

# README

## Beschreibung

Zusammenfassung für Computer Engineering 2 auf Grundlage der Vorlesung FS 16 von Erwin Brändle  
Bei Korrekturen oder Ergänzungen wendet euch an einen der Mitwirkenden.

## Modulschlussprüfung

Kompletter Stoff aus Skript, Vorlesung, Übungen und Praktikum

- Vorlesungsskript CompEng2 V1.2 komplett  
(die Kapitel 2 und 7 sind im Selbststudium individuell aufzuarbeiten)
- Korrigenda zum Skript, falls eine solche vorliegt
- Übungen im Vorlesungsskript
- Inhalt aller Praktika (inkl. Pre-/Post-Lab Übungen)
- in Vorlesungen und Praktika zusätzlich vermittelte Informationen
- Inhalt und Umgang mit dem Quick-Reference/Summary V1.2

**Die Prüfung besteht aus 2 Teilen:**

- |                |                |   |
|----------------|----------------|---|
| <b>1. Teil</b> | closed Book    | Theoretische Fragen zum ganzen Prüfungsinhalt                                       |
| <b>2. Teil</b> | semi-open book | Aufgaben im Stil der Übungen, Praktika und der in den Vorlesungen gelösten Aufgaben |

## Plan und Lerninhalte

Fokus: ARM Cortex-M Architektur

- RISC-Architektur, Core-Components, Register Model, Memory Model, Exception Model, Instruction Set Architecture
- Konzept und Umsetzung der vektorisierten Interrupt Verarbeitung
- Abbildung von typischen C Programmstrukturen und Speicherklassen in das Programmiermodell der CPU
- Systembus: Address-, Daten-, Control-Bus, Adressdekodierung, Memory- und I/O-Mapping
- Speicher- und ausgesuchte Peripherieschnittstellen

## Contributors

Luca Mazzoleni luca.mazzoleni@hsr.ch

Stefan Reinli stefan.reinli@hsr.ch

Flurin Arquint flurin.arquint@hsr.ch

## License

Creative Commons BY-NC-SA 3.0

Sie dürfen:

- Das Werk bzw. den Inhalt vervielfältigen, verbreiten und öffentlich zugänglich machen.
- Abwandlungen und Bearbeitungen des Werkes bzw. Inhaltes anfertigen.

Zu den folgenden Bedingungen:

- Namensnennung: Sie müssen den Namen des Autors/Rechteinhabers in der von ihm festgelegten Weise nennen.
- Keine kommerzielle Nutzung: Dieses Werk bzw. dieser Inhalt darf nicht für kommerzielle Zwecke verwendet werden.
- Weitergabe unter gleichen Bedingungen: Wenn Sie das lizenzierte Werk bzw. den lizenzierten Inhalt bearbeiten oder in anderer Weise erkennbar als Grundlage für eigenes Schaffen verwenden, dürfen Sie die daraufhin neu entstandenen Werke bzw. Inhalte nur unter Verwendung von Lizenzbedingungen weitergeben, die mit denen dieses Lizenzvertrages identisch oder vergleichbar sind.

Weitere Details: <http://creativecommons.org/licenses/by-nc-sa/3.0/ch/>

# ComEng2 Zusammenfassung

L. Mazzoleni S. Reinli F. Arquint

18. Dezember 2019

## Inhaltsverzeichnis

<b>1 V1</b>	<b>2</b>
1.1 Anwendung und Grundlage der uP-Technik . . . . .	2
1.2 Aufbau . . . . .	2
1.3 RISC vs CISC . . . . .	3
1.4 Hardware . . . . .	4
1.5 Software . . . . .	4
<b>2 V2</b>	<b>5</b>
2.1 Compiler-Schritte . . . . .	5
2.2 Busorientierte Systeme . . . . .	5
2.3 Befehlszyklus . . . . .	6
<b>3 V3</b>	<b>7</b>
3.1 Halbleiter Speicher . . . . .	7
3.2 Speicherorganisation . . . . .	7
3.3 Busanschluss und Adressverwaltung . . . . .	8
<b>4 V4</b>	<b>9</b>
4.1 Cortex M Varianten . . . . .	9
4.2 Cortex-M3/M4 . . . . .	10
4.3 System-Komponenten . . . . .	10
4.4 GNU-Tool-Chain Entwicklungsablauf . . . . .	11
4.5 Programm Status Register . . . . .	11
4.6 Stack . . . . .	12
<b>5 V5</b>	<b>13</b>
5.1 Data Alignment . . . . .	13
5.2 Bit-Banding . . . . .	13
<b>6 V6</b>	<b>14</b>
6.1 Exceptions and Interrupts . . . . .	14
6.2 Reset und Reset-Sequenzen . . . . .	14
6.3 Fault-Handling . . . . .	15
6.4 Spezial-Register . . . . .	15
<b>7 V7 (ausgefallen)</b>	<b>16</b>
<b>8 V8</b>	<b>16</b>
8.1 Cortex M3 Instruction Set . . . . .	16
8.2 Instruction Pipelining . . . . .	16
8.3 Logikstruktur des Cortex-M Prozessor . . . . .	16
8.4 Anwendungen . . . . .	16

8.5	Software Development Prozess . . . . .	17
8.6	Assembly-Language Syntax . . . . .	17
8.7	Unified Assembler Language (UAL) . . . . .	17
8.8	Addressing . . . . .	17
<b>9</b>	<b>V9</b>	<b>19</b>
9.1	Interne Datenverschiebung . . . . .	19
9.2	Memory Access Instructions . . . . .	19
9.3	Stack Push und Pop . . . . .	19
9.4	Shift and Rotate Instructions . . . . .	20
9.5	Bit-Field Processing Instructions . . . . .	20
9.6	Compare and Test . . . . .	20
9.7	Program Flow Control . . . . .	20
<b>10</b>	<b>V10</b>	<b>21</b>
10.1	C/C++ Strukturen Umsetzen . . . . .	21
<b>11</b>	<b>V11</b>	<b>23</b>
11.1	Subroutinen . . . . .	23
11.2	Architecure Producer Call Standart (AAPCS) . . . . .	23
<b>12</b>	<b>V12, Serielle Schnittstelle</b>	<b>24</b>
12.1	Grundlagen . . . . .	24
12.2	Synchrone Datenübertragung . . . . .	26
12.3	Asynchrone Datenübertragung . . . . .	26
12.4	Standardisierte serielle Interfaces . . . . .	27
<b>13</b>	<b>V13</b>	<b>28</b>
13.1	Kommunikation zwischen Rechnersystemen . . . . .	28
13.2	Allgemeiner Ablauf von Exceptions und Interrupts . . . . .	29
13.3	Alternative Ansätze . . . . .	29
<b>14</b>	<b>V14</b>	<b>30</b>
14.1	Spezielle Eigenschaften des NVIC . . . . .	30
14.2	Cortex-M3 Exceptions und Priority-Levels . . . . .	30

# 1 V1

## 1.1 Anwendung und Grundlage der uP-Technik

### 1.2 Aufbau

Versteh die wesentlichen Systemkomponenten des Rechnersystems auf einem IC (Integrated Circuit)

#### 1.2.1 Anwendungen

- Supercomputer
- Arbeits und Server-Rechnern
- Smartphones
- Navigationssysteme
- Digitalkameras
- Drucker
- ...

#### 1.2.3 Havard vs Von Neumann Architektur

Harvard Rechnermodell

#### 1.2.2 Aufbau von uP-basierten Systemen

- Zentraleinheit CPU mit
  - Rechenwerk ALU
  - Steuerwerk CU
  - Registersatz
- Speicher
- Eingabe-/Ausgabe-Schnittstellen

von Neumann Rechnermodell

## 1.2.4 Programmierung eins uP

Ein  $\mu$  P kann durch individuelle Programmierung auf ganz unterschiedliche Art angepasst werden.

→ entscheidend für die Durchdringung im Markt.

Ein Programm enthält in aufeinanderfolgender Anordnung die Maschinen-Befehle oder -Instruktionen für den  $\mu$  P. Diese Maschiene-Befehle teilen der CPU mit, welche Operationen in welcher Reihenfolge und auf welche Daten angewendet werden sollen.

Die Befehlsfolge des Programms wird innerhalb der CPU vom Steuerwerk gesteuert und schrittweise ausgeführt. Dazu wird der aktuell zur bearbeitende Befehl durch einen Programmzähler (PC) im Speicher adressiert.

Der PC enthält laufend die Adresse der Speicherzelle des jeweiligen Befehls im Speicher.

## 1.2.5 Befehlsformate

Die Art und Wirkung eines Befehls wird im Befehlswort (**OpCode**) codiert. Darin sind neben der Operation auch die Operanden spezifiziert. Die Codierung des Befehlswortes erfolgt abhängig vom  $\mu$  P. Der Maschinencode setzt sich aus einem OpCode und einem oder mehreren Operanden zusammen.

## 1.3 RISC vs CISC

### CISC

Complex Instruction Set Computer

### RISC

Reduced Instruction Set Computer

### 1.3.1 RISC-Rechner

effizienter als CISC-Rechner

- besteht aus einer kleinen Anz. von Befehlen mit wenigen Adressierungsarten
- Registersatz enthält eine grosse Anzahl von allg. verwendbaren Registern General Purpose Register (GPR)
- Speicherzugriff erfolgt über spezielle Lade- und Speicher-Befehle
  - Arithmetisch-logische Operationen arbeiten auf Registeroperanden
- Pipeline-Architecture ← Leistungssteigernde Architektur
- Eine grosse semantische Lücke entsteht bei der Übersetzung aus der Hochsprache

### 1.3.2 u Architektur

Beschreibt die architektonischen Details bei der Implementierung der  $\mu$  P aus Sicht der Programmierer. Dies umfasst die Beschreibung der Zentraleinheit (CPU), des Rechenwerks (ALU) und des Steuerwerks (CU).

## 1.4 Hardware

### 1.4.1 Registersatz

Register sind schnelle Zwischenspeicher für temporäre Daten im  $\mu$  P.

### 1.4.3 Taktfrequenz

Das Taktsignal steuert die zeitliche Abfolge im  $\mu$  P

$$f_{Takt} = \frac{1}{T_{takt}}$$

### 1.4.2 Hardware- /Software-Schnittstelle

$\uparrow$  Taktrate  $\Leftrightarrow$   $\uparrow$  Leistungsaufnahme  
Um Energie zu sparen ist es sinnvoll die Taktrate laufend anzupassen.

### 1.4.4 Leistungsaufnahme

$$P_{Gate} = \frac{1}{2} \cdot C_{Last} \cdot V_{DD}^2 \cdot f_{Takt}$$

$P_{Gate}$	Leistung pro CMOS Gate
$C_{Last}$	Lastkapazität
$V_{DD}$	Versorgungsspannung
$f_{Takt}$	Taktfrequenz

## 1.5 Software

### 1.5.1 Ablauf

- Der **Präprozessor** bereitet das Quellprogramm für den Compiler vor
- Der **Compiler** übersetzt das Programm von einer Hochsprache in ein Assembly-Programm
- Der **Binder** fasst verschiedene Dateien, die verschiebbaren Maschinencode enthalten, zu einem Programm zusammen.
- Der **Loader** wandelt die verschiebbaren Adressen in absolute Adressen um und lädt sie in den Speicher des Systems.

## 2 V2

### 2.1 Compiler-Schritte

#### 1. Lexikalische Analyse (Scanning):

Die Symbole der Sprache werden erkannt und Gruppiert. Leerzeichen werden eliminiert

#### 2. Syntaxanalyse (Parsing):

Die erkannten Symbole werden in Sätzen zusammengefasst und in einem Parsbaum dargestellt

#### 3. Semantische Analyse:

Das Quellprogramm wird auf Fehler überprüft (zBsp. Typfehler) und der Parsbaum erhält Informationen über die verwendeten Bezeichner

#### 4. Zwischencode-Erzeugung:

Einige Compiler erzeugen Code in einer Zwischensprache (abstrakte Maschinen)

#### 5. Code-Erzeugung:

Erzeugen von verschiebbarem Maschinencode.

### 2.2 Busorientierte Systeme

#### 2.2.1 Speicher

##### RAM

- Random Access Memory
- Schreibe-/Lese-Speicher
- Spannungsversorgung erforderlich
- für temporäre Daten

##### Adressbus

- unidirektional
- bestimmt Grösse des Adressraums

\* Die gesammte Menge der über den Adressbus adressierbaren Speicherzellen wird **Adressraum** genannt

\* Die Anzahl parallel geführten **Datenleitungen** entspricht der maximal zu übertragenden Datenbreite

\* Kontrollsignale werden über den **Steuerbus** übertragen

##### ROM

- Read Only Memory
- Festwert Speicher
- auch ohne Spg. bleiben Daten erhalten

##### Datus

- bidirektional

##### Steuerbus

- kontrolliert Buszugriffe
- zeitlicher Ablauf der Signale

#### 2.2.2 Architektur eines uP

AR	Adressregister
PC	Programm Counter
PSR	Program Status Register
IR	Instruction Register

Flags, welche sich unter anderem im PSR befinden

N	<b>Negative</b>	Das von der ALU berechnete Ergebniss ist negativ
Z	<b>Zero</b>	Das von der ALU berechnete Ergenis ist gleich 0
C	<b>Carry</b>	Die Berechnung der ALU hat zu einem Übertrag geführt
V	<b>Overflow</b>	Die Berechnung der ALU hat zu einem Overflow geführt

## 2.3 Befehlszyklus

## 3 V3 3.1 Halbleiter Speicher

### Zentraler Speicher

- direkt am Bussystem angeschlossen

### Peripherer Speicher

- über I/O-Schnittstelle angeschlossen

#### 3.1.1 ROM-Festwertspeicher

#### 3.1.2 RAM-Speicher-/Lese-Speicher

## 3.2 Speicherorganisation

#### 3.2.1 Little/Big Endian

Der ARM Cortex verwendet standardmäßig **LittleEndian** für die Speicherorganisation.

#### 3.2.3 Speicherraumadressierung

#### 3.2.2 I/O - Schnittstelle

##### Datenregister

- enthalten die zu verarbeitenden Daten

##### Steuerregister

- dienen zur Konfiguration der Ein-/Ausgabe Schnittstelle

##### Statusregister

- signalisieren den Zustand der Ein-/Ausgabe Schnittstelle

##### Isolierte Adressierung

- Gesamter Adressraum steht verschiedener Blöcken zur Verfügung
- Blöcke werden mit einem Steuersignal ausgewählt

##### Memory-Mapped I/O (Cortex M3)

- Verschiedene Blöcke werden in einen Adressraum eingebettet
- Benötigt eine Adresskodierung

### 3.3 Busanschluss und Adressverwaltung

Eine Adressverwaltung hat verschiedene Aufgaben zu erfüllen:

- Jede Adresse spricht nur einen einzigen Speicher- oder I/O-Baustein an.
- Adressraum möglichst gut ausnützen
- Adressräume von Speicherbausteinen müssen lückenlos aufeinander folgen.
- Jeder interne Speicherplatz bzw. jedes Register erscheint unter einer eigenen Adresse im Systemadressraum.

#### 3.3.1 Adresskodierung

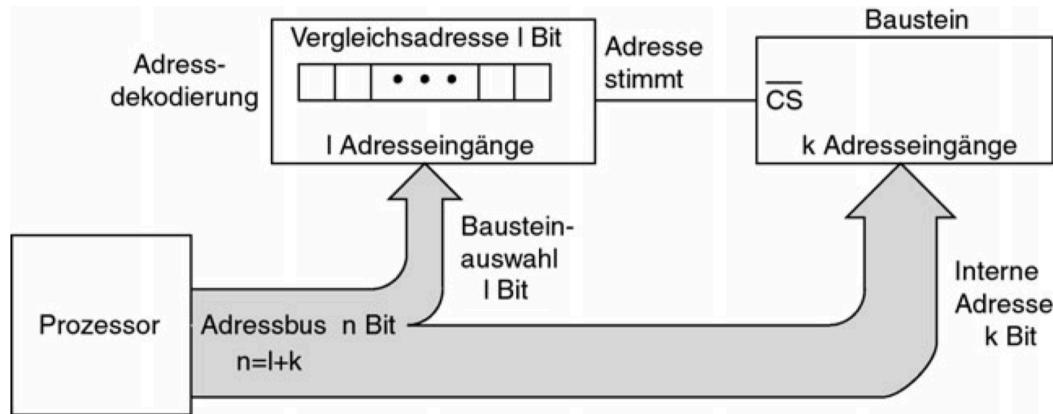


Figure 1-30 Grundprinzip der Adressdekodierung [Wüst]

Der Kernpunkt der Adresskodierung ist die Teilung des Adressbuses. Dabei werden die  $k$  niedrigsten Adressleitungen direkt an die Adresseingänge des Bausteines geführt und dienen zur Auswahl des gewünschten internen Speicherplatzes oder Registers. Die nächstfolgenden  $l$  Adressleitungen werden zur Adresskodierung auf einen Adressdekoder geführt.

## 4 V4

### 4.1 Cortex M Varianten

#### Cortex M0 und M0+

- kleinster Vertreter der CortexFam
- Ersatz von 8Bit- uC

#### Cortex M3

- erster Vertreter der CortexFam
- 32 Bit Architektur
- ersetzt 8 & 16 Bit uC
- Thumb ISA (Instruction Set Architecture)  
Mix aus 16 und 32Bit langen anweisungen

#### Cortex M1

- als Softcore Implementiert
- Vergleichbar mit Cortex-M0

#### Cortex M4

- vergleichbar mit M3 jedoch mit:
  - Digital Signal Processing (DSP)
  - Floating Point Unit (FPU)

#### Cortex-A

- HighEnd Anwendungen und Betriebssysteme
- hohe Rechenleistung
- Chache Memory

#### Cortex-R

- Echtzeitfähigkeit
- hohe Zuverlässigkeit
- System on Chip (SOC)

#### Cortex-M

- Speziell für  $\mu$  C-Markt
- Low Cost, Low Energy
- System on Chip (SOC)

#### 4.1.1 Vorteile der Cortex-M-Prozessoren

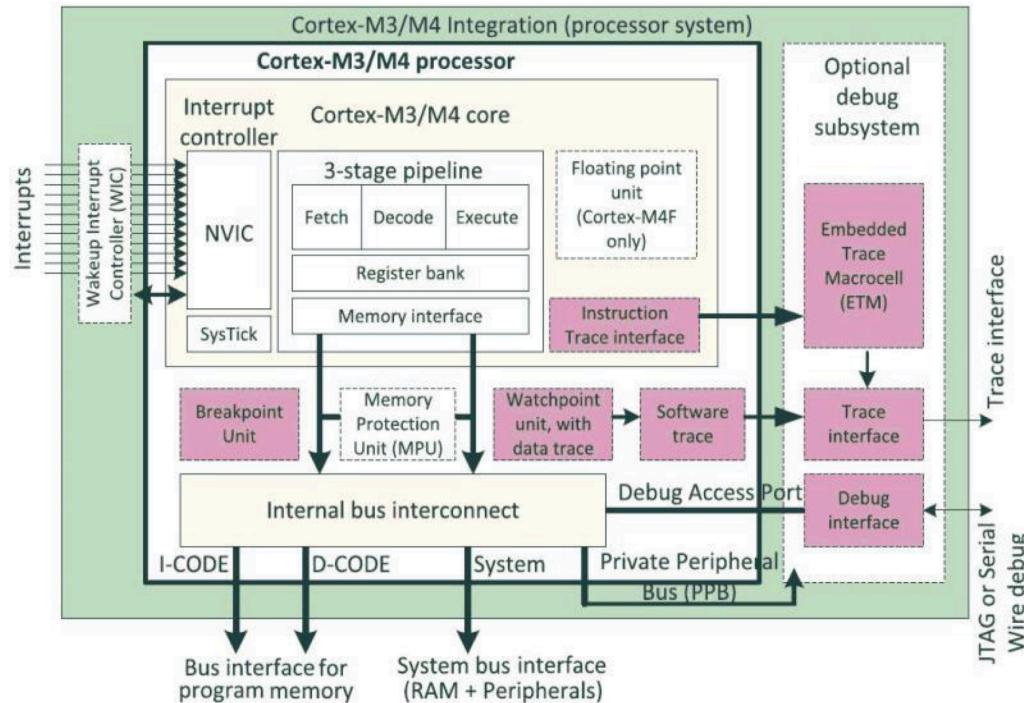
- Low Power  
 $< 200\mu A / MHz$
- Performance  
 $> 1.25 DMIPS / MHz$
- Energy Efficiency  
low Power, high performance
- Code Density  
Thumb 2 Befehlssatz

- Interrupts  
240 Interrupts
- Easy of Use, C Friedly
- Scalability
- Debug Features
- Software portability and Reuseability
- OS Support
- Choices (Derivers, Tools, OS,...)

## 4.2 Cortex-M3/M4

- Harvard Architecture  
→ Zugriffe auf Instruktionen und Daten können gleichzeitig stattfinden
- Internal Bus Interconnect  
→ mehrere Bus-Interface
- Nested Interrupt Controller (**NVIC**)
- Standart Timer (**SYSTICK**)  
**Optional:**
- Memory Protection Unit (**MPU**)
- Floating Point Unit (**FPU**)

## 4.3 System-Komponenten



### 4.3.1 NVIC

- Non-Maskable Interrupt (NMI)
- Bis zu 240 externe Interrupts
- 8 bis 256 Prioritätslevel

→ ISR benötigt 12 Taktzyklen

Siehe auch: Spezielle Eigenschaften des NVIC S30

### 4.3.2 WIC (Wakeup Interrupt Controller)

Für die Umsetzung von Low-Power-Modes.

Dadurch kann 99% der Cortex M3-Prozessoren im Low-Power-Bereich arbeiten.

Ist mit dem NVIC verknüpft und holt den Prozessor aus diesem Modus heraus, um auf einen Interrupt reagieren zu können

### 4.3.5 SYSTICK

- 24-Bit Countdown-Timer mit automatischer Relaod-Funktion
- Wird für einen periodischen Interrupt verwendet

Wenn der Zähler den Wert 0x000000, wird dies dem NVIC signalisiert und der Reload-Wert wird aus dem Reload-Register gelesen.

### 4.3.3 FPU - (nur Cortex M4!)

Mit der FPU lassen sich IEEE754 Signal Precision Floating-Point Operationen in sehr wenigen Takten ausführen

### 4.3.4 MPU

- ermöglicht Zugriffsregel für den Privileged Access und User Programm Access zu definieren
- → Wird eine Zugriffsrege verletzt erfolgt eine Exception-Regelung wodurch der Exception Handler das Problem analysiert und ggf. beheben kann
- → Ausserdem ist es möglich gewisse Bereiche als read-only zu deklarieren

## 4.4 GNU-Tool-Chain Entwicklungsablauf

<b>APSR</b>	Application Program Status Register
<b>IPSR</b>	Interrupt Program Status Register
<b>EPSR</b>	Execution Program Status Register

### 4.4.1 SP-zugriffe (Assembler)

## 4.5 Programm Status Register

<b>N</b>	Negativ
<b>Z</b>	Zero
<b>C</b>	Carry/borrow
<b>V</b>	Overflow
<b>Q</b>	Sticky saturation flag
<b>ICI/IT</b>	Interrupt-Continauble Instruction(ICE) bits IF-THEN instruction status bit
<b>T</b>	Thumb state, always 1; Trying to clear this bit will caus a fault exception
<b>Exception number</b>	Indicates which exception the processor is handling

### 4.5.1 Q-Flag

This flag is set to 1 if any of the following occurs:

- Saturation of the addition result in a *QADD or QDADD* instruction
- Saturation of the subtraction result in a *QSUB or QDSUB* instruction
- Saturation of the doubling intermediate result in a *QDADD or QDSUB* instruction
- Signed overflow during an *SMLA<x><y> or SMLAW<y>* instruction

The Q flag is sticky in that once it has been set to 1, it is not affected by whether subsequent calculations saturate and/or overflow.

**An example:**

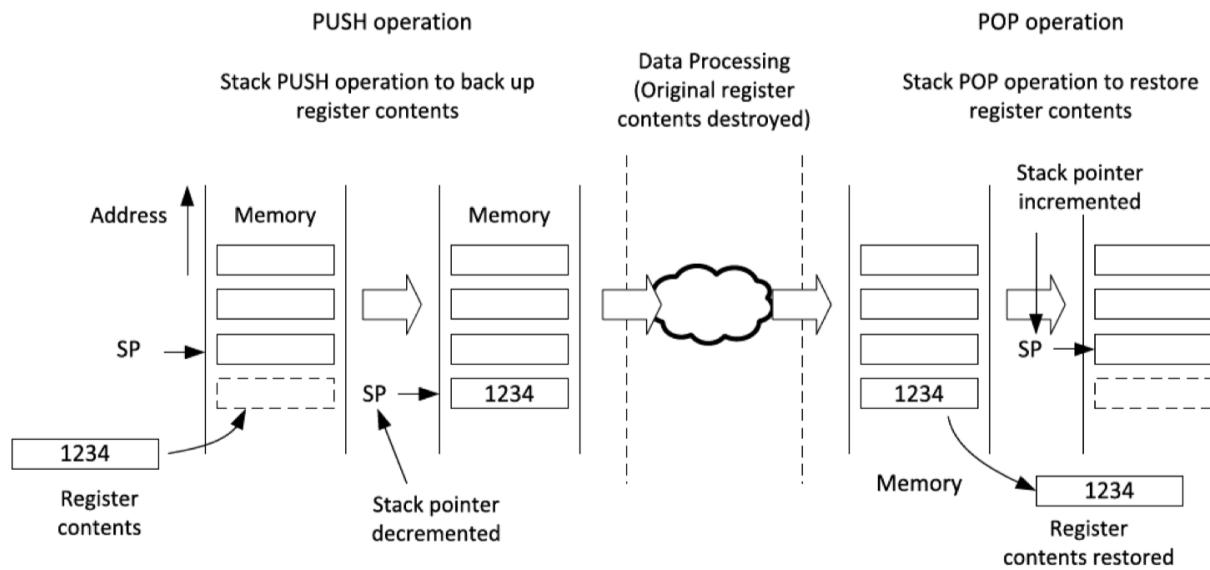
$0x70000000 + 0x70000000$  would become  $0xE0000000$ , but since qadd is saturating, the result is saturated to  $0x7FFFFFFF$  (the largest positive 32-bit integer) and the Q flag is set.

## 4.6 Stack

### 4.6.1 Eigenschaften

- Temporäre Zwischenspeicherung von Daten während der ausführung einer Funktion
- Übergabe von Informationen an Funktionen oder Subroutinen
- Speichern von lokalen Variablen
- Erhalten von Prozessor-Status und Register-Werten, während Exceptions oder Interrupts ausgeführt werden
- PUSH-POP-Instruktionen werden ausgeführt
- LIFO-Prinzip(Last In, First Out)

### 4.6.4 PUSH-POP Operationen



Der Stack wird von der höchsten Adresse aus gefüllt, das heisst das bei einer PUSH-Operation der Stack-Pointer dekrementiert wird. Es gibt auch kombinierte PUSH/POP Operationen. Die wohl wichtigste ist die POP/Return Operation, bei welcher das Link-Register (LR) beim PUSH auf den Stack gelegt wird und bei der POP-Operation direkt in den Programm-Counter (PC) geschrieben wird. Da alle Register in der Registerbank 32-Bit breit sind, wird der Stack-Pointer immer eine *4-Byte aligned Adresse* enthalten.

## 5 V5

### 5.1 Data Alignment

In der Cortex-M Nomenklatur werden direkt aufeinanderfolgende Bytes wie folgt bezeichnet:

<b>2 Byte</b>	<b>16-Bit Half-World</b>	Halbwort
<b>4 Byte</b>	<b>32-Bit Word</b>	Wort
<b>8 Byte</b>	<b>64-Bit Double-Word</b>	Doppelwort

Pro Operation (read oder write) können auf dem internen 32-Bit breiten Datenbuses 4 Byte gemeinsam übertragen werden. Sehr oft sind aber kleinere Dateneinheiten (Half-Word oder Byte) zu übertragen. Zudem kommt es recht häufig vor, dass die Daten nicht auf die 32-Bit Wortgrenze ausgerichtet sind, also als **misaligned data** im Speicher abgelegt sind. Misaligned Half-Word, Word oder Double-Word Operanden im Speicher benötigen somit mehr als einen *Memory-Cycle*. Wenn man zum Beispiel einen Operanden von der Adresse 0x102 lesen will, wird im ersten *Memory-Cycle* ein Word von der Adresse 0x100 gelesen und dann im zweiten *Memory-Cycle* wird ein Word von der Adresse 0x104 gelesen. Der Operand wird dann aus den beiden gelesenen Word zusammengesetzt.

#### 5.1.1 Aligned Data

Operanden können mit einem *Memory-Cycle* ausgelesen/geschrieben werden.

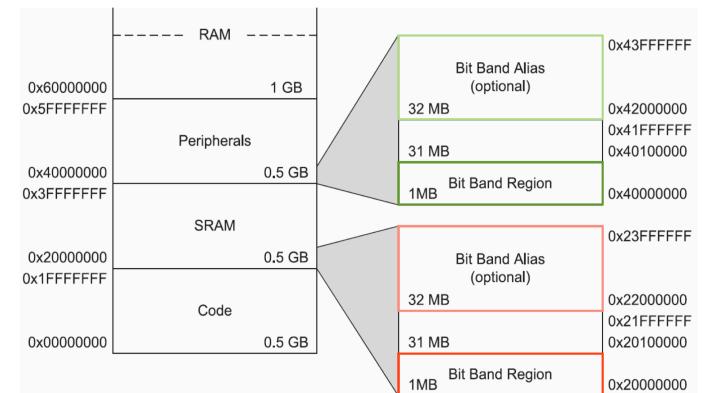
#### 5.1.2 Unaligned Data

Operanden müssen aus mehreren *Memory-Cycle* zusammengesetzt werden.

## 5.2 Bit-Banding

Bit-Banding verwendet zwei verschiedene Regionen im Adressraum, um effektiv auf dieselben physischen Daten zu verweisen:

- Im *Bit-Band Region* Speicherbereich referenziert jede Adresse genau ein einzelnes Byte an Daten, bestehend aus 8 Bits
- Im *Bit-Band Alias* Speicherbereich wird mit jeder Adresse nur ein einzelnes Bit in der primären *Bit-Band Region* angesprochen.



### 5.2.1 Berechnung der Adresse

$$\text{BBAA} = \text{BBAB} + (\text{MA} - \text{BBRB}) \cdot 2^5 + 4 \cdot \text{BNr}$$

$$\text{BNr} = [0x0000001F \& (\text{BBAA} - \text{BBAB})] \cdot 2^{-2}$$

$$\text{MA} = (\text{BBAA} - \text{BBAB}) \cdot 2^{-5} + \text{BBRB}$$

Bei der zurückrechnung ist **MA** die Byte-Adresse und **BNr** immer zwischen 0-7. Wenn nun die aligned Half-Word oder aligned Word Adresse gefragt ist muss die Adresse und die Bitnummer noch dementsprechend angepasst werden.

Legende:

BBAA	Bit-Band Alias Address
BBAB	Bit-Band Alias Base
MA	Memory-Address
BBRB	Bit-Band Region Base
BNr	Bit-Number

## 6 V6

### 6.1 Exceptions and Interrupts

Exceptions sind Ereignisse, die den sequenziellen Programmablauf verändern. Der Prozessor unterbricht den normal laufenden Programmablauf (Background) und führt einen Exception-Handler (Foreground) aus. Exceptions werden vom NVIC (nested vectored interrupt controller) verarbeitet. Der NVIC kann eine Reihe von *Interrupt Request (IRQs)* und einen *Non-Maskable Interrupt (NMI)* verarbeiten, wobei der *NMI* beispielsweise von einem Watchdog-Timer aktiviert werden kann.

Der Prozessor selbst ist auch eine Quelle von Exceptions.

Jede Exception-Quelle hat eine zugehörige Exception-Number:

- Exception-Number 1-15 gelten als *System-Exceptions*
- Exception-Number 16-255 sind *Interrupts*

Der NVIC kann bis zu 240 IRQs verarbeiten, in der Praxis sind es aber oft weniger.

Exception-Handler für Interrupts werden als Interrupt Service Routine (ISR) bezeichnet, wobei die Interrupt-Latenzzeit (*Interrupt Latency*) gerade mal 12 Clockzyklen beträgt.

#### 6.1.1 NVIC

Jeder Interrupt kann individuell aktiviert/deaktiviert werden, außerdem kann sein *Pending State* durch die Software gesetzt/gelöscht werden. Den Exceptions können *Priority-Level* zugeordnet werden und somit ausgelöste ISRs mit tieferer Priorität durch einen Interrupt höherer Priorität unterbrochen werden.

Es gibt 8 verschiedene Prioritätsstufen, wobei diese noch in Prioritätsgruppen aufgeteilt werden können. Daraus entstehen dann *Preemptive levels* und *Subpriority levels*. Wenn nun zum Beispiel zwei Interrupts das gleiche *Preemptive level* besitzen, kann mit dem *Subpriority level* bestimmt werden, welcher Interrupt beim gleichzeitigen Auftreten zuerst ausgeführt wird. Ein Interrupt mit einer höheren *Subpriority* kann jedoch keinen einer tiefen *Subpriority* unterbrechen, wenn sie das gleiche *Preemptive level* besitzen.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Preemptive priority	Sub prio.						not used

Priority Value

### 6.2 Reset und Reset-Sequenzen

#### 6.2.1 Reset

Es gibt 3 Arten von Reset:

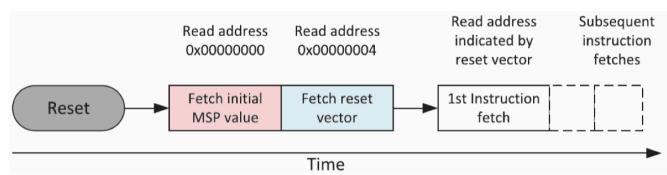
**Power-on Reset** Resettet den gesamten  $\mu$  C, auch alle Peripherien und Debug-Komponenten

**System Reset** Resettet nur den Prozessor und die Peripherien, aber nicht die Debug-Komponenten

**Processoer Reset** Resettet nur den Prozessor

#### 6.2.2 Reset Sequenz

Nach einem Reset und bevor der Cortex-M Prozessor mit der eigentlichen Programmausführung startet, liest die CPU die ersten beiden 32-Bit Word aus dem Speicher. Am Anfang in der Vektor-Tabelle steht der **Initial Main-Stack-Pointer (MSP)** gefolgt vom **Initial Program Counter (PC)**. Das Setup des MSP ist notwendig, um von Beginn weg einen gültigen Stack zu haben.



## 6.3 Fault-Handling

Der Fault-Exception Mechanismus erlaubt eine schnelle Reaktion auf Systemfehler und gibt der Software die Möglichkeit, Notfallszenarien einzuleiten. Standardmäßig sind die Exceptions *Bus Fault*, *Usage Fault* und *Memory Management Fault* deaktiviert, stattdessen triggern alle drei Faults die *Hard Fault Exception*.

## 6.4 Spezial-Register

Register um Exceptions ein oder auszuschalten:

→ **PRIMASK**, **FAULTMASK**, **BASEPRI**

### 6.4.1 PRIMASK

- 1-bit Register
- Wenn das aktiv ist, werden NMI-Interrupts erlaubt
  - alle anderen Interrupts werden überdeckt
  - Default-Wert = 0, also deaktiviert

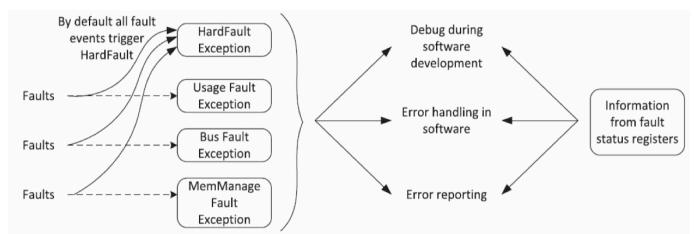
### 6.4.3 BASEPRI

- Wenn das gesetzt wird, werden alle Interrupts mit gleicher oder tieferer Stufe deaktiviert

### 6.4.4 Control-Register

Das Kontroll-Register definiert:

1. Die Auswahl zwischen MSP (Main-SP) und PSP (Process-SP)
2. Die Zugriffsstufe und Thread-Mode  
(Ob Privileged oder unprivileged)



### 6.4.2 FAULTMASK

- 1-bit Register
- Wenn das aktiv ist, werden nur noch NMI-Interrupts akzeptiert. Alle anderen Interrupts und Exception-Handlings werden deaktiviert
  - Default-Wert = 0

## 7 V7 (ausgefallen)

## 8 V8

### 8.1 Cortex M3 Instruction Set

#### 8.1.1 Thumb-2 Instruction Set

Ziel:  
 -Erhöht die Code-Dichte  
 -Mehr Leistung

Cortex M3 Processor: 1.25 DMIPS / MHz

### 8.2 Instruction Pipelining

Mit Pipelining bearbeitet der Cortex-M Prozessor zur gleichen Zeit drei Befehle - jeweils um einen Takt versetzt.

Fetch: 16-Bit Instruktion aus Programmspeicher laden  
 Decode: Vorangehende Instruktion decodieren  
 Execute: Nochmals vorangehende Instruktion ausführen

Bei einem Sprung im Programmcode muss eine Pipeline komplett geleert werden, da die sequentielle Reihenfolge je nach Konstellation nicht korrekt ist.

### 8.4 Anwendungen

#### 8.4.1 Cortex-M0/M0+ / M1

- einfaches I/O Handling

#### 8.4.2 Cortex-M3

- Komplexe Datenverarbeitung  
 - anspruchsvolle Applikationen

#### 8.4.3 Cortex-M4

- DSP-Funktionalität  
 - Floating Point Support

### 8.3 Logikstruktur des Cortex-M Prozessor

Sourceoperanden: Rn, Rm

Destinationsoperand: Rd

MAC: Memory Access Calculator

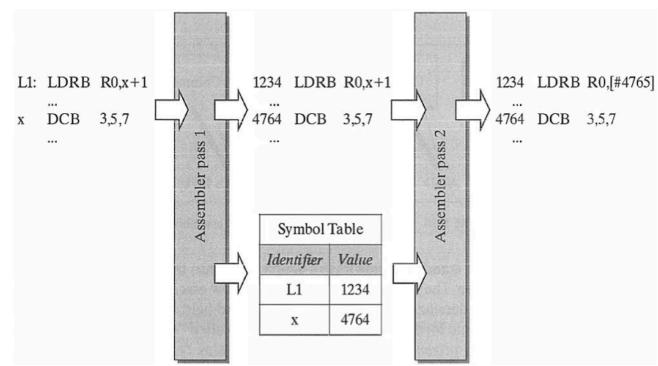
Ein *Barrel-Shifter* vereinfacht Berechnungen, da Multiplikationen einfacher realisiert werden können.

Das 32-Bit breite Programm-Status-Register (xPSR) enthält die Flags. Das xPSR gliedert sich in drei verschiedenen Darstellungen, je nach aktuellem Prozessorstatus als: *Application-Program Register (APSR)*, *Interrupt-Program Status Register (IPSR)* oder *Execution-Program Status Register (EPSR)*.

Die wichtigsten Flags (N, Z, C, V) sind die vier höchsten Bits des APSR, welche die Basis für bedingte Verzweigungen darstellen.

## 8.5 Software Development Prozess

Die Umsetzung von Hochsprachen Source-Code in Assembly-Language und weiter in den binären OpCode des jeweiligen Prozessors ist ein komplexer, mehrstufiger Prozess. Im ersten Durchgang baut der Assembler eine Symboltabelle auf, welche Informationen über sogenannte *programmer-defined Identifiers* enthält (z.B. Adressen von Sprungmarken, Subroutinen, Variablen, I/O-Port Registern, usw.). Während des zweiten Durchgangs benutzt der Assembler diese Informationen, um die einzelnen Assembler-Instruktionen zu vervollständigen. Erst in einem weiteren Teilschritt der Assemblierung werden die binären OpCode ermittelt.



## 8.6 Assembly-Language Syntax

Label	OpCode	Operand	Comment
L1	ADD	R0,R1,#5	Replace R0 by sum of R1 and 5
FUNC	MOV	R0,#100	this sets R0 to value 100
	BX	LR	this is a function return

<b>Label</b>	optional
<b>OpCode</b>	spezifiziert den Befehl
<b>Operand</b>	Parameter
<b>Comment</b>	optionale Beschreibung

## 8.7 Unified Assembler Language (UAL)

Syntax für ARM und Thumb Instructionen.  
Die meisten Instruktionen arbeiten mit Registern  
**BSP**

MOV	R2,#100	;R2=100,Direkte Zuweisung
LDR	R2,[R1]	;R2= den Wert von R1
ADD	R2,R0	;R2=R2+R0
ADD	R2,R0,R	;R2=R0+R1

### 8.7.1 Register List

Norm. Form	reglist	;R1,R2...Rn
PUSH	LR	;save LR on stack
POP	LR	;remove from stack; place in LR
PUSH	R1-R3,LR	;save R1,R2,R3; return address
POP	R1-R3,PC	;restore R1,R2,R3 and return

## 8.8 Addressing

### 8.8.1 Immediate Addressing

Der Datenwert ist unmittelbar in der Instruktion erhalten.  
Daher kein zusätzlicher Speicherzugriff erforderlich  
Form: # imm

MOV R0,# 100 ;R0=100, immediate addressing

### 8.8.2 Indirect Addressing

Bei der *Indirect Addressing Modes* sind die Daten im Memory. Ein Register enthält irgendwie einen Zeiger auf diese Daten. Nach der *Fetch-Phase*, bei welcher die Instruktion aus dem Programmspeicher gelesen wird, sind noch einer oder mehrere Speicherzugriffe erforderlich um die Daten zu lesen oder zu schreiben.

Form: [Rn]

LDR R0,[R1] ;R0=value pointed to by R1

R1 wird nicht verändert

### 8.8.3 Register Addressing with Displacement

Dasselbe, nur wird hier dem Wert R0 noch # 4 hinzugefügt  
R1 bleibt weiterhin unverändert.

Form: [Rn,# imm]

LDR R0,[R1,# 4] ;R0=word pointed to by R1+4

### 8.8.4 Register Indirect with Index

Form: [Rn,Rm]

LDR R0,[R1,R2] ;R0= word pointed to by R1+R2

### 8.8.5 Register Indirect with shifted Index

Form: [Rn,Rm,LSL # imm]

LDR R0,[R1,R2;LSL #2] ;R0= word pointed to by R1+4\*R2

### 8.8.6 Register Indirect with Pre-index

Form: [Rn,# offset]!

LDR R0,[R1,#4]! ;first R1=R1+4, then R0= word pointed to by R1

### 8.8.7 Register Indirect with Post-index

Form: [Rn],# offset

LDR R0,[R1],#4 ;R0= word pointed to by R1, then R1=R1+4

### 8.8.8 PC-relativ

PC wird als Pointer verwendet. Form: lable

B Location ;jump to Location

BL Subroutine ;call Subroutine, Rücksprungadresse wird gespeichert

### 8.8.9 Speicher- und I/O-Zugriffe

Es benötigt immer zwei Instruktionen um auf Daten im RAM oder I/O zuzugreifen. → PC-Relative Addressierung wird verwendet

1. Erstellt Zeiger auf das Objekt
2. Greift über den Zeiger Indirekt auf den Speicher zu

LDR R1,Count ;R1 points to variable Count

LDR R0,[R1] ;R0= value pointed to by R1

## 9 V9

### 9.1 Interne Datenverschiebung

Für die Verschiebung von Daten von Register zu Register oder der Beschreibung eines Registers mit einer Konstanten kann die Operation **MOV** verwendet werden. Diese folgend angewendet werden:

Instructions for Transferring Data within the Processor			
Instruction	Dest	Source	Operation
MOV	R4,	R0	;Copy value from R0 to R4
MOVS	R4,	R0	;Copy value from R0 to R4 w. APSR (flags) update
MRS	R7,	PRIMASK	;Copy value of PRIMASK (special register) to R7
MSR	CONTROL,	R2	;Copy value of R2 into CONTROL (special register)
MOV	R3,	#0x34	;Set R3 value to 0x34
MOVS	R3,	#0x34	;Set R3 value to 0x34 with APSR update
MOVW	R6,	#0x1234	;Set R6 to a 16-bit constant 0x1234
MOVT	R6,	#0x8765	;Set the upper 16-bit of R6 to 0x8765
MVN	R3,	R7	;Move negative value of R7 into R3

### 9.2 Memory Access Instructions

Der Cortex-M3 verfügt über viele verschiedene Instruktionen für den Speicherzugriff. Dies wegen den verschiedenen Adressierungsarten und Datengrößen. Für normale Datentransfers, sind folgenden Instruktionen vorhanden:

Memory Access Instructions for Various Data Sizes		
Data Type	LOAD (Read from Memory)	STORE (Write to Memory)
8-bit unsigned	LDRB	STRB
8-bit signed	LDRSB	STRB
16-bit unsigned	LDRH	STRH
16-bit signed	LDRSH	STRH
32-bit	LDR	STR
Multiple 32-bit	LDM	STM
Double-word (64-bit)	LDRD	STRD
Stack operations (32-bit)	POP	PUSH

Mit der **LDM** bzw. **STM** Instruktion ist es möglich gerade eine ganze Registerliste zu Laden bzw. zu Speichern.

### 9.3 Stack Push und Pop

```
PUSH    R0
PUSH    R1
PUSH    R2
POP     R3
POP     R4
POP     R5
```

**PUSH** und **POP** können auch Registerlisten mitgegeben werden, wie folgendes Beispiel zeigt:

```
PUSH {R4-R6,LR} ;Save R4 to R6 an LinkRegister at the beginning of a subroutine.
                  ; LR contains the return address
...
;processing the subroutine
POP  {R4-R6,PC} ;POP R4 to R6, and return address from stack. The return address
                  ;is stored into PC directly, this triggers a branch (subroutine return)
```

#### 9.3.1 Generelle Regeln bei der Verwendung des Stacks

1. Funktionen sollten die gleiche Anzahl Push und Pop Befehle aufweisen.
2. Stackzugriff nur innerhalb des allozierten Bereichs
3. Es sollte nicht über den SP auf den Stack geschrieben oder gelesen werden.
4. Stack sollte zuerst den SP dekrementieren und erst dann die Daten ablegen.
5. Stack sollte die Daten zuerst lesen und erst dann den SP inkrementieren.

## 9.4 Shift and Rotate Instructions

LSL: Signed, unsigned Multiplikation mit  $2^n$   
 LSR: Unsigned Division mit  $2^n$

## 9.6 Compare and Test

Die **COMPARE** und **TEST** Instruktionen aktualisieren die Flags im *APSR*, welche für den *Conditional Branch* oder *Conditional Execution* benötigt werden.

Instructions for Compare and Test	
Instructions	Description
CMP Rn, Rm	Compare: Calculate Rn-Rm. APSR is updated but the result is not stored.
CMP Rn, #immed	Compare: Calculate Rn-immediate data. APSR is updated but the result is not stored.
CMN Rn, Rm	Compare negative: Calculate Rn+Rm. APSR is updated but the result is not stored.
CMN Rn, #immed	Compare negative: Calculate Rn+immediate data. APSR is updated but the result is not stored.
TST Rn, Rm	Test (bitwise AND): Calculate AND result between Rn and Rm. N bit and Z bit in APSR are updated but the AND result is not stored. C bit can be updated if barrel shifter is used.
TST Rn, #immed	Test (bitwise AND): Calculate AND result between Rn and immediate data. N bit and Z bit in APSR are updated but the AND result is not stored.
TEQ Rn, Rm	Test (bitwise XOR): Calculate XOR result between Rn and Rm. N bit and Z bit in APSR are updated but the XOR result is not stored. C bit can be updated if barrel shifter is used.
TEQ Rn, #immed	Test (bitwise XOR): Calculate XOR result between Rn and immediate data. N bit and Z bit in APSR are updated but the EXOR result is not stored.

## 9.7.3 Conditional Branches

*Conditional Branches* sind bedingte Sprünge, bei welchen nur zur angegeben Adresse gesprungen wird, wenn die Flags die Bedingung erfüllen. Die Flags (*APSR*) werden immer vor einem *Conditional Branch* mit einer **CMP**-Instruktion evaluiert.

Instructions for Conditional Branch	
Instructions	Description
B{cond} label	Branch to label if condition is true. E.g., CMP R0,#1 BEQ loop ;Branch to "loop" if R0 equal 1.
B{cond}.W label	If a branch range of more than +/-254 Bytes is needed, B.W can be used as a 32-bit version of branch instruction for wider range.

Ein Beispiel:

```
CMP R0,#1 ;compare R0 to 1
BEQ p2 ;if Equal, then go to p2
MOV R3,#1 ;R3=1
B p3 ;go to p3
p2: ;label p2
p3: ;label p3
... ;other subsequence operation
```

## 9.5 Bit-Field Processing Instructions

Der Cortex-M3/M4 Prozessor verfügt über viele verschiedene *Bit-Field Processing Operations*-Instruktionen, einzelne sind folgend aufgelistet:

Instructions for Bit-Field Processing	
Instructions	Description
CLZ Rd, Rm	Count leading zero
RBIT Rd, Rn	Reverse bit order in register
BFC Rd, Rn, #lsb, #width	Clear bit field within a register
BFI Rd, Rn, #lsb, #width	Insert bit field to a register
SBFX Rd, Rn, #lsb, #width	Copy bit field from source and sign extend it
UBFX Rd, Rn, #lsb, #width	Copy bit field from source register

## 9.7 Program Flow Control

### 9.7.1 Unconditional Branches

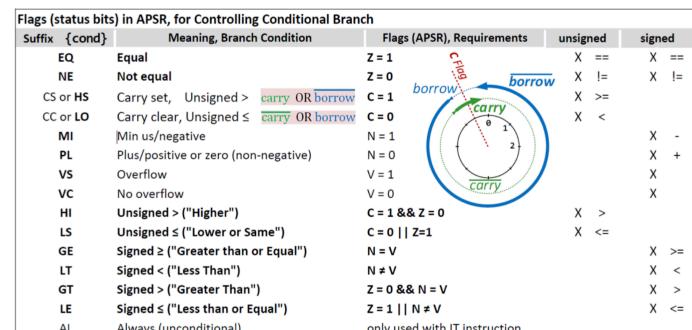
*Unconditional Branches* sind Sprünge zu einem Label, welche wie folgt implementiert werden können.

Unconditional Branch Instructions	
Instructions	Description
B label	Branch to label. If a branch range of over +/- 2KB is needed, you might need to specify B.W to use 32-bit version of branch instruction for wider range.
BX Rm	Branch and exchange. Branch to an address value stored in Rm, and set the execution state of the processor (T-bit) based on bit 0 of Rm. <b>(Bit 0 of Rm must be 1 because Cortex-M processor only supports Thumb state.)</b>

### 9.7.2 Function Calls

Für den Funktionsaufruf wird die Instruktion *Branch and Link* verwendet.

Instructions for Calling a Function	
Instructions	Description
BL label	Branch to a labeled address and save the return address in LR.
BLX Rm	Branch to an address specified by Rm, save the return address in LR, and update T-bit in EPSR with LSB of Rm.



## 10 V10

### 10.1 C/C++ Strukturen Umsetzen

#### 10.1.1 Entscheidungen

In Assembler-Sprache ist eine Entscheidung praktisch immer in einem 2-Stufigen Ablauf umgesetzt.

- Benötigte Flags ermitteln
- Zugehörige bedingte Sprünge ausführen

Dabei wird nach folgendem Ablauf gearbeitet:

1. Vergleich: Zwei Werte werden subtrahiert, dabei wird nur auf die Flags geschaut.
2. Anhand der Flags werden dann die bedingten Sprünge ausgeführt

#### 10.1.2 Analyse von Hochsprachencode und Assembly-Code

Für die Bestimmung der richtigen *Conditional Branch* Instruktion kann wie am folgendem Beispiel gezeigt werden vorgegangen werden:

if (...)	
then	else
foo_one();	foo_two();

Thumb-2 Assembly Code	C Source Code
<pre> LDR    R0, G2 LDR    R1, G1 LDRSH.W R0, [R0, #0] LDR    R1, [R1] CMP    R0, R1 ;G2-G1 <b>BGE</b>  else_if <b>then_if:</b>     BL    foo_one     B     end_if <b>else_if:</b>     BL    foo_two <b>end_if:</b> </pre>	<pre> signed Long G1; //signed 32-bit signed short G2; //signed 16-bit ... if (G1 &gt; G2)      //0 &gt; G2-G1 {     foo_one(); } else {     foo_two(); } </pre>

1. Bestimmung des Datenformates der Variablen: *signed* oder *unsigned*  
⇒ Beispiel: signed
2. Gleichung für den *Conditional Branch* aufstellen: **CMP Rn, Rm** entspricht Flags für  $Rn - Rm$   
⇒ Beispiel:  $G1 > G2 \Leftrightarrow 0 > G2 - G1$

#### if-Block

Für einen Sprung in den if-Block wäre nun der Operator  $\textcircled{>}$  entscheidend.  
⇒ Conditional Branch Instruction: **BLT**

Im Beispiel folgt jedoch der if-Block nach dem else-Sprung.

#### 10.1.3 Beispiele

##### IT-Instruktion

Thumb-2 Assembly Code	C Source Code
<pre> ;A=R0, B=R1, C=R2 ... <b>CMP</b>  R0, #0x0    ;A=0 <b>ITTEE</b> GT          ;4 instr. <b>ADDGT</b> R1, #0x1    ;B += 1 <b>MOVGT</b> R2, #0x0    ;C = 0 <b>ADDLE</b> R2, R1      ;C += B <b>MOVLE</b> R1, #0x64   ;B = 100 </pre>	<pre> register signed long A,B,C; ... <b>if (A &gt; 0)</b> {     B += 1;     C = 0; } <b>else</b> {     C += B;     B = 100; } </pre>

#### else-if-Block

Die Gleichung wird zusätzlich negiert:

$$0 > G2 - G1 \Rightarrow 0 \leq G2 - G1$$

Für einen Sprung in den else-if-Block wäre nun der Operator  $\textcircled{\leq}$  entscheidend.  
⇒ Conditional Branch Instruction: **BGE**

⇒ Conditional Branch Instruction: **BGE**

Wenn eine if..then..else Struktur nur wenige Sequenzen von Anweisungen enthält und keine *Branches* kann dies auch mit der *IT-Instruktion* umgesetzt werden, wobei diese maximal vier nachfolgende Anweisungen zu kontrollieren vermag.

Die erste Instruktion im *IT-Block* ist aktiviert, wenn der Condition-Code im Operandenfeld erfüllt ist. Die weiteren Anweisungen im Block werden durch anhängen von eins bis drei Buchstaben mnemonisch gesteuert: T ⇒ then; E ⇒ else

Allen Anweisungen im *IT-Block* muss der entsprechende Condition-Code angehängt werden. Es werden **immer** gleich viele Taktzyklen ausgeführt, egal welcher Pfad gefahren wird.

## FOR-Schleife

Thumb-2 Assembly Code	C Source Code
<pre>;XYZ=R0 MOV R0, #0x0 CMP R0, #0x5 BHS end_for start_for: BL foo_one ... ADD R0, R0, #0x1 ;YXZ++ CMP R0, #0x5 ;check loop BLO start_for ;loop again end_for: BL foo_two ... ;XYZ=R0 MOVS R0, #0xA ;check loop BEQ end_for ;endup loop start_for: BL foo_one ... SUBS R0, R0, #0x1 BNE start_for end_for: BL foo_two ...</pre>	<pre>register uint32_t XYZ; ... for (XYZ=0; XYZ&lt;5; XYZ++) {     foo_one();     ...     //BLO ≡ BCC     foo_two();     ... }</pre>
	<pre>register uint32_t XYZ; ... for (XYZ=10; XYZ!=0; XYZ--) {     foo_one();     ...     foo_two();     ... }</pre>

Eine universelle Schleife (Iteration) besteht im Allgemeinen aus vier Teilen:

- Initialisierung
- Test auf Abbruch oder Fortsetzung
- Update der Loop-Control Variable
- Loop-Body (zu wiederholender Programmcode)

Bei FOR-Schleifen, welche die Laufvariable gegen Null vergleichen, können so manche Instruktionen erspart werden. So können statt der CMP-Instruktion die **MOVS-/SUBS-Instruktionen** verwendet werden, welche gerade das *Zero-Flag* evaluieren. Solange die Laufvariable nicht im Body verwendet wird, kann jede FOR-Schleife damit optimiert werden.

11 V11

## 11.1 Subroutinen

## 11.2 Architecture Producer Call Standard (AAPCS)

### 11.2.1 Regeln

- Bei Funktionsaufrufen werden die Register **R0-R3** als **Parameter** an eine C-Funktion verwendet
  - Werden mehr als vier Funktionsparameter benötigt, werden diese vom Caller auf den Stack gelegt und auch wiedervom Caller entfernt.
  - Die Funktionen müssen die Inhalte der Register **R4-R11**(falls benutzt) während der Ausführung sichern, um sie am Ende wieder rekonstruieren
  - Der **Rückgabewert** einer Subroutine (8-bit, 16-bit,32-bit) wird in den **Registern R0** übertragen. Handelt es sich um einen 64-bit Rückgabewert, so sind die unteren 32-bit im Register R0 und die oberen 32-bit im Register R1 übertragen
  - Mit PUSH und POP wird immer eine **gerade Anzahl von Registern auf dem Stack** gelegt bzw. vom Stack eine **8-byte Alignment** auf dem Stack einzuhalten

## 11.2.2 Beispiel

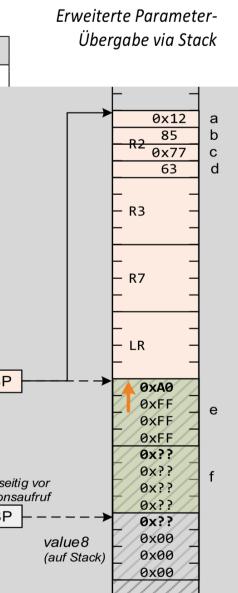
Die Parameterübergabe bei Funktionen soll mit dem nachfolgenden Beispiel erklärt werden.

Die Funktion besitzt einen unsigned 8-Bit Rückgabewert und sechs signed 8-Bit Parameter.

( $\Rightarrow$  2 Parameter werden über den Stack übergeben)

1. **Stackpointer SP (R13)**  
wird um 8 nach unten verschoben
  2. Variablen  $e$  und  $f$  werden vom Caller auf den Stack gelegt.
  3.  $R0 - R3$  werden Parameter zugewiesen
  4. Sprung zum Label  $foo\_4$   
 $LR \Leftarrow retAdr$
  5.  $R2, R3, R7, LR$  werden auf den Stack gelegt ( $SP$  wird um 16 nach unten verschoben)
  6.  $R7$  wird für Zugriff auf  $e$  und  $f$  als Framepointer gesetzt.
  7.  $R0 - R3$  werden auf den Stack gelegt

Thumb-2 Assembly Code	C Source Code
uint8_t foo_4 (int8_t a, int8_t b, int8_t c, int8_t d, int8_t e, int8_t f);	
Thumb-2 Assembly Code	C Source Code
foo_4:	
⑤ PUSH {R2, R3, R7, LR}	uint8_t foo_4 (int8_t a, int8_t b, int8_t c, int8_t d, int8_t e, int8_t f)
⑥ ADD R7, SP, #0x10	
⑦ STRB.W R3, [R13, #3]	{
STRB.W R2, [R13, #2]	return (a ^ e ^ f);
STRB.W R1, [R13, #1]	}
STRB.W R0, [R13, #0]	
↑ Momentaufnahme Stack	
LDRSB.W R0, [R7, #0]	
LDRSB.W R1, [R7, #4]	
LDRSB.W R2, [R13, #0]	
EOR R0, R2	
EOR R1, R0	
UXTB R0, R1	
POP {R2, R3, R7, PC}	
;	
value8 auf dem Stack	
...	
① SUB SP, #8	
② MVN.W R0, #95	
STR R0, [SP]	
LDRSB.W R0, [R13, #8]	uint8_t value8;
③ MOV R1, #0x55	...
MOV R2, #0x77	value8 = foo_4(0x12, 85,
MOV R3, #0x3F	0x77, 63,
↓ STR R0, [SP, #0x4]	0xA0, value8);
MOV R0, #0x12	...
④ BL foo_4	
STRB.W R0, [R13, #8]	
ADD SP, #8	
...	



## 12 V12, Serielle Schnittstelle

### 12.1 Grundlagen

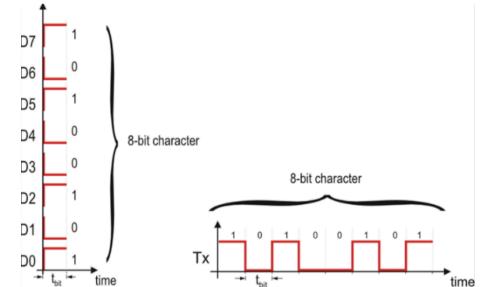
#### 12.1.1 Serielle vs. Parallele Datenübertragung

Bei einem seriellen Kommunikationskanal werden die einzelnen Datenbits sequenziell hintereinander über eine Signalleitung übertragen. Im Gegensatz dazu können bei einem parallelen Kommunikationskanal mehrere Datenbits auf einmal transferiert werden. Sowohl bei der seriellen wie auch bei der parallelen Übertragung ist dabei immer eine bestimmte Bit-Übertragungszeit  $t_{bit}$  einzuhalten.

Bei seriellen Kommunikationskanälen werden allgemein zwei unterschiedliche Metriken unterschieden, die *Bit Rate* und die *Baud Rate*.

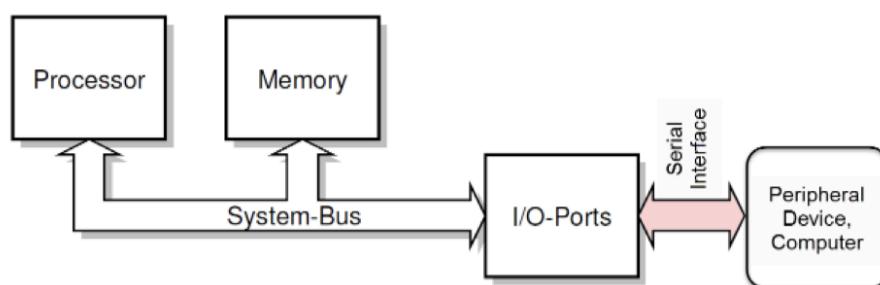
Die *Bit Rate* sagt aus, wie viele *bits-per-second* oder *bts* über den Datenkanal transferiert werden können.

Die *Baud Rate* sagt aus, wie viele sogennante *Symbols* sich in einer Sekunde über den Datenkanal transferieren lassen. Ist jedes Symbol durch ein Bit repräsentiert spricht man von einer *Dualen Codierung* ( $\text{Bit/s} = \text{bps} \equiv \text{Baud}$ )



#### 12.1.2 Parallel-Seriell Umsetzung

Der Mikroprozessor verarbeitet die Daten intern in aller Regel parallel. Für die serielle Datenübertragung ist daher eine koordinierte Parallel-Serien-Wandlung in eine dafür vorgesehenen I/O-Port erforderlich.



#### 12.1.3 Aufbau von Datenkanälen

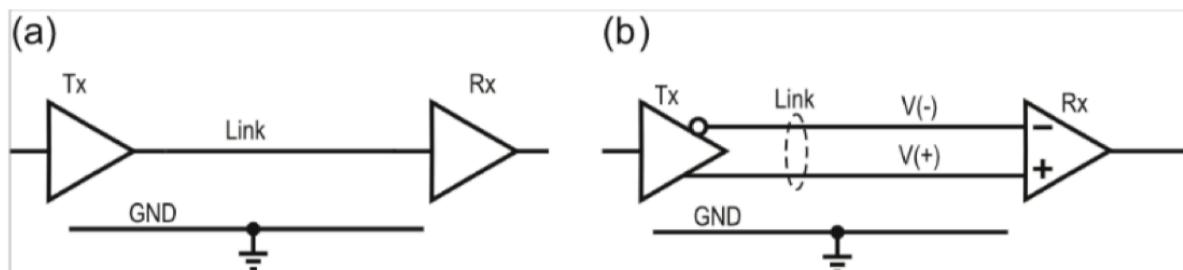
Kommunikationskanäle können auf verschiedene Arten implementiert werden. So können drahtgebundene Kanäle beispielsweise als *Single-Ended* oder als *Differential Link* umgesetzt werden.

##### Single Ended (a)

Bei einer Single-Ended Verbindung werden einzelne, getrennte Signalleitungen benötigt, sowie eine Referenzleitung mit Ground-Potenzial

##### Differential Link (b)

Bei einer Differential Verbindung wird der Datenlink durch eine Differenzspannung zwischen zwei zusammengehörenden Signalleitungen  $V(+)$  und  $V(-)$  repräsentiert. Dies ist deutlich robuster als Single-Ended. Für eine serielle Differential Datenschnittstelle genügen insgesamt drei Leitungen.

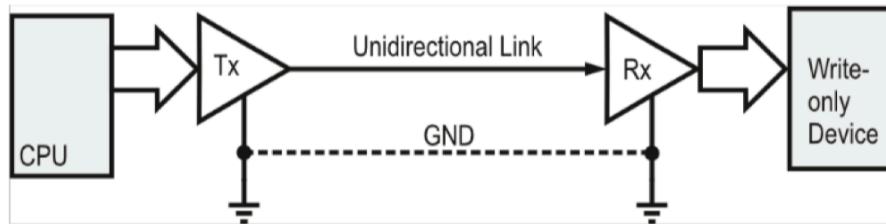


### 12.1.4 Simplex, Half- und Full-Duplex

Datenkanäle werden mit *Simplex*, *Half-Duplex* oder *Full-Duplex* bezeichnet, je nach Art der Konektivität.

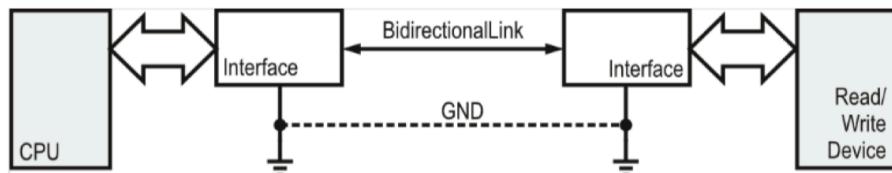
#### Simplex

Ein Simplex serieller Kanal überträgt permanent in nur einer Richtung, über eine dedizierte Verbindung. An einem Ende des Kommunikationskanals arbeitet ein Transmitter, während auf der anderen Seite ein Receiver steht. Simplex Kanäle haben keine Möglichkeit, den Empfang der Daten zu quittieren.



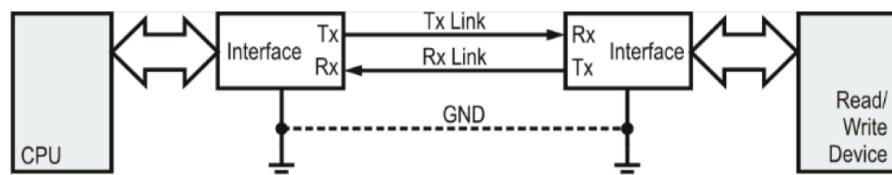
#### Half-Duplex

Ein Half-Duplex serieller Kanal verfügt ebenfalls nur über eine einzige Verbindung. Diese erlaubt jedoch eine beidseitige Kommunikation, aber nur in einer Richtung zur gleichen Zeit. Auf beiden Seiten des Kommunikationskanals steht ein serieller Transceiver, der sowohl als Sender wie auch als Empfänger arbeiten kann. Wird die Übertragungsrichtung geändert, so müssen die beiden Transceiver ihre Betriebsart wechseln. Das Umschalten der Datenrichtung erfordert klare Regeln auf beiden Seiten, um beispielsweise ein gleichzeitiges Senden zu vermeiden.



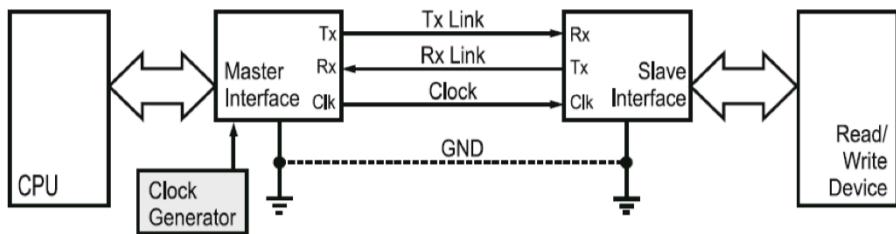
#### Full-Duplex

Full-Duplex serielle Kanäle verfügen über zwei separate Datenlinks; einer um Daten zu senden und ein anderer um Daten zu empfangen. Damit ist eine gleichzeitige Kommunikation in beide Richtungen möglich. Auf beiden Seiten des Kommunikationskanals steht ein serieller Transceiver, der zeitgleich als Sender und Empfänger arbeitet.



## 12.2 Synchrone Datenübertragung

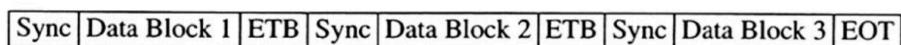
Synchrone serielle Kanäle werden dadurch gekennzeichnet, dass Sender und Empfänger auf das gleiche Clock-Signal synchronisiert sind  $\Rightarrow$  zusätzliche Clock-Leitung. Synchrone Kanäle arbeiten üblicherweise in einer *Master/Slave-Beziehung*.



### 12.2.1 Datenübertragung

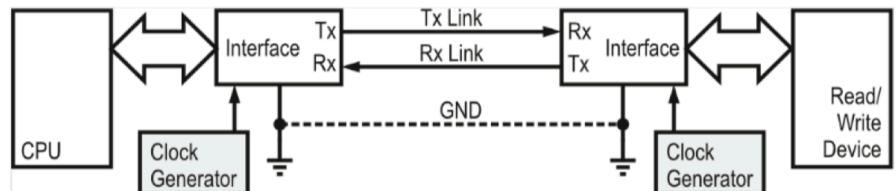
Beim der synchronen seriellen Datenkanal werden die Daten typischerweise in Blöcken mit variabler Länge übertragen. Parallel dazu erfolgt die Übertragung der Clock Information. Dabei ist wichtig, dass Sender und Empfänger auf die gleiche Flanke Daten übernehmen.

Im folgenden Bild ist exemplarisch eine ganze Message mit drei Data Packets (Datagrams) dargestellt: Das Synchronisationszeichen (Sync) kennzeichnet als Header den Beginn eines zusammengehörenden Data Pakets. Grundsätzlich wird nach jedem Datenblock (Body) das Zeichen ETB (End of Transmission Block) als Footer übertragen. Das Ende der gesamten Message wird durch ein EOT (End of Transmission) gekennzeichnet. Das ETB kann vor einem EOT je nach Protokoll entfallen.

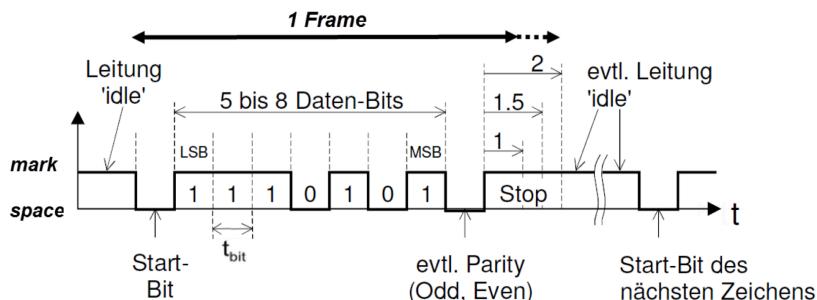


## 12.3 Asynchrone Datenübertragung

Beim asynchronen seriellen Datenkanal laufen auf der Sender- und Empfängerseite zwei unabhängige Clock-Generatoren. Dabei müssen diese auf die selbe Baud-Rate eingestellt werden.



Auf die fallende Datenflanke des Datenframes wird der jeweilige Clock-Generator synchronisiert. Dabei haben Datenpakete folgenden Aufbau:



Datenframe einer UART Schnittstelle (7-Bit, Odd Parity)

Aus den Informationen der Kommunikationsschnittstelle lässt sich zudem die Nutzdatenrate bestimmen.

$$\text{Nutzdatenrate} = \frac{\#Databit \cdot \text{Baudrate}}{\#Startbit + \#Databit + \#Paritybit + \#Stopbit} \cdot \frac{1\text{Byte}}{8\text{Bit}}$$

Die beteiligten Kommunikationsschnittstellen benötigen folgende Informationen um ihre Interfaces auf den asynchronen seriellen Bitstrom abzustimmen:

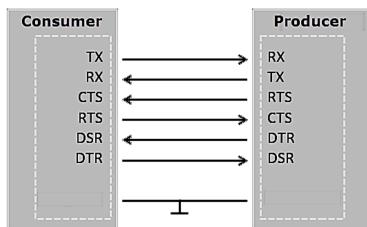
- Baudrate =  $\frac{1}{t_{bit}}$
- Anzahl Daten-Bits (5 bis 8)
- Parity Information (Even / Odd / none) Das Paritybit ergänzt immer auf *Odd* oder *Even*.
- Anzahl Stop-Bit (1 / 1.5 / 2)

### 12.3.1 Datenflusssteuerung

Bei der seriellen Kommunikation muss oft der Datenfluss gesteuert werden. Dies ist zum Beispiel dann dringend notwendig, wenn empfängerseitig der Datenpuffer voll wird und ein Datenüberlauf droht. Dieser Zustand muss an den Datensender signalisiert werden.  $\Rightarrow$  HW- und SW-Handshake

#### Hardware Handshaking

Mit Hardware-Signalen teilen sich die Kommunikationspartner gegenseitig mit, ob sie bereit sind weitere Daten aufzunehmen oder nicht.

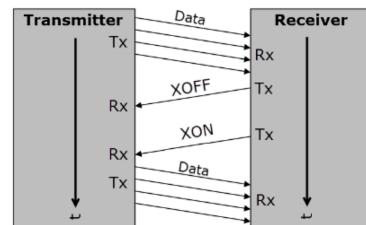


**RTS (request to send), CTS (clear to send)**

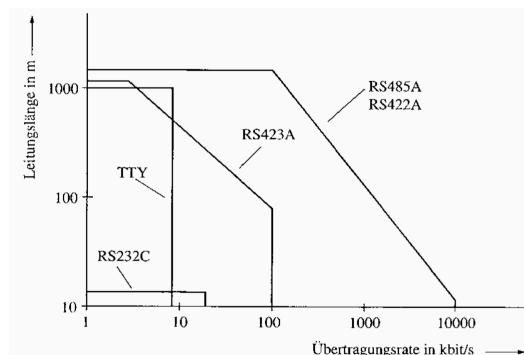
#### Software Handshaking

Beim Software Handshaking kommen anstelle von zusätzlichen Handshake-Leitungen zwei extra für diesen Zweck reservierte ASCII-Zeichen zur Anwendung:

- XON ASCII DC1 0x11
- XOFF ASCII DC3 0x13



### 12.4 Standardisierte serielle Interfaces



	RS-232-C (V.24 / V.28)	RS-423-A (V.10)	RS-422-A (V.11)	RS-485-A
max. Übertragungsrate	20 kbit/s	100 kbit/s	10 Mbit/s	1 Mbit/s
max. Übertragungsweg	15 m	$\geq 1000$ m	$\geq 1000$ m	500 m
Übertragungsverfahren	asymmetrisch	asymmetrisch	symmetrisch	symmetrisch
max. Senderanzahl	1	1	1	32
max. Empfängeranzahl	1	10	10	32
Signalpegel max.	$\pm 15$ V	$\pm 7$ V	$\pm 7$ V	-7 V, +12 V
Signalpegel min.	$\pm 3$ V	$\pm 0,3$ V	$\pm 0,3$ V	$\pm 0,3$ V

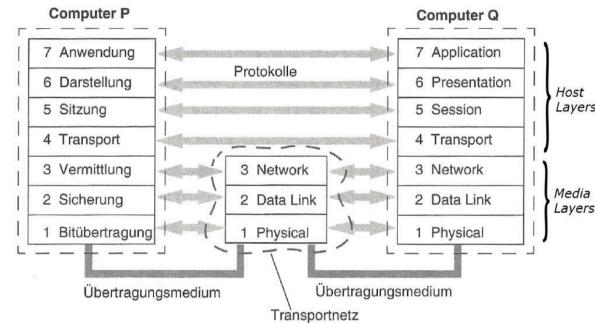
## 13 V13

### 13.1 Kommunikation zwischen Rechnersystemen

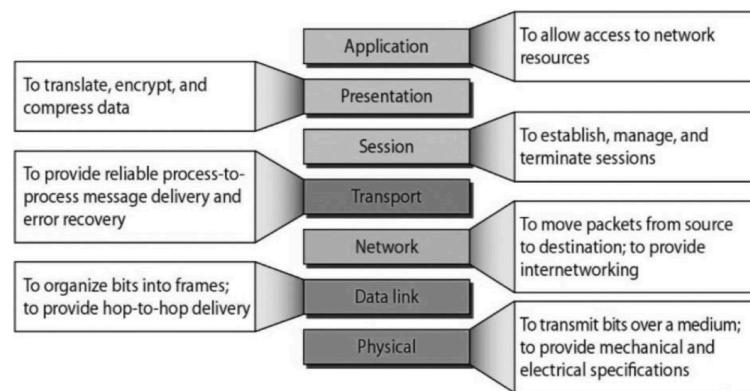
Für eine rechnerübergreifende Kommunikation stehen heutzutage verschiedene Protokolle und Spezifikationen zur Verfügung. Trotz der Vielfalt basieren heute die meisten Implementierungen auf den Grundkonzepten des OSI-Referenzmodells und legen damit schon die entscheidenden Weichen für geordnete Kommunikationsabläufe und saubere Netzwerkstrukturen.

#### 13.1.1 ISO/OSI-Modell

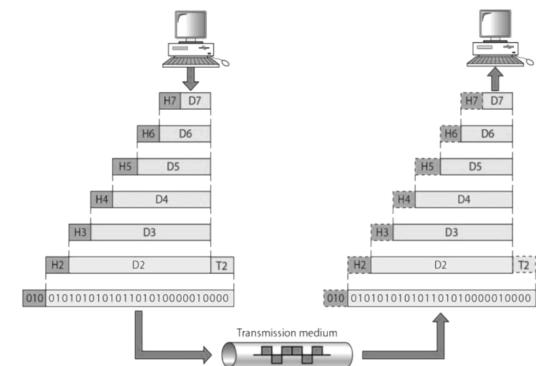
Das OSI-Referenzmodell beschreibt modellhaft die Kommunikation in 7 hierarchischen Schichten (Layer) unterschiedlicher Abstraktion. Hierbei stellt eine Schicht **i** Dienstleistungen für die darüber liegende Schicht **i+1** zur Verfügung, sie selbst bezieht Dienstleistungen von der unter ihr liegenden Schicht **i-1**.



Das OSI-Referenzmodell stellt eine Konstruktionsanleitung für Netzwerksoftware da. Über insgesamt sieben Layer sind abstrakte Funktionen definiert, die aufeinander aufbauen. Beim Datentransport werden die Nutzdaten in den jeweiligen Protokollrahmen eingebettet und der unteren Schicht weiter gereicht. ⇒ Kapselung



#### Kapselung



#### Bitübertragung (Physical Layer)

- Übertragung der abstrakten Informationseinheiten '0' und '1' mittels physikalischer Mittel über ein Medium.
- Überträgt einen Bitstrom von Punkt A nach B, ohne dessen Informationsgehalt zu modifizieren. Diese Übertragung erfolgt ungesichert, also ohne Fehlertoleranz.

#### Vermittlung (Network Layer)

- Verbindet Endsysteme miteinander (z.B. Computer P und Q). Dies kann die Benutzung eines oder mehrerer Kommunikationsnetze einschliessen.
- Wählt geeigneten Route (Leitweglenkung, Routing) für das Netzwerk, da meistens verschiedene vorhanden.

#### Sitzung (Session Layer)

- Stellt Mittel für eine geordnete Kommunikationsbeziehung zur Verfügung. Dazu gehört die Eröffnung, Durchführung und Beendigung der Session. Jede Session kennt die Phasen Verbindungsaufbau, Datentransfer und Verbindungsabbau.

#### Anwendung (Application Layer)

Über diese Schicht werden den Anwendungen die Kommunikationsdienste in anwendungsunabhängiger Form zugänglich gemacht.

#### Sicherung (Data Link Layer)

- Unveränderte übertragen von Daten, d.h. ohne Fehler und in der richtigen Reihenfolge. Sie stellt den übergeordneten Schichten gesicherte Verbindungen zur Verfügung.
- Durchführung der Flussregelung, also z.B. die Verhinderung eines Datenüberlaufs beim Empfänger (z.B. XON/XOFF-Protokoll).

#### Transport (Transport Layer)

- Stellt den Anwendungen transparente Datenkanäle zur Verfügung, sie schafft damit Verbindungen zwischen Anwendungsprozessen auf verschiedenen Systemen. Die Transparenz besteht darin, dass von der übergeordneten Schicht übernommene Daten (Bitstrom) unverändert übertragen werden.

#### Darstellung (Presentation Layer)

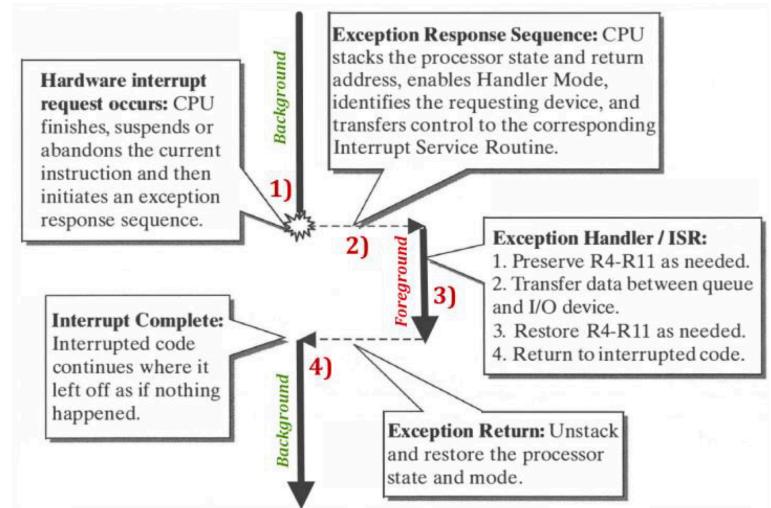
- Wird benötigt, wenn es um die Beschreibung von Daten geht, sofern diese Beschreibung nicht schon Teil der Applikation selbst ist. Zu diesem Zweck wird die Datenbeschreibung ASN1 (Abstract Syntax Notation One) eingesetzt.

## 13.2 Allgemeiner Ablauf von Exceptions und Interrupts

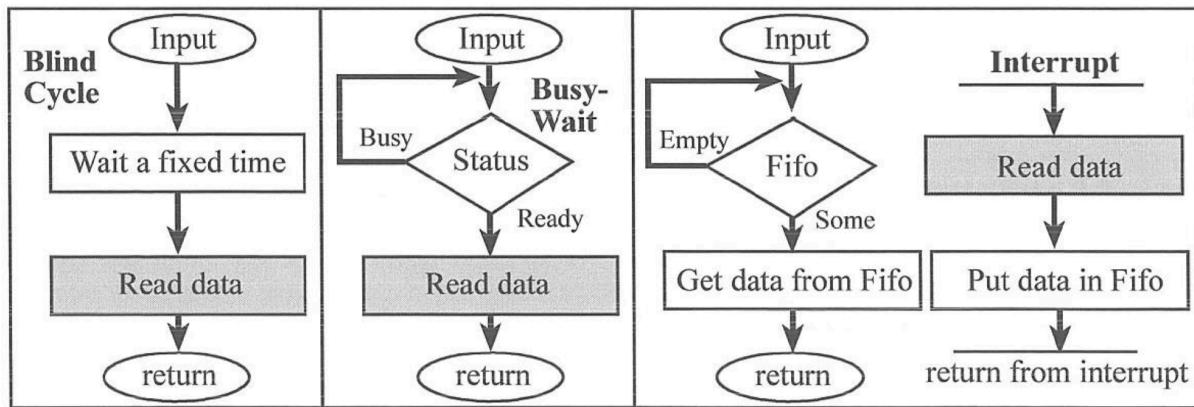
Interrupts werden in der Regel von der umgebenen Peripherie oder externen Input-Pins generiert und als Ereignis der CPU-Infrastruktur signalisiert, welche dann eine Handler-Routine einschalten.

**Siehe** Exceptions and Interrupts Seite: 14  
Der allgemeine Ablauf gliedert sich in mehrere Teilschritte:

1. Die Peripherie meldet einen Interrupt Request (IRQ) beim Prozessor an
2. Prozessor beendet die laufende Instruktion (Background)
3. Prozessor führt einen Exception Handler oder Interrupt Service Routine (ISR) aus (Foreground)
4. Prozessor fährt bei der nächsten Instruktion im Background weiter



## 13.3 Alternative Ansätze



### 13.3.1 Blind-Cycle

In der Software wird immer eine fixe Zeit abgewartet, bis die Daten an der Hardware abgeholt werden. Diese fixe Blind-Cycle Time ist so zu bemessen, dass die Daten von der HW in jedem Falle bereitgestellt werden können (worst case scenario).

### 13.3.2 Busy-Wait

In der Software wird solange in einem Loop verweilt, bis die Hardware über eine Status-Information signalisiert, dass die Daten bereitliegen. Damit können die empfangenen Daten mit Sicherheit abgeholt werden. Es etabliert sich so ein dynamisches Handshaking zwischen der Software und der Hardware.

### 13.3.3 Interrupt

Die Hardware meldet über einen IRQ asynchron zum aktuellen Programmablauf, dass die Daten bereitliegen. Die zugehörige ISR unterbricht den sequenziellen Ablauf im Background-Programm. In der ISR (Foreground) werden die Daten von der Hardware abgeholt und in einen FIFO-Buffer gelegt. Die Daten können danach im Background asynchron aus dem FIFO-Buffer gelesen und verarbeitet werden.

## 14 V14

### 14.1 Spezielle Eigenschaften des NVIC

#### 14.1.1 Tail Chaining

Wenn eine Exception auftritt während bereits eine andere Exception-Behandlung mit gleicher oder höherer Priorität läuft, so wird die neue Exception hintenangestellt. Nach Abschluss des laufenden Exception Handlers, kann die CPU sofort den neuen Exception Request behandeln

#### 14.1.2 Late arrival

Wenn der Prozessor einen auftretenden Exceptionrequest akzeptiert, dann startet er die Stacking-Sequenz. Kommt während dem stacking eine weitere Exception mit höherer Priorität hinzu, so kann diese Late-Arrival-Exception noch bevorzugt behandelt werden.

#### 14.1.3 POP Preemption

Diese Funktion stellt gewissermassen eine Umkehrung des Late-Arrivals dar. Wenn eine Exception Request während dem Unstacking auftritt, so wird das Unstacking abgebrochen, und sofort VectorFetch und Instruction Fetch für den neuen Request durchgeführt. → Geschwindigkeitsoptimierung

### 14.2 Cortex-M3 Exceptions und Priority-Levels

## Anhang

### Glossar, Abkürzung