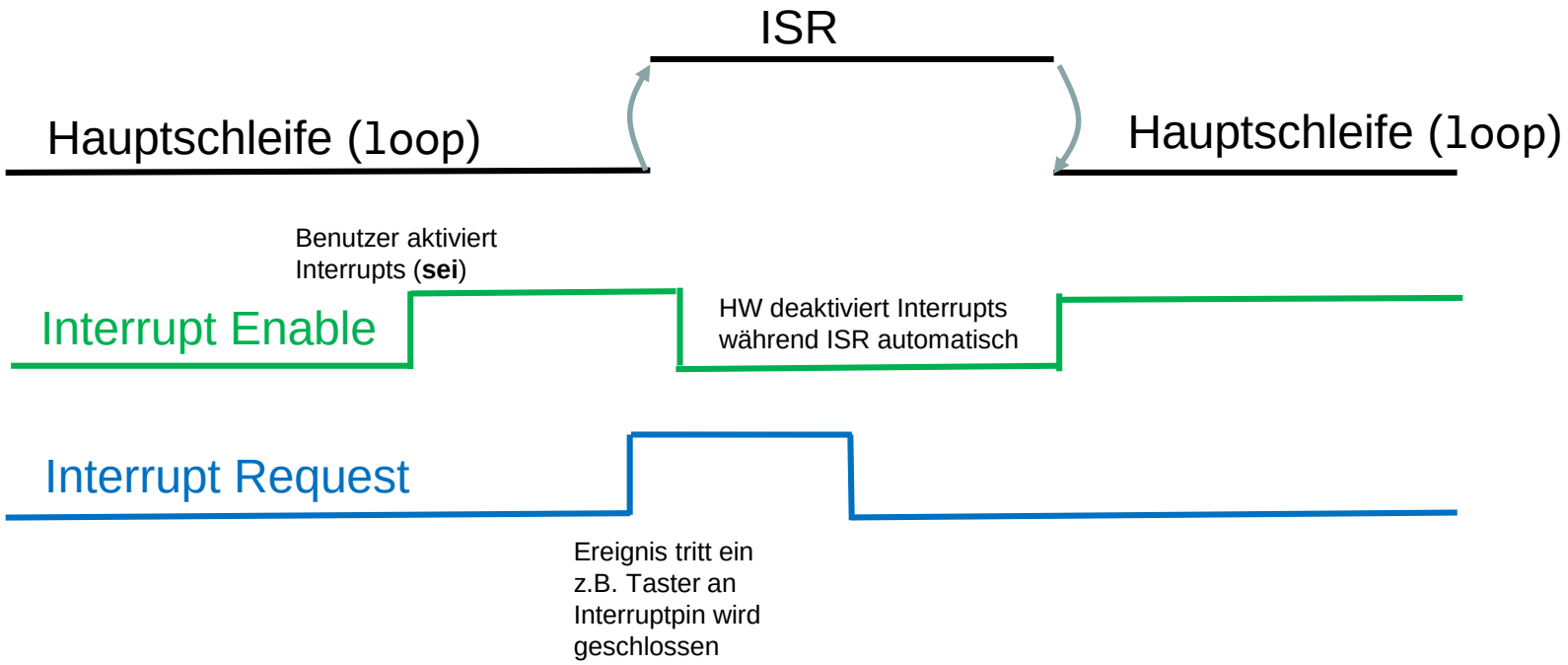
Vector No.	Program Address(2)	Source	Interrupt Definition
1	$0000(1)	RESET	External Pin, Power-on Reset, Brown-out Reset, Watchdog Reset, and JTAG AVR Reset
2	$0002	INT0	External Interrupt Request 0
3	$0004	INT1	External Interrupt Request 1
4	$0006	INT2	External Interrupt Request 2
5	$0008	INT3	External Interrupt Request 3
6	$000A	INT4	External Interrupt Request 4

Interrupt Vector Table ATmega2560 – **Auszug** aus Datenblatt [2, Seite 101ff.]

Interrupts: Zeitlicher Verlauf

- HW unterbricht aktuelle Programmausführung bei Eintreten eines Ereignisses.

Zeitlicher Verlauf eines Interrupts



Mehrere Interrupts

- ***ATmega***: Bereits laufende ISR nicht unterbrechbar durch weiteren Interrupt.
  - Zu Beginn der ISR: µController-HW deaktiviert automatisch weitere Interrupts (SREG Register: I-Bit)
  - ISR deshalb nicht unterbrechbar.
  - Am Ende werden dann Interrupts automatisch durch HW wieder aktiviert.

- **„Nested“ Interrupts**
  - Andere Mikrocontroller erlauben, dass Interrupts mit höherer Priorität eine aktuell laufende ISR unterbrechen.

- Bei mehreren gleichzeitig auflaufenden Interrupt Requests (IRQs):
  - Meist merkt man sich nur 1 IRQ pro Quelle.
  - Es können somit Requests verloren gehen.
  - Interrupts mit höherer Priorität werden bevorzugt behandelt.

Externe vs. interne Interrupts

- ***Externe Interrupts***
  - Controller tastet ***GPIO Pin*** zu Beginn jedes Taktzyklus ab.
  - Falls Interrupt aktiviert: Aufruf der ISR.
  - Probleme
    - Abtastung verursacht leichte *Zeitverzögerung* bis Ereignis erkannt wird.
    - *Spurious Interrupts*: Ggfs. HW-/SW-Entprellung notwendig.

- ***Interne Interrupts***
  - Beispiele: Timer, A/D-Wandler, etc.
  - Timer läuft aus → HW unterbricht Ausführung der "normalen" Software.

- *Hinweis*: Bei Atmega2560 gibt es 2 Typen externer Interrupts
  - *Echter Interrupt*: Pins, für die es eine eigene ISR gibt.
  - *Pin Change Interrupts*: Ports, bei denen sich alle Pins 1 ISR teilen.

Inhalt

- Einführung
- Funktionsweise von Interrupts
- **Interrupt-Programmierung**
- Speicher

Interrupt Konfiguration: Allgemeines Vorgehen

- ***Globales Aktivieren von Interrupts***
  - Interrupt-Funktionalität kann komplett abgeschaltet werden.
  - Dann funktionieren keinerlei Interrupts.

- ***Aktivieren einzelner Interrupts***
  - ISR wird bei Eintritt eines Ereignisses nur aufgerufen falls
    - Interrupts global aktiviert sind und
    - falls betreffender Interrupt explizit aktiviert ist (*Interrupt Enable Bit)*.

- ***Interrupt Flags***
  - Signalisieren, *ob* und *welches* Interrupt-Ereignis aufgetreten ist.
  - Werden gesetzt durch Hardware.
  - Jederzeit abfragbar durch Software, auch außerhalb einer ISR.
  - Flag wird am Ende der ISR meist automatisch gelöscht.

- ***Interrupt Modus***
  - Was triggert einen Interrupt?
  - Steigende oder fallende Flanke? LOW? HIGH?

ATmega2560: Konfiguration externer Interrupts

- Globales Aktivieren: **SREG**-Register, I-Bit
  - Wird durch C-Kommando `sei()` gesetzt und durch `cli()` gelöscht.

Bit	7	6	5	4	3	2	1	0	
0x3F (0x5F)	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	

- Aktivieren einzelner Interrupts: **EIMSK** Register

Bit	7	6	5	4	3	2	1	0	
0x1D (0x3D)	INT7	INT6	INT5	INT4	INT3	INT2	INT1	INT0	EIMSK
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	

- Interrupt Flags: **EIFR** Register

Bit	7	6	5	4	3	2	1	0	
0x1C (0x3C)	INTF7	INTF6	INTF5	INTF4	INTF3	INTF2	INTF1	IINTF0	EIFR
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	

- Interrupt Modus, **EICRA** Register
  - Steigende / fallende Flanke: Datenblatt Kap. 15.2.1 und 15.2.2

Quelle: [1]

AVR-Libc vs. Arduino

- **AVR-Libc**
  - `#include <avr/interrupt.h>`
  - Globales De-/Aktivieren von Interrupts mit `sei()` bzw. `cli()`.
  - Konfiguration der Register: z.B. EIMSK, EICRA, etc.
  - Definition einer ISR mit Makro `ISR(vector)`
    - `vector`: Bezeichnet Interrupt-Quelle
    - http://www.nongnu.org/avr-libc/user-manual/group__avr__interrupts.html
- **Arduino Library**
  - `attachInterrupt(<quelle>, <ISR name>, <mode>)`
  - Aktiviert einen Interrupt, ordnet eine ISR zu und konfiguriert bei welchem Signalverlauf der Interrupt aktiviert werden soll.
  - https://www.arduino.cc/en/Reference/HomePage

Exkurs: Schlüsselwort "volatile"

Kein Prüfungsstoff

- **`volatile`**
  - Teilt Compiler mit, dass Inhalt der Variablen
    - vor jedem Lesezugriff aus Speicher gelesen und
    - nach jedem Schreibzugriff in Speicher geschrieben wird.
- **Problem:** Compiler-Optimierung
  - Compiler geht bei Übersetzung von *while*-Schleife davon aus, dass sich der Wert von *i* innerhalb der Schleife niemals ändern kann. Er hält deshalb Variable *i* ggfs. in einem Register und wertet diese nur von dort aus.
  - Diese Annahme ist jedoch falsch: Eine ISR kann zu jeder Zeit unterbrechen. Der korrekte Wert von *i* = 5 steht dann im SRAM, wird aber innerhalb der while-Schleife mit i = 1 aus Register gelesen.
- Compiler-Anweisung **volatile**
  - Auf die Variable ***i*** wird *immer* im SRAM zugegriffen.
  - **Richtlinie**: Globale Variablen, die in ISR vorkommen, sollten immer als ***volatile*** markiert werden.

`volatile uint8_t i`

```
uint8_t i;

void setup() {
  Serial.begin(9600);
  i = 1;
  Interrupt INT 0 wird aktiviert
}

void loop() {
   while (1) {
     if (i == 5) {
       Serial.println("Bin da");
     }
  }
}

ISR (INT0_vect)
{
  i = 5;
}
```

Register-Zugriff

SRAM-Zugriff

Polling vs. Interrupts: Vor- und Nachteile

- **Polling**
  - Nachteile
    - Ereigniserkennung muss ggfs. über das Programm "verstreut" werden.
    - Hochfrequentes Pollen → hohe Prozessorlast → hoher Energieverbrauch
  - Vorteile
    - Implizite Datenkonsistenz durch festen, sequentiellen Programmablauf
    - Programmverhalten gut vorhersagbar

- **Interrupts**
  - Nachteile
    - Höhere Komplexität durch Nebenläufigkeit → Synchronisation erforderlich
    - Programmverhalten schwer **vorhersagbar.**
  - Vorteile
    - Ereignisbearbeitung kann im Programmtext gut separiert werden
    - Prozessor wird nur beansprucht, wenn Ereignis tatsächlich eintritt

- Beide Verfahren bieten spezifische Vor- und Nachteile → Auswahl anhand des konkreten Anwendungsszenarios

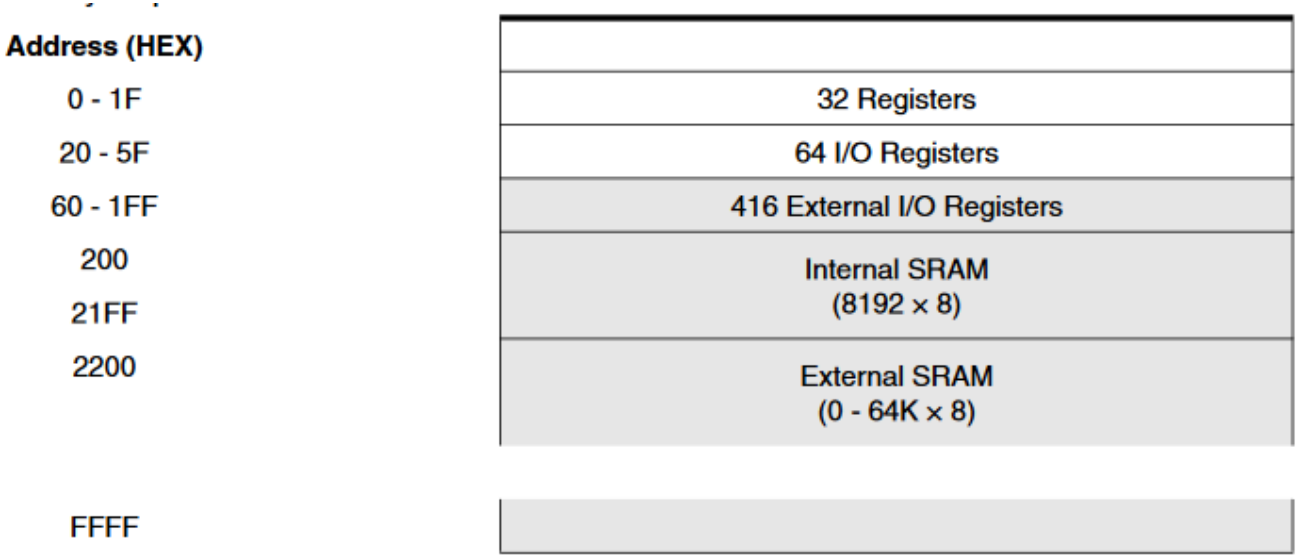
Inhalt

- Einführung
- Funktionsweise von Interrupts
- Interrupt-Programmierung
- **Speicher**

Speicher in Mikrocontrollern

- ***Register***
  - Kleiner, schneller Speicherbereich in der CPU
  - Für Berechnungen und Zugriff auf µController-HW: Ports und Schnittstellen

- ***Harvard-Architektur***
  - Daten- und Instruktionsspeicher sind getrennt (anders als bei „von Neumann“)
  - Instruktionsspeicher: Maschinenbefehle liegen in nicht-flüchtigem Flash Speicher
  - Daten: Temporäre Daten während Programmausführung können in flüchtigem SRAM abgelegt werden.



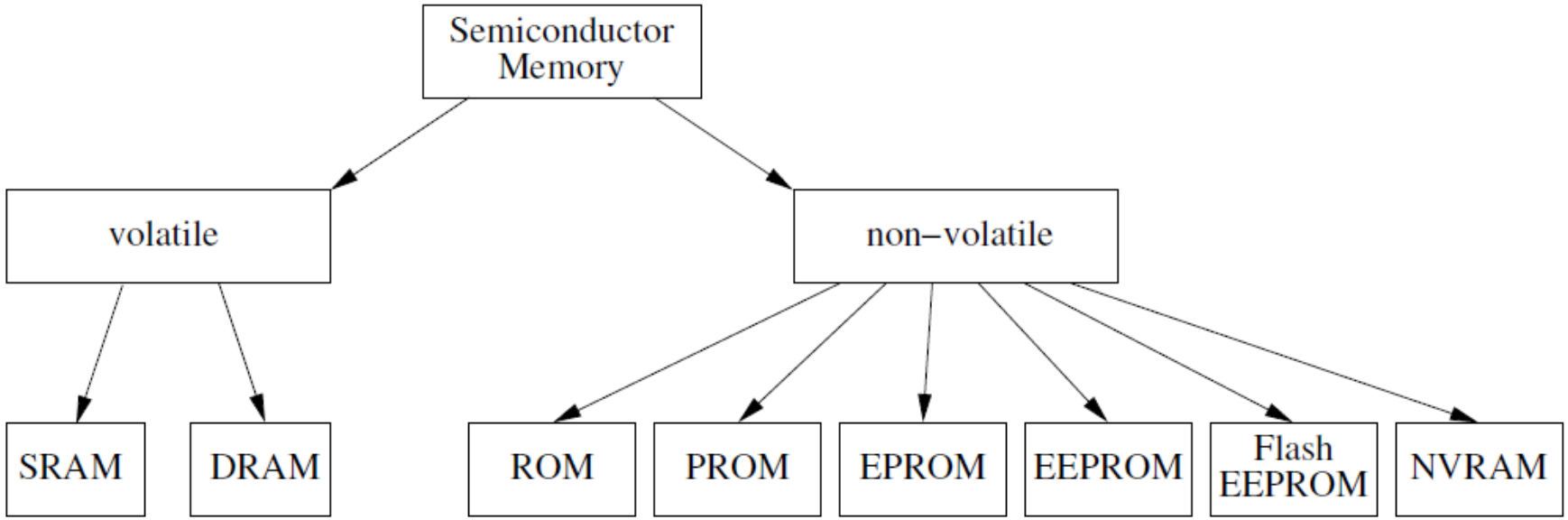
Register im Atmega2560
- 32 8-Bit Arbeitsregister
- 64 8-Bit I/O Register (für GPIO Pins)
- 416 Register zum Ansteuern von HW wie AD-Wandler, Timer, etc.

Quelle: [2, S.22]

Speicherklassifizierung nach Bautyp

- ***Flüchtig vs. nichtflüchtig***
  - Flüchtig: Datum geht ohne Stromzufuhr verloren, z.B. nach Neustart.

- ***In-System Programmierung von nichtflüchtigen Speichern.***
  - CPU kann zur Laufzeit (Programm)speicher schreiben.
  - Beispiel: Arduino erlaubt Laden eines Programms über USB Schnittstelle zur Laufzeit.



Typen von Halbleiterspeichern

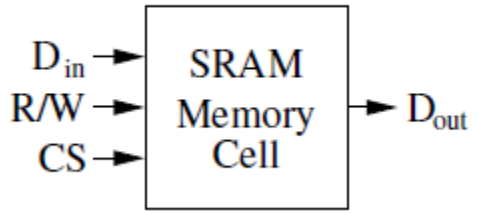
Quelle: [1]

Flüchtiger Speicher: SRAM und DRAM

- ***Static RAM (SRAM)***
  - Besteht aus 1-Bit Speicherzellen (Flipflops)
  - `CS = 1`: Wert am Eingang D_{in} wird auf Ausgang D_{out} übernommen.
  - `CS = 0`: Wert am Ausgang D_{out} ändert sich nicht, unabhängig vom Eingang D_{in}
  - Anordnung einzelner Zellen zu Matrix, siehe [1, S. 30]
  - Teuer, aber schnell!

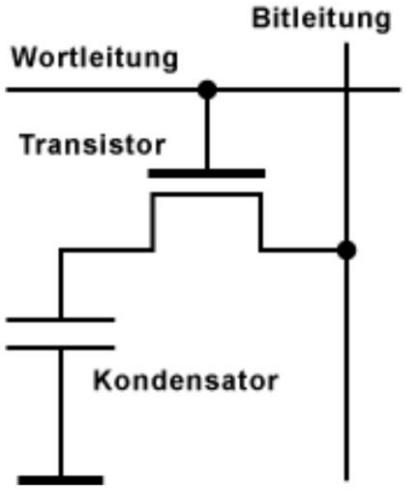
- ***Dynamic RAM (DRAM)***
  - Vorteil: Platzersparnis: 1 statt 6 Transistoren pro Bit
  - Transistor entscheidet ob Read oder Write, siehe rechts
  - Kondensator speichert Ladung, muss aber periodisch geladen werden.
  - Nachteile: Langsamer als SRAM, komplexere Ansteuerung

- Häufig: SRAM bei Mikrocontrollern, DRAM bei PCs



Blackbox SRAM

Quelle: [1]



Aufbau einer DRAM Speicherzelle Quelle [6]

Nichtflüchtiger Speicher

- ***Read Only Memory* (ROM)**
  - Bitinformation bereits während der Fertigung fest „verdrahtet“
  - Kein späteres Schreiben möglich.

- ***One-Time Programmable Read Only Memory* (OTPROM)**
  - Matrizen von Speicherzellen mit Silikon-Sicherung
  - Hohe Spannung zerstört Sicherung, Bit wird dauerhaft auf 1 gesetzt

- ***Electrically Erasable and Programmable ROM* (EEPROM)**
  - Bits werden mittels spezieller Transistoren gespeichert.
  - Elektronen können ähnlich wie in einem Kondensator gespeichert werden.
  - Nur sehr langsame Entladung, wenn keine Stromversorgung besteht!
  - Kann nur durch Anlegen einer höheren Spannung entladen werden
  - **Nachteil: Begrenzte Anzahl von Schreib- und Lesezyklen.**

- ***Flash***
  - Kostengünstiger als EEPROM.
  - Einfachere Zugriffslogik durch Lesen und Schreiben größerer Datenblöcke (nicht individueller Bytes wie bei EEPROM)

Speichereinsatz in typischen Mikrocontrollern

- ***Flash***
  - Programmdaten / Programmcode
  - Bei "Produkten" wird der Programmcode selten geändert → Firmware
  - Programmcode muss über "Bootloader" geladen werden (siehe späteres Kapitel)

- ***EEPROM***
  - Nichtflüchtige Daten.
  - Konfigurationsdaten
  - Kalibrierungsdaten

- ***SRAM***
  - "Arbeitsspeicher" des Mikrocontrollers
  - Speichert flüchtige Daten während der Laufzeit.
  - Register, Stack, etc.

Quellenverzeichnis

[1] G. Gridling und B. Weiss. *Introduction to Microcontrollers*, Version 1.4, 26. Februar 2007, Kapitel 2.5, verfügbar online: https://ti.tuwien.ac.at/ecs/teaching/courses/mclu/theory-material/Microcontroller.pdf (abgerufen am 08.03.2017)

[2] Datenblatt ATmega2560, http://www.atmel.com/Images/Atmel-2549-8-bit-AVR-Microcontroller-ATmega640-1280-1281-2560-2561_datasheet.pdf, (abgerufen am 19.03.2017)

[3] AVR-GCC Tutorial, https://www.mikrocontroller.net/articles/AVR-GCC-Tutorial#Programmieren_mit_Interrupts (abgerufen am 02.04.2017)