



Modul BTE5053

Abteilung Elektrotechnik und Informationstechnologie

Hardwarenahe Softwareentwicklung

**Hardware-Architekturen
Assembler- und C-Programmierung
Peripheriebausteine**

Autoren: Roger Weber
Datum: 23. Dezember 2021
Version: 5.0

Vorwort

Das vorliegende Skript wird den Studierenden im Unterrichtsmodul „Hardwarenahe Softwareentwicklung“ der Abteilung Elektrotechnik und Informationstechnologie der BFH-TI in Burgdorf und Biel abgegeben. Es wird ergänzt durch Übungen, welche zusätzlich während dem Unterricht abgegeben werden.

Im ersten Teil des Skripts wird der Aufbau der Hardware eines Rechners erklärt. Es wird gezeigt wie eine CPU organisiert ist und wie sie mit Speicher und Peripherie kommuniziert. Im zweiten Teil steigen wir in die Assembler- und C-Programmierung ein. Dieser Einstieg erfolgt am Beispiel des STM32H743, einem 32-Bit Microcontroller von STMicroelectronics, basierend auf der ARM Architektur. Der dritte Teil befasst sich mit Peripheriebausteinen, welche die Umwelt an die CPU anbinden.

Am Schluss dieses Moduls sind Sie in der Lage:

- Einen Hardware-Schaltplan mit einer CPU, Speicher und Peripherie zu analysieren und zu entwerfen,
- die Software für ein Embedded System zu programmieren.

Dieses Skript basiert auf einem Vorgängerskript, bei welchem auch Gerhard Krucker und Tobias Rüetschi mitgearbeitet haben. Dieses wurde 2014 für den Microcontroller STM32F407 angepasst und um wesentliche Kapitel erweitert. Die aktuellste Version wurde für unsere Leguan-Boards und den Microcontroller STM32H743 angepasst und enthält auch Kapitel über Peripherie-Bausteine eines Microcontrollers.

Burgdorf, August 2021

Roger Weber

Inhaltsverzeichnis

I. Hardware-Architekturen	1
1. Computer-Systeme und Recheneinheiten	3
1.1. Kategorien	3
1.1.1. Klassifizierung nach Flynn	3
1.1.2. Kommunikation bei MIMD-Computern	5
1.2. Architekturen	7
1.2.1. Von-Neumann Architektur	7
1.2.2. Harvard Architektur	7
1.3. CPU	8
1.3.1. Steuerwerk	8
1.3.2. Rechenwerk	9
1.4. FPU	9
1.4.1. Floating Point Berechnungen	9
1.4.2. Floating Point Darstellung	10
1.4.3. Floating Point Programmierung in C	10
1.5. DSP	11
1.5.1. Einsatzgebiete	11
1.5.2. Architekturen	12
1.5.3. Programmierung in C	13
1.6. Co-Prozessoren	13
1.7. Prozessor-Typen	14
1.7.1. Prozessoren und Mikroprozessoren	14
1.7.2. Microcontroller	14
1.7.3. DSP	15
1.7.4. CISC / RISC	15
1.8. Beispiele von Microcontroller-Familien	15
1.9. Auswahlkriterien	16
2. Bussysteme und Speicher	19
2.1. Bus-Systeme	19
2.1.1. Adress-, Daten- und Control-Bus	19
2.1.2. Memory-Map	21
2.1.3. Addressdecoder	21
2.2. Speicher	22
2.2.1. Technologien	22
2.2.2. Organisation	23
2.2.3. Architekturen	24
2.2.4. Datenblätter	26
2.3. Speicher Hierarchie	27
3. Cache, MPU, MMU, DMA	29
3.1. Cache	29
3.1.1. Einleitung	29
3.1.2. Cache Architekturen	30
3.1.3. Cache Memory	31
3.1.4. Cache Controller	31
3.1.5. Beziehungen zwischen Cash und Main Memory	32
3.1.6. Caches und MMUs	33

3.1.7. Write Buffers	34
3.1.8. Cache Policies	34
3.2. MPU	34
3.2.1. Einleitung	34
3.2.2. Architektur	35
3.2.3. Empfehlungen für die Definition von Memory Regions	36
3.3. MMU	36
3.3.1. Einleitung	36
3.3.2. Virtuelle und physikalische Adressen	37
3.3.3. Pages und Frames, Page Table	38
3.3.4. Translation Lookaside Buffer	38
3.4. DMA	40
3.4.1. Einleitung	40
3.4.2. Funktionsweise des DMA-Controllers	40
4. ARM und Cortex-Mx Prozessoren	43
4.1. ARM Prozessoren	43
4.1.1. Einleitung	43
4.1.2. ARM-Architekturen	44
4.2. Cortex-Mx-Prozessoren	46
4.2.1. Übersicht	46
4.2.2. Vorteile	46
4.2.3. Architektur	47
4.2.4. Dokumentation	48
5. STM32H743	49
5.1. Eigenschaften	49
5.2. Blockdiagramm	50
5.3. Betriebsmodi	51
5.4. Program Status Register	51
5.5. Statusbits	52
5.6. Speichermodell, Datenformat	54
5.7. Memorymap STM32H7xx	55
5.8. Memorymap Leguan	56
5.9. Pinbelegung STM32H743	57
5.10. Anschluss SDRAM	58
5.11. Schaltplan	60
II. Assembler- und C-Programmierung	61
6. Einstieg Assembler-Programmierung	63
6.1. Einleitung	63
6.2. Sprachebenen	63
6.3. Aufbau einer Assembler-Datei	64
6.4. Syntax	65
6.4.1. Aufbau einer Assemblerzeile	65
6.4.2. Symbole	67
6.4.3. Konstanten	67
6.4.4. Operatoren und Operanden	68
6.5. Entwicklungsumgebungen	70
6.5.1. Übersicht	70
6.5.2. Editor	71
6.5.3. Assembler	71
6.5.4. Compiler	72
6.5.5. Linker / Locator	72
6.5.6. Debugger	73
6.5.7. Projektverwaltung	73

7. Instruktionssatz ARM V7M	75
7.1. Grundlagen	75
7.1.1. Einleitung	75
7.1.2. Syntax	75
7.1.3. Operanden	75
7.1.4. Bedingte Ausführung	77
7.2. Befehlsübersicht	77
7.3. Syntaxnotation	79
7.3.1. Indirektion, Indexierung []	80
7.3.2. Auswahl aus Aufzählung 	80
7.3.3. Element aus einer Aufzählung oder Menge <>	80
7.3.4. Optionale Elemente {}	80
7.4. Datentransfer	80
7.4.1. Registertransportbefehle MOV, MVN	80
7.4.2. Lade- und Speicherbefehle LDR, STR	81
7.4.3. Adressiermodi für Lade- und Speicherbefehle	82
7.4.4. Mehrfach Lade- und Speicherbefehle LDM, STM	84
7.4.5. Stackoperationen	85
7.5. Arithmetische und Logische Instruktionen	86
7.5.1. Integer-Arithmetik	87
7.5.2. Logische Instruktionen	89
7.5.3. Shift und Rotate	90
7.6. Programmverzweigungen	91
7.6.1. Generelles	91
7.6.2. Unbedingte Programmverzweigungen	92
7.6.3. Bedingte Programmverzweigungen	94
8. Assembler Direktiven	97
8.1. Einleitung	97
8.2. Kurzübersicht	97
8.3. .arm, .thumb, .syntax	98
8.4. .global, .extern	99
8.5. .align	100
8.6. .ascii, .asciz	101
8.7. .byte	101
8.8. .hword, .2byte	102
8.9. .word, .4byte	102
8.10. .space	103
8.11. .include	103
8.12. .equ, .set, =	104
8.13. .org	104
8.14. .section	104
8.15. .end	107
8.16. Assembler Operatoren	107
8.17. Assembler Kontrollstrukturen	108
8.17.1. .if	108
8.17.2. .ifdef	109
8.17.3. .ifndef	109
8.18. .macro	110
8.19. Assembler Wiederholungsstrukturen	110
8.19.1. .rept	111
8.19.2. .irp	112
9. Subroutinen	115
9.1. Einführung	115
9.2. Aufruf und Rücksprung	116
9.2.1. Unverschachtelte Aufrufe von Subroutinen	116

9.2.2. Verschachtelte Aufrufe von Subroutinen, Stack	116
9.3. Retten von Registern	120
9.4. Parameterübergabe	121
9.4.1. Parameterübergabe über Register	121
9.4.2. Parameterübergabe über den Stack	122
9.5. AAPCS	124
9.6. Lokale Variablen	125
9.7. Fehlerquellen	126
9.8. Vergleich Subroutinen und Makros	127
9.9. Heap	127
10. Startup-Code	129
10.1. Startup-Vorgang Übersicht	129
10.2. Reset	129
10.3. Bootloader	130
10.4. Startup-Code der Applikation	130
10.4.1. Übersicht	130
10.4.2. Vektortabelle	131
10.4.3. CPU Register	131
10.4.4. Initialisierte globale Variablen setzen	131
10.4.5. Uninitialisierte globale Variablen auf 0 setzen	133
10.4.6. Weitere Initialisierungen	133
10.5. Hauptprogramm	133
11. Projekte C / Assembler	137
11.1. Einleitung	137
11.2. Aufruf von Assembler-Subroutinen aus C	137
11.3. Parameterübergabe an Assembler-Subroutinen	138
11.4. Vom C-Compiler verwendete Register	138
11.5. Verwendete gemeinsame Variablen	138
11.6. Inline-Assembler	139
III. Peripheriebausteine	143
12. Zugriff auf die Peripherie	145
12.1. Übersicht	145
12.2. Zugriff auf Memory-Mapped Register	145
12.2.1. Einleitung	145
12.2.2. #define nach ANSI-C	146
12.2.3. Const Pointer	146
12.2.4. Absolute Sections	146
12.2.5. Spezielle Schlüsselwörter	147
12.3. Libraries	148
12.4. CMSIS	148
12.4.1. Die Vorteile von CMSIS	148
12.4.2. Die Layerung von CMSIS	149
12.4.3. Die Einbindung von CMSIS in ein Projekt	150
12.4.4. Beispiel	151
12.5. Leguan BSP	151
12.5.1. Board Support Package	152
12.5.2. FatFs	152
12.5.3. HAL Driver Library	152
12.5.4. CubeMX	154
13. GPIO	155
13.1. Anwendungen	155
13.2. Aufbau	155

13.3. Programmierung	157
13.4. Ausgangstreiber	158
13.4.1. Push-Pull	158
13.4.2. Open Drain	159
13.4.3. Tristate	159
14. Exceptions und Interrupts	161
14.1. Einleitung	161
14.2. Eigenschaften	162
14.3. Interrupts in Embedded Systems	162
14.3.1. Hardware-Interrupts	162
14.3.2. Software-Interrupts	163
14.4. Maskierung	163
14.4.1. Übersicht	163
14.4.2. Programmierung in C mit CMSIS	164
14.4.3. Programmierung in Assembler	164
14.5. Prioritäten	164
14.5.1. Übersicht	164
14.5.2. Programmierung in C mit CMSIS	165
14.5.3. Programmierung in Assembler	165
14.6. NVIC	166
14.6.1. Übersicht	166
14.6.2. Zustände	166
14.6.3. Register	167
14.6.4. Programmierung in C mit CMSIS	167
14.6.5. Programmierung in Assembler	168
14.7. Vektortabelle	168
14.8. External Interrupt/Event Controller (EXTI)	169
14.8.1. Übersicht	169
14.8.2. Programmierung in C mit der HAL-Library	171
14.8.3. Programmierung in C der EXTI-Register	171
14.9. Ablauf einer Exception	171
14.9.1. Exception Eingangssequenz	171
14.9.2. Ausführung des Exception Handler	172
14.9.3. Exception Rücksprung	172
14.9.4. Zusammenfassung	173
14.9.5. Spezialfälle	174
14.10 Zeitverhalten von Exceptions	174
14.11 Varianten von Interrupt-Handlern	175
15. RS232	177
15.1. Übersicht Serielle Schnittstellen	177
15.2. Einleitung RS232	177
15.3. Leitungslänge und Übertragungsrate	178
15.4. Steckerbelegung	178
15.5. Verkabelung	179
15.6. Signalpegel	179
15.7. Bitcodierung	180
15.8. Handshake-Signale	181
15.9. RS232 zu USB Konverter	181
15.10 Blockschaltbild	182
15.11 Programmierung der RS232-Peripherie auf dem STM32H7xx	184
16. I2C	187
16.1. Einleitung	187
16.2. Busprotokoll	187
16.3. Bitübertragung	188
16.4. Anwendungs-Beispiel	189

16.5. I2C-Peripherie auf dem STM32H7xx	190
16.6. Programmierung der I2C-Peripherie auf dem STM32H7xx	191
17. SPI	193
17.1. Einleitung	193
17.2. Bussignale	193
17.3. Interner Aufbau	194
17.4. Beispiel	195
17.5. Programmierung der SPI-Peripherie auf dem STM32H7xx	196
18. Timer, Counter, PWM, Watchdog	199
18.1. Einleitung	199
18.2. Timer/Counter	199
18.3. Timer mit Autoreload-Funktion	200
18.4. Compare-Einheit, PWM	200
18.5. Capture-Einheit	201
18.6. Watchdog	202
18.6.1. Einführung	202
18.6.2. Controller-Externer Watchdog	202
18.6.3. Controller-Interner Watchdog	202
18.6.4. Aufbau eines Watchdog	203
18.6.5. Starten des Watchdog	203
18.6.6. Refresh des Watchdog	203
18.7. Timer auf dem STM32H7xx	204
18.8. Programmierung der Timer auf dem STM32H7xx	205
19. A/D-Wandler	207
19.1. Einleitung	207
19.2. A/D-Wandler des STM32H7xx	207
19.3. Programmierung der ADC auf dem STM32H7xx	208
20. D/A-Wandler	211
20.1. Einleitung	211
20.2. D/A-Wandler des STM32H7xx	211
20.3. Programmierung der DAC auf dem STM32H7xx	212
IV. Anhang	215
A. Informationseinheiten	217
B. Zahlenformate	219
B.1. Zweierkomplement	219
B.2. Floating Point	219
C. AAPCS	223
C.1. Einleitung	223
C.2. Aufruf von Subroutinen	223
C.3. Verwendung des Stacks	225
C.4. Verwendete Datentypen	225
D. ASCII Tabelle	227
E. Befehlsübersicht Cortex-Mx	229
F. Kontroll- und Datenstrukturen in Assembler	235
F.1. Einführung Kontrollstrukturen	235
F.2. Einfachentscheidung	235
F.3. Mehrfachentscheidung	237

F.4. Schleife mit Vorabprüfung	238
F.5. Schleife mit Endprüfung	239
F.6. Zählschleifen	240
F.7. Einführung Datenstrukturen	240
F.8. Eindimensionale Arrays	241
F.9. Zweidimensionale Arrays	242
F.10. Strings	243
F.11. Strukturen	243
F.12. Stack	244
F.13. Queue / Ringbuffer	245
F.14. Kommentare	247
Glossar	249
Literaturverzeichnis	251
Stichwortverzeichnis	253

Abbildungsverzeichnis

1.1.	Klassifizierungsverfahren von Computern nach Flynn	3
1.2.	SISD	4
1.3.	SIMD	4
1.4.	MISD	5
1.5.	MIMD	5
1.6.	Shared Memory Systeme	6
1.7.	Distributed Memory Systeme	6
1.8.	Mischung aus Shared Memory und Distributed Memory Systeme	7
1.9.	von-Neumann Architektur	7
1.10.	Harvard Architektur	8
1.11.	Aufbau einer CPU	8
1.12.	Floating Point Single Precision Datenformat	10
1.13.	Sharc ADSP21469 Blockdiagramm. Quelle: Datenblatt Analog Device ADSP-21467	12
2.1.	Address-, Data- und Controlbus	19
2.2.	Bustiming	20
2.3.	Memory-Map eines 16-Bit Adressbusses	21
2.4.	Programmierbarer Adressdecoder	21
2.5.	Blockschaltbild mit Adressdecoder, Speicher und Peripherie	22
2.6.	Aufbau eines byte-organisierten Speichers	23
2.7.	Architektur eines ROM-Bausteins	24
2.8.	Architektur eines SRAM-Bausteins	25
2.9.	Block-Diagramm des SDRAMs K4S561633F von Samsung (4M * 16 Bit * 4 Banks) Quelle: [Samsung SDRAM]	26
2.10.	Zeitlicher Verlauf eines Lesezugriffs auf das SDRAM	26
2.11.	Speicherhierarchie	27
3.1.	a) System ohne Cache und b) System mit Cache	29
3.2.	4KB Cache bestehend aus 256 Cache Lines zu je vier 32-bit Words	30
3.3.	Direct Mapped Cache	32
3.4.	4KB Four-Way Set Associative Cache, bestehend aus 4*64 Cache Lines zu je vier 32-bit Words.	33
3.5.	a) Virtual Cache und b) Physical Cache	33
3.6.	Architektur MPU, Quelle: [13]	35
3.7.	MMU zwischen CPU und Speicher	37
3.8.	Vereinfachtes Blockschaltbild einer MMU	37
3.9.	Page Table für 16 * 4kByte Pages	38
3.10.	MMU mit Page Table und TLB	39
3.11.	Datentransfer ohne DMA, über die CPU	40
3.12.	Datentransfer mit DMA	40
4.1.	Die ARM Cortex-Mx-Prozessor-Familie. Quelle: [13]	46
5.1.	STM32H743 Blockdiagramm, Quelle: [12]	50
5.2.	Operation States und Modes. Quelle: [2] und [13]	51
5.3.	APSR, IPSR und EPSR. Quelle: [5]	52
5.4.	Zahlenkreis vorzeichenloser Dualzahlen	53
5.5.	Zahlenkreis vorzeichenbehafteter Dualzahlen	54
5.6.	Memory Map STM32H7xx	56
5.7.	Memory Map Leguan-Board, Peripherie und SDRAM	57
5.8.	STM32H743 Pinout, Quelle: [12]	58

5.9. Ansteuerung eines externen 16-Bit SDRAM-Bausteins am Beispiel IS42S16320D. Quelle: Schaltplan Leguan	59
6.1. Verschiedene Sprachebenen	63
6.2. Möglicher Aufbau einer Assembler-Datei	64
6.3. Aufbau einer Assemblerzeile	65
6.4. Ausschnitt aus dem Memory mit der Konstanten 0x12345678 auf Adresse 0x080001e4 bis 0x080001e7	68
6.5. Entwicklungsablauf	70
6.6. Informationen aus dem elf-File mit OBJDUMP.EXE	73
7.1. ALU mit Barrel-Shifter	76
7.2. Mehrfach Lade- und Speicherbefehle LDM und STM	85
7.3. Beispiel Speicherbefehle STMIB	85
7.4. Beispiel PUSH Instruktion	86
7.5. Absolute Programmverzweigung	91
7.6. relative Programmverzweigung	92
8.1. Beispiel .global und .extern Assembler Direktiven	99
8.2. Mit .rept wiederholte Struktur im Speicher	112
9.1. Aufruf einer Subroutine	116
9.2. Verschachtelter Aufruf von Subroutinen	117
9.3. Stack	117
9.4. Ablegen der Rücksprungadresse auf dem Stack	118
9.5. Rücksprungadressen auf dem Stack	
a) Stack im aufrufenden Programm	
b) Stack innerhalb von „subr_1“	
c) Stack innerhalb von „subr_2“	120
9.6. Parameterübergabe auf dem Stack:	
a) vor dem Aufruf der Subroutine super_sum und dem Speichern der Parameter auf dem Stack	
b) nach dem Eintritt in die Subroutine, die Parameter p5 und p6 liegen auf dem Stack	
c) innerhalb der Subroutine nach der Instruktion PUSH	124
9.7. Lokale Variablen auf dem Stack	126
10.1. Übersicht Startup-Vorgang	129
10.2. Initialisierung von globalen Variablen	132
10.3. Hauptprogramm	134
10.4. Datenaustausch zwischen ISR und Applikationsfunktionen	135
12.1. Library Layerung	148
12.2. CMSIS Layerung. Quelle: [6]	149
12.3. Einbindung der Device Driver Library und der CMSIS Library in ein Projekt. Quelle: [13]	150
12.4. Layerung Leguan Libraries	151
12.5. Konfiguration des Microcontrollers mit CubeMX	154
13.1. Beispiel Verwendung von GPIOs	155
13.2. Aufbau eines GPIO Port Pins, [11]	156
13.3. Port Output Data Register, [11]	157
13.4. Port Bit set/reset Register, [11]	157
13.5. Ausgangstreiber	158
13.6. Interrupt-Quellen mit Open-Drain-Ausgängen	159
14.1. Quellen von Exceptions, [13]	161
14.2. Programmablauf beim Eintreffen einer Exception	162
14.3. Mehrstufige Maskierung von Interrupts beim STM32F743	164
14.4. Priorisierung und Verschachtelung von Exceptions	165
14.5. Interrupt pending und Interrupt aktiv, [13]	167
14.6. Blockschaltbild EXTI STM32H7xx, [11]	170

14.7. Exception Stack nach Aufruf des Handlers (ohne FPU)	172
14.8. Ablauf einer Exception-Bearbeitung	173
14.9. Verzögerungszeiten bei der Verarbeitung von Exceptions	174
14.10 Nonnested Interrupt Handler	175
15.1. Prinzipieller Aufbau einer seriellen Schnittstelle	177
15.2. RS-232-Stecker, oben Stecker (male), unten Buchse (female)	178
15.3. MAX3245, RS-232 Bus-Transceiver	180
15.4. RS-232 Frame	180
15.5. Beispiel RS-232 zu USB converter FT230XQ	182
15.6. Generisches Blockschaltbild RS232-Schnittstelle mit FIFO	183
15.7. UART/USART im STM32H7xx	183
16.1. Applikation mit einem I2C-Bus, Quelle: [8]	187
16.2. Beispiel Protokoll mit Slaveadresse und 2 Byte Daten, Quelle: [8]	187
16.3. Start- und Stop-Condition	189
16.4. Übertragung auf Bitebene	189
16.5. Geodatenlogger, Kommunikation Microcontroller, Farbsensor und RTC über I2C	190
16.6. I2C-Peripherie auf dem STM32H7xx, Quelle [11]	191
17.1. SPI Bus mit einem Master und drei parallelen Slaves	194
17.2. SPI Schieberegister, ein Master und zwei Slaves	195
17.3. Geodatenlogger, Kommunikation Microcontroller und Flash über SPI	196
18.1. Prinzipieller Aufbau eines Timer/Counter	199
18.2. Timer mit Autoreload-Mode	200
18.3. Compare-Einheit	200
18.4. Compare-Mode	201
18.5. Capture-Mode	201
18.6. Externer Watchdog	202
18.7. Aufbau eines Watchdogs	203
18.8. Blockschaltbild der General Purpose Timer des STM32H7xx, Quelle: [11]	204
19.1. Prinzipieller Aufbau eines A/D-Wandlers	207
19.2. Blockschaltbild A/D-Wandler des STM32H7xx, Quelle: [11]	208
20.1. Blockschaltbild D/A-Wandler des STM32H7xx, Quelle: [11]	212
A.1. Bit	217
A.2. Nibble	217
A.3. Byte	217
A.4. Halfword	217
A.5. Word	218
A.6. Double Word	218
B.1. Floating Point Single Precision Datenformat	219
B.2. Floating Point Double Precision Datenformat	220
F.1. Einfachentscheidung	235
F.2. Einfachentscheidung in C und Assembler	236
F.3. Mehrfachentscheidung	237
F.4. Mehrfachentscheidung in C und Assembler	237
F.5. Schleife mit Vorabprüfung	238
F.6. Schleife mit Vorabprüfung in C und Assembler	238
F.7. Schleife mit Endprüfung	239
F.8. Schleife mit Endprüfung in C und Assembler	239
F.9. Zählschleife	240
F.10. Zählschleife in C und Assembler	240
F.11. Zweidimensionale Array	242

F.12. Zeilenorientiertes Speichern des zweidimensionalen Arrays	242
F.13. Stack nach der Initialisierung	244
F.14. Queue	245
F.15. Head und Tail im Ringbuffer, gemäss Variante 1	246
F.16. Beispiel Ringbuffer mit 8 Elementen	246

Tabellenverzeichnis

1.1. Beispiele Floating Point Werte, Single Precision, normalisierte Form	10
1.2. Verschiedene Microcontroller-Familien	16
2.1. Adress- Daten- und Controlbus	19
3.1. Mögliche Einträge im TLB	39
4.1. Übersicht der ARM-Architekturen, Teil I	44
4.2. Übersicht der ARM-Architekturen, Teil II	45
5.1. Statusbits	53
5.2. Byte-Speichermodell	55
5.3. Halfword-Speichermodell	55
5.4. Word-Speichermodell	55
5.5. Signale zur Ansteuerung des PSRAM-Bausteins	59
6.1. Regel für Konstanten	67
6.2. Operatoren für die Assembler-Programmierung	69
6.3. Spalten des Assembler-Listings	72
7.1. Barrel-Shifter	76
7.2. Condition code suffixes	77
7.3. Instruktionen Thumb-2, Zusammenfassung Teil I	78
7.4. Instruktionen Thumb-2, Zusammenfassung Teil II	79
7.5. Syntaxelemente für formale Befehlsbeschreibungen	79
7.6. Datentransferbefehle	80
7.7. Lade- und Speicherbefehle	81
7.8. Adressiermethoden für den Speicherzugriff	82
7.9. Syntax der Adressiermethoden für einfachen Speicherzugriff mit Word	83
7.10. Zulässige Adressiermethoden für signed /unsigned Halfword und signed/unsigned Byte Daten	83
7.11. Adressmodi für LDM-/STM-Instruktionen	84
7.12. Zusammenfassung der arithmetisch-logischen Instruktionen	87
7.13. Integer-Arithmetik Instruktionen, Teil I	87
7.14. Integer-Arithmetik Instruktionen, Teil II	88
7.15. Logische Instruktionen	90
7.16. Programmverzweigungen	93
7.17. Bedingungen (Condition Codes) für bedingte Programmverzweigungen	94
8.1. Kurzübersicht Assembler-Direktiven	98
8.2. Vordefinierte Segmentnamen	105
8.3. Zugriffs-Flags für die Segmentdefinition	105
8.4. Assembler Vorzeichenoperatoren	107
8.5. Assembler Operatoren	108
9.1. Stackoperationen beim Aufrufen und Beenden von Subroutinen	119
9.2. Verwendung der ARM CPU-Register gemäss AAPCS	125
14.1. Übersicht Register NVIC	167
14.2. Vektortabelle ARM Cortex-M3, Cortex-M4 und Cortex-M7	169
14.3. Übersicht Register EXTI-Controller	170
14.4. Die einzelnen Bits von EXC_RETURN	173

15.1. Maximale Kabellänge bei RS-232	178
15.2. Pinbelegung RS-232 DSUB-9	179
15.3. RS-232 Bitcodierung	181
15.4. Übersicht Register UART-Controller	184
15.5. Übersicht HAL-Funktionen UART-Controller	184
16.1. Beispiele I2C-Protokolle	188
16.2. Übersicht Register I2C-Controller	191
16.3. Übersicht HAL-Funktionen I2C-Controller	192
17.1. Übersicht Register SPI-Controller	197
17.2. Übersicht HAL-Funktionen SPI-Controller	197
18.1. Übersicht Register Timer	205
18.2. Übersicht HAL-Funktionen Timer	205
19.1. Übersicht Register ADC	208
19.2. Übersicht HAL-Funktionen ADC	209
20.1. Übersicht Register DAC	212
20.2. Übersicht HAL-Funktionen DAC	213
B.1. Zweierkomplement	219
B.2. Floating Point Werte, Single Precision	220
B.3. Floating Point Werte, Double Precision	220
C.1. Verwendung der ARM CPU-Register gemäss AAPCS	224
C.2. Fundamentale Datentypen gemäss AAPCS	225
D.1. ASCII-Tabelle	227
E.1. Assembler Befehlsübersicht	229
E.1. Assembler Befehlsübersicht	230
E.1. Assembler Befehlsübersicht	231
E.1. Assembler Befehlsübersicht	232
E.1. Assembler Befehlsübersicht	233

Teil I.

Hardware-Architekturen

1. Computer-Systeme und Recheneinheiten

Dieses Kapitel befasst sich mit der Architektur von Microcomputern und mit dem Aufbau der Hardware. Das Kapitel ist sehr allgemein gehalten und geht mit Ausnahme des Kapitels 1.8 nicht auf einzelne Hersteller und CPU-Familien ein.

1.1. Kategorien

1.1.1. Klassifizierung nach Flynn

Computer können gemäss Michael J. Flynn in vier Kategorien eingeteilt werden. Dabei wird zwischen der Anzahl Programminstruktionen (Programmbefehlen) und der Anzahl Daten unterschieden, welche gleichzeitig verarbeitet werden können. Programminstruktionen und Daten können je die Zustände „single“ und „multiple“ annehmen.

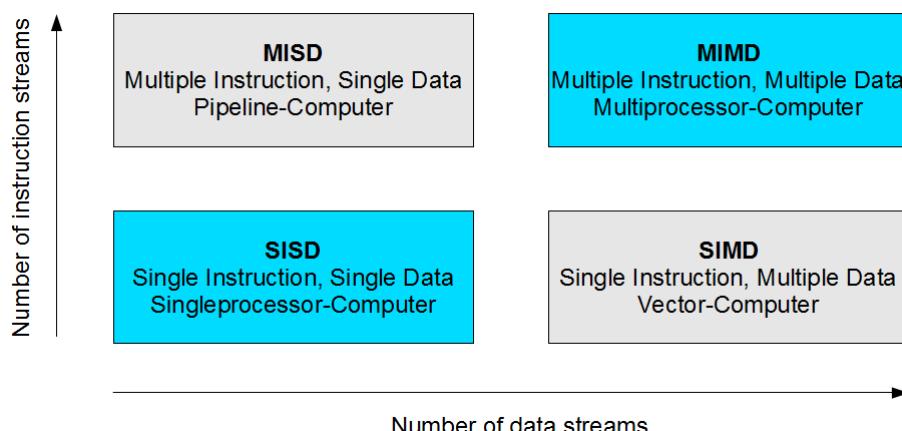


Abbildung 1.1.: Klassifizierungsverfahren von Computern nach Flynn

SISD

SISD (Single Instruction Single Data) Computer arbeiten sequenziell. Sie bearbeiten pro Zeiteinheit einen Datensatz mit einer Programminstruktion:

- Single Instruction heisst, dass pro Taktzyklus genau eine Programminstruktion ausgeführt wird.
- Single Data heisst, dass pro Taktzyklus nur ein Datensatz verarbeitet wird.

SISD Computer sind heute weitaus am häufigsten anzutreffen, insbesondere in einfacheren Embedded Systems.

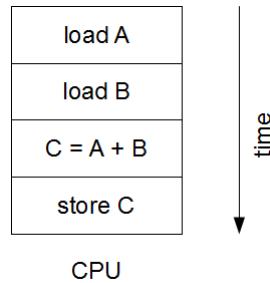


Abbildung 1.2.: SISD

SIMD

SIMD (Single Instruction Multiple Data) Computer erlauben die gleichzeitige Ausführung derselben Programminstruktion auf mehrere Datensätze. Sie werden beispielsweise in der Bildverarbeitung eingesetzt.

- Single Instruction heisst, dass alle CPUs dieselbe Programminstruktion pro Taktzyklus ausführen.
- Multiple Data heisst, dass jede CPU einen anderen Datensatz bearbeitet.

Viele moderne Mikroprozessoren wie PowerPC, x86 oder ARM Cortex Mx besitzen SIMD-Erweiterungen mit speziellen zusätzlichen Befehlssätzen, sodass eine CPU mit einer Programminstruktion gleichzeitig mehrere Datensätze verarbeiten kann.

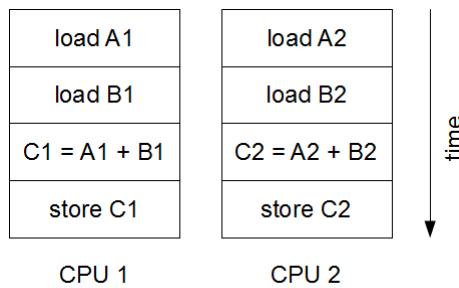


Abbildung 1.3.: SIMD

MISD

Bei MISD (Multiple Instruction Single Data) Computern wird ein Datensatz gleichzeitig von mehreren CPUs mit unterschiedlichen Programminstrukturen verarbeitet.

- Multiple Instruction heisst, dass jede CPU eine andere Programminstruktion pro Taktzyklus ausführt.
- Single Data heisst, dass alle CPUs denselben Datensatz bearbeitet.

Für MISD-Computer gibt es heute fast keine sinnvollen Anwendungen.

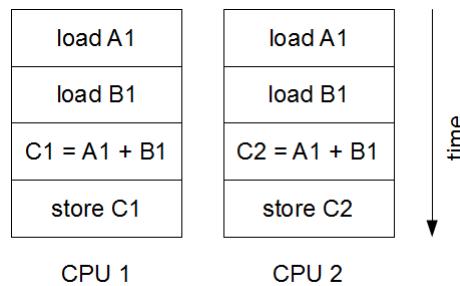


Abbildung 1.4.: MISD

MIMD

MIMD (Multiple Instruction Multiple Data) Systeme werden häufig durch den Einsatz von Multi-Core Prozessoren gebildet. Typische Anwendungen sind PCs und leistungsfähige Embedded Systems. Aber auch verteilte Rechner systeme (sogenannte Cluster) gehören in die Kategorie der MIMD Systeme.

- Multiple Instruction heisst, dass jede CPU eine andere Programminstruktion pro Taktzyklus ausführt. Häufig werden ganze Programme oder Teile davon (Prozesse, Threads, Tasks) auf unterschiedliche CPUs aufgeteilt.
- Multiple Data heisst, dass jede CPU einen anderen Datensatz bearbeitet.

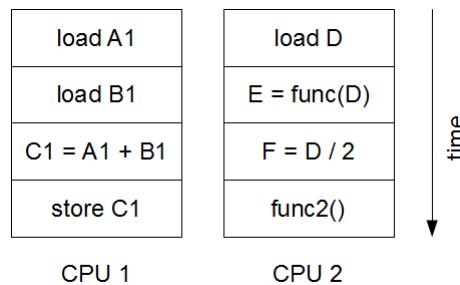


Abbildung 1.5.: MIMD

1.1.2. Kommunikation bei MIMD-Computern

Bei Microcomputern mit mehreren CPUs gibt es unterschiedliche Ansätze, wie die einzelnen CPUs untereinander und mit dem Speicher kommunizieren. Diese werden in den nächsten Unterkapiteln diskutiert.

Shared Memory (Eng gekoppelte Systeme)

Shared Memory Systeme verwenden einen gemeinsamen Speicher für verschiedene CPUs. Für diese Systeme gilt:

- Mehrere CPUs teilen sich einen gemeinsamen Speicher (Shared Memory).
- Die CPU arbeiten unabhängig voneinander. Ändert eine CPU einen Speicherbereich, so ist dies für die anderen CPUs sofort ersichtlich. Die CPUs können über das Shared Memory einfach und schnell miteinander kommunizieren.
- Der Zugriff auf den Speicher muss unter den CPUs synchronisiert werden, es können nicht gleichzeitig mehrere CPUs auf den Speicher zugreifen.

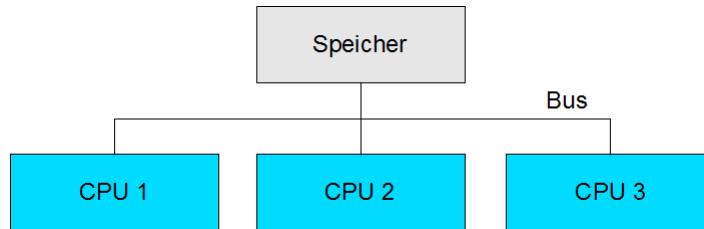


Abbildung 1.6.: Shared Memory Systeme

Distributed Memory (Lose gekoppelte Systeme)

Distributed Memory Systeme bestehen aus mehreren Einzelsystemen, die miteinander kommunizieren. Für diese Systeme gilt:

- Mehrere CPUs kommunizieren über ein Netzwerk miteinander.
- Die CPUs arbeiten unabhängig voneinander. Jede CPU hat ihren eigenen Speicher. Der Datenaustausch zwischen den CPUs erfolgt über ein Netzwerk.
- Dieses System kann aus mehreren SISD-Systemen mit Netzwerkanbindung aufgebaut werden.

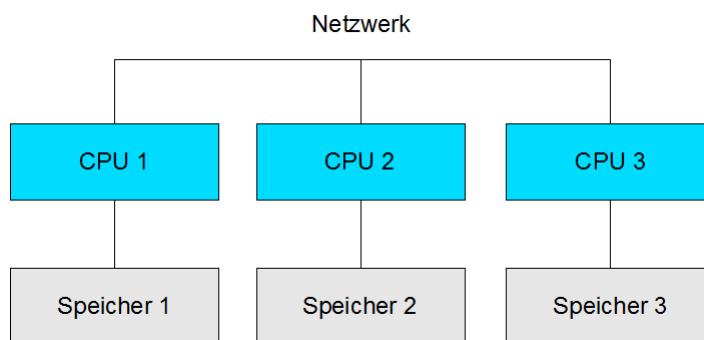


Abbildung 1.7.: Distributed Memory Systeme

Mischung aus Shared Memory und Distributed Memory Systemen

Ein Vergleich von Shared Memory und Distributed Memory Systemen ergibt folgendes:

- Distributed Memory Systeme sind sehr gut skalierbar. Braucht man mehr Rechenleistung, fügt man zusätzliche Einzelsysteme hinzu.
- Der Zugriff auf den lokalen Speicher ist bei Distributed Memory Systemen schneller als bei Shared Memory Systemen.
- Der Datenaustausch zwischen den CPUs ist bei Distributed Memory über das Netzwerk komplizierter und langsamer als bei Shared Memory Systemen über den gemeinsamen Speicher.

Da sowohl Shared-Memory als auch Distributed-Memory Systeme ihre Vor- und Nachteile haben, verwenden schnelle Computer oft eine Mischung beider Systeme.

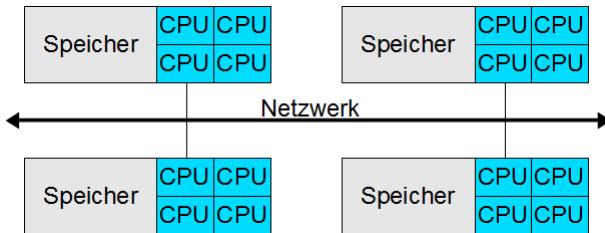


Abbildung 1.8.: Mischung aus Shared Memory und Distributed Memory Systeme

1.2. Architekturen

Für den Zugriff von der CPU auf den Speicher existieren bei Computern zwei gängige Architekturen: Die „von-Neumann“ und die „Harvard“ Architektur. Beide Architekturen gehören nach dem Klassifizierungsverfahren von Michael J. Flynn zur Kategorie der SISD-Systeme.

1.2.1. Von-Neumann Architektur

Der grösste Teil der heutigen Computer verwendet die „von-Neumann“ Architektur, benannt nach dem ungarischen Mathematiker János von Neumann (1903 - 1957). Bei dieser Architektur liegen Programmspeicher, Datenspeicher und Peripherie im gleichen Adressraum. Dadurch kann die CPU sowohl auf den Programmspeicher als auch auf den Datenspeicher und die Peripherie über denselben Bus zugreifen. Programminstruktionen und Daten können nur sequenziell gelesen werden, d.h. es sind mindestens zwei aufeinander folgende Taktzyklen notwendig, um eine Programminstruktion und die zugehörigen Daten zu lesen.

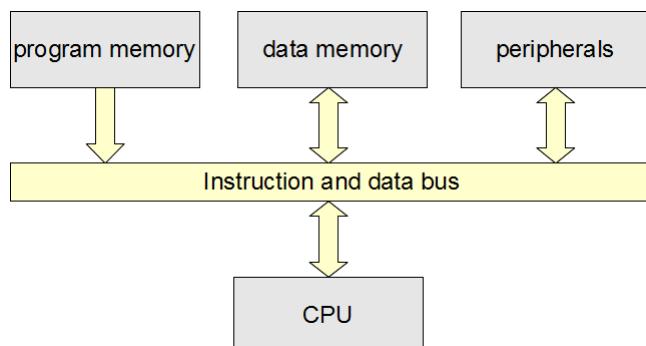


Abbildung 1.9.: von-Neumann Architektur

Im Programmspeicher wird das Programm in Form von einzelnen Programminstruktionen gespeichert. Der Programmspeicher ist normalerweise nicht flüchtig, d.h. die Programminstruktionen gehen bei Spannungsunterbrüchen nicht verloren. Im Datenspeicher werden Variablen und Anwenderdaten abgelegt, dieser Speicher ist flüchtig. Die CPU liest die Programminstruktionen aus dem Programmspeicher und manipuliert (lesen und schreiben) die Daten im Datenspeicher und der Peripherie abhängig von den Programminstruktionen.

Als von-Neumann-Flaschenhals wird der Sachverhalt bezeichnet, dass das Bussystem zum Engpass zwischen der CPU und dem Speicher wird.

1.2.2. Harvard Architektur

Bei der Harvard-Architektur werden Programmspeicher und Datenspeicher in unterschiedlichen, physikalisch getrennten Adressräumen abgelegt und es führen unterschiedliche Busse (Instruktionsbus, Datenbus) zu diesen Speicherbereichen. Programminstruktionen und Daten können parallel (d.h. gleichzeitig) geladen werden. Die Busbreiten für Instruktionen und Daten können unterschiedlich sein.

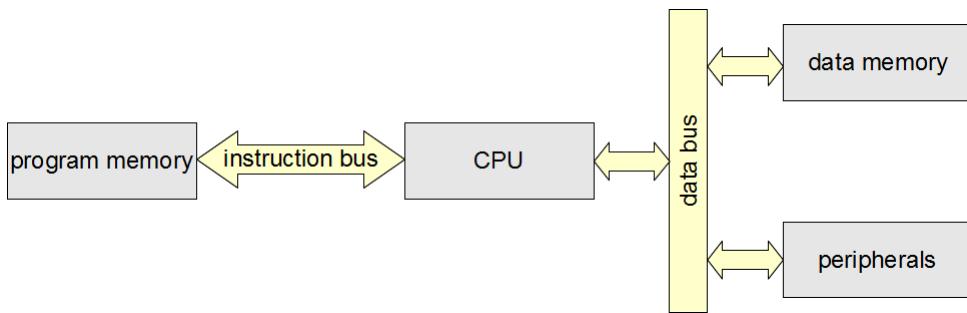


Abbildung 1.10.: Harvard Architektur

Die Harvard-Architektur findet man oft bei DSP's (Digitale Signal-Prozessoren), vereinzelt auch bei Mikroprozessoren. Der Vorteil dieser Architektur besteht darin, dass Programminstruktionen und die zugehörigen Daten in einem einzigen Taktzyklus in die CPU geladen werden können. Somit ist die Harvard-Architektur schneller als die von-Neumann Architektur, jedoch auch aufwendiger, was sich schlussendlich im Preis niederschlägt.

Die folgenden Kapitel beschränken sich auf die von-Neumann Architektur.

1.3. CPU

Die CPU (Central Processing Unit) ist das eigentliche Herz eines Computers. Sie steuert den Programmablauf und verarbeitet die Daten. Abbildung 1.11 zeigt den klassischen Aufbau einer CPU. Je nach Hersteller und Typ kann dieser Aufbau etwas variieren.

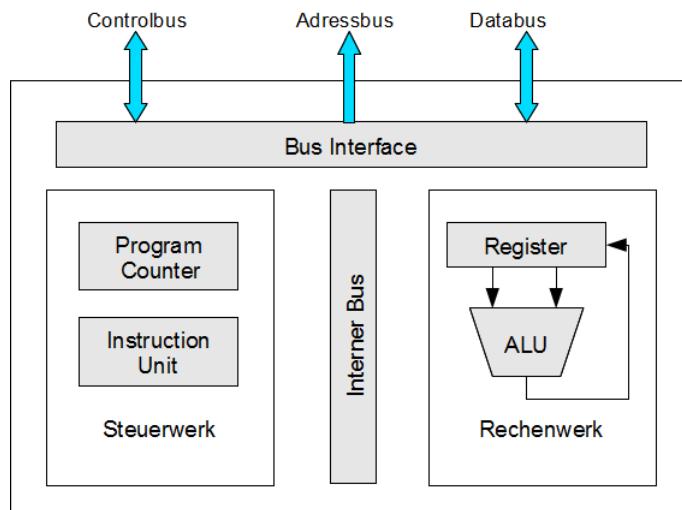


Abbildung 1.11.: Aufbau einer CPU

1.3.1. Steuerwerk

Das Steuerwerk ist für den Programmablauf verantwortlich. Es besteht im wesentlichen aus folgenden Elementen:

- Instruction Unit (Schaltwerk): Sie interpretiert die Instruktionen und ist für deren Verarbeitung zuständig.
- Program-Counter (PC): Er enthält die Adresse der nächsten auszuführenden Programminstruktion im Programmspeicher.

Die Aufgaben des Steuerwerks sind folgende:

- Laden der nächsten Programminstruktion aus dem Programmspeicher (fetch).

- Entschlüsseln der Programminstruktion (decode) nach Operation und Operanden.
- Befehlausführung (execute) in folgenden Schritten:
 - Steuersignale für ALU oder andere Funktionseinheiten generieren
 - Operanden adressieren und laden
 - Ergebnis speichern
 - Program-Counter anpassen

1.3.2. Rechenwerk

Das Rechenwerk ist für die Verarbeitung der Daten zuständig. Es besteht aus folgenden Komponenten:

- ALU (Arithmetic Logical Unit), führt die Rechenoperationen durch. Die ALU kann nur Integer-Operationen durchführen. Für Floating-Point Instruktionen oder komplexe mathematische Funktionen wird oft auch noch eine FPU (Floating Point Unit) verwendet, siehe Kapitel 1.4. Typische Operationen der ALU sind:
 - Transfer Operationen (laden und speichern)
 - Boolsche Operationen (logisch AND, OR, EXOR, NOT)
 - Arithmetische Operationen (Addition, Subtraktion, Multiplikation)
 - Vergleiche und logische Entscheidungen (Vergleiche und Verzweige falls gleich)
 - Schiebe-Operationen (shift left, shift right)
- Register, welche die Operanden und die Resultate der Operation speichern (oft auch Akkumulatoren genannt). Ein Register ist ein schneller, prozessorinterner Speicher, welcher nicht über den Bus sondern direkt von der CPU angesprochen wird.

Der interne Bus verbindet Steuerwerk, Rechenwerk und Businterface. Die im Rechenwerk auszuführenden Operationen werden durch das Steuerwerk bestimmt. Andererseits liefert das Rechenwerk dem Steuerwerk Informationen über den aktuellen Zustand. Das Businterface ist für die Ansteuerung aller externen Einheiten zuständig (Speicher, Peripherie).

1.4. FPU

1.4.1. Floating Point Berechnungen

Reelle Zahlen werden in Computern als Floating Point (Gleitpunktzahlen), üblicherweise im IEEE-754 Format, dargestellt. Dazu wird die Floating Point Zahl durch ein Vorzeichen (sign), einen Exponent in Bezug auf die Basis 2, sowie eine Fraction (Mantisse) abgebildet (siehe auch Kapitel 1.4.2).

Die Darstellung von Zahlen im Floating Point Format hat im Vergleich zu Integern folgende Vorteile:

- Der Wertebereich der Zahlen ist sehr gross. Es gibt keine Überläufe wie bei Integer, was z. Bsp. bei der Berechnung von Reglern von grossem Vorteil ist.
- Es können ebenfalls sehr kleine Zahlenwerte dargestellt werden.

Die Berechnung von Floating Point Operationen kann entweder in Software mit Hilfe von Libraries erfolgen, oder in Hardware durch die Verwendung einer FPU (Floating Point Unit). Durch die Verwendung einer FPU wird die Berechnung wesentlich schneller, dafür ist der Aufwand in der Hardware grösser und damit teurer. Die Berechnung in Software andererseits braucht Programmspeicher für die Library und erfordert eine längere Berechnungszeit.

FPU's unterstützen üblicherweise folgende Operationen:

- Addition, $z = a + b$
- Subtraktion, $z = a - b$

- Multiplikation, $z = a * b$
- Division, $z = a / b$
- Quadratwurzel, $z = \text{sqrtf}(a)$
- Negation, $z = -a$
- Betrag, $z = \text{absf}(a)$

FPU's führen die oben genannte Operationen normalerweise im Single Precision Format aus (siehe auch Kapitel 1.4.2). Werden andere Operationen wie `sinf()`, `cosf()` usw. gewünscht, oder soll die Berechnung im Double Precision Format ausgeführt werden, ist trotzdem eine Software-Library erforderlich.

1.4.2. Floating Point Darstellung

IEEE-754 definiert verschiedene Datenformate für Floating Point. Die wichtigsten und am häufigsten verwendeten sind Single Precision (32 Bit) sowie Double Precision (64 Bit). Das Single Precision Format wird an dieser Stelle kurz vorgestellt. Weiter Informationen sind im Anhang B zu finden.

Das Single Precision Format besteht aus einem Vorzeichen (sign, 1 Bit), einem Exponent (8 Bit) sowie einer Fraction (23 Bit).

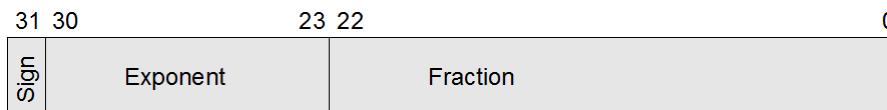


Abbildung 1.12.: Floating Point Single Precision Datenformat

Der Exponent nimmt in der Regel einen Wert zwischen 1 und 254 ein, dies entspricht der normalisierten Form. Die Fraction hat dann einen Wert zwischen 1.0 und 2.0. Der dargestellte Wert berechnet sich zu:

$$\text{Value} = (-1)^{\text{Sign}} * 2^{(\text{Exponent} - 127)} * (1 + (\frac{1}{2} * \text{Fraction}[22]) + (\frac{1}{4} * \text{Fraction}[21]) + \dots + (\frac{1}{2^{23}} * \text{Fraction}[0]))$$

Beispiele für Floating Point Werte (aus [13]):

Floating Point Value	Sign	Exponent	Fraction	Hex Value
1.0	0	127 (011 1111 1)	000 0000 0000 0000 0000 0000	0x3F800000
1.5	0	127 (011 1111 1)	100 0000 0000 0000 0000 0000	0x3FC00000
1.75	0	127 (011 1111 1)	110 0000 0000 0000 0000 0000	0x3FE00000
0.04 → $1.28 * 2^{-5}$	0	127 - 5 (011 1101 0)	010 0011 1101 0111 0000 1010	0x3D23D70A
-4.75 → $-1.1875 * 2^2$	1	127 + 2 (100 0000 1)	001 1000 0000 0000 0000 0000	0xC0980000

Tabelle 1.1.: Beispiele Floating Point Werte, Single Precision, normalisierte Form

1.4.3. Floating Point Programmierung in C

Floating Point Variablen werden in C wie folgt definiert:

```
float pi = 3.141592F;           // Single Precision, 32-Bit
double pi2 = 3.14159265358979323846264338; // Double Precision, 64-Bit
```

Listing 1.1: Definition von Floating Point Variablen

Berechnungen sollten wenn immer möglich im Single Precision Format durchgeführt werden. FPUs rechnen normalerweise im Single Precision Format, sodass Berechnungen im Double Precision Format zusätzliche Libraries brauchen und damit Runtime kosten (siehe auch Kapitel 1.4.1). Ein häufig anzutreffender Fehler in der Softwareprogrammierung ist, dass ungewollt im Double Precision Format gerechnet wird, weil der Code falsch geschrieben wurde.

Beispiel für Single Precision Floating Point Berechnungen:

```
float x, y, z;
...
z = sinf(x) + cosf(y) + 1.0F;
```

Listing 1.2: Single Precision Floating Point Operationen

Beispiel für Double Precision Floating Point Berechnungen:

```
double x, y, z;
...
z = sin(x) + cos(y) + 1.0;
```

Listing 1.3: Double Precision Floating Point Operationen

Zu beachten sind insbesondere:

- Die Konstante 1.0 wird defaultmäßig vom Compiler als Double Precision interpretiert. Für Single Precision muss 1.0F geschrieben werden.
- Library-Funktionen sind defaultmäßig als Double Precision implementiert. Für Single Precision müssen die Funktionen sinf(), cosf() usw. verwendet werden.

1.5. DSP

1.5.1. Einsatzgebiete

DSP (Digital Signal Processor) werden für Berechnungen von mathematischen Algorithmen in zeitkritischen Anwendungen eingesetzt. Mögliche Einsatzgebiete sind beispielsweise:

- Berechnung von FIR oder IIR Filtern
- Berechnung von FFTs
- Audio und Video-Verarbeitungen allgemein
- Motor & Process Control

DSPs können viele mathematische Funktionen in Hardware ausführen, was die Berechnungszeit im Vergleich zu einer Software-Lösung (Library-Funktionen) massiv verkürzt. Typische Funktionen sind:

- Single Cycle MAC: Eine MAC (Multiply ACCumulate) Einheit kann in einem Taktzyklus eine Multiplikation mit anschliessender Addition durchführen. Dadurch lassen sich mathematische Ausdrücke wie $z = \sum_{k=0}^{N-1} x[k] * y[k]$ effizient berechnen. Weiter sind MAC-Einheiten in der Lage, verschiedene Operandengrößen zu unterstützen, z.Bsp. 16 Bit * 16 Bit, das Resultat in 64 Bit addiert.
- Saturation: Der maximal positive oder minimal negative Wert eines Resultates wird beschränkt (gesättigt). Das bedeutet, dass es bei einer arithmetischen Operation durch die Sättigung keinen Über- oder Unterlauf gibt. So ist beispielsweise der maximale Wert eines vorzeichenbehafteten Integers 32'667. Sollte eine 16-Bit Operation ein Resultat grösser als diesen Wert haben, wird er auf 32'667 beschränkt.
- SIMD: Single Instruction Multiple Data, siehe auch Kapitel 1.1.1. Viele DSPs sind in der Lage, SIMD-Operationen durchzuführen. Sind die Register des DSP 32-Bit breit, so können vier Operationen zu je 1 Byte oder zwei Operationen zu 2 Byte (Halfword) parallel durchgeführt werden.

- Circular Addressing: FIFOs werden im Speicher als Ringbuffer implementiert. Dazu werden Pointer verwendet, welche auf die aktuellen Schreib- und Lesepositionen zeigen. Der Umbruch (wrap around) der Pointer vom Ende des Buffers an den Anfang erfolgt üblicherweise in der Software. DSPs unterstützen mit Circular Addressing diesen Umbruch hardwaremäßig, was effizienter ist.
- Hardware Acceleration: Zusätzliche Hardware für die Beschleunigung der Berechnung von FIR-Filter, IIR-Filter und FFT.
- Zero Overhead loops: Zusätzliche Beschleunigung für die effiziente Durchführung von Loops.

1.5.2. Architekturen

Die Architektur von DSPs ist so ausgelegt, dass komplexe mathematische Operationen möglichst schnell aufgeführt werden können. DSP Architekturen besitzen folgende Eigenschaften:

- Verwendung einer Harvard-Architektur (siehe auch Kapitel 1.2.2).
- Parallelität in der Hardware. So ist es möglich, verschiedene Load / Store-Operationen (d.h. Daten vom Speicher lesen oder in den Speicher schreiben) sowie mathematische Operationen gleichzeitig und damit parallel auszuführen. Dazu wird der on-Chip Speicher in verschiedene Blöcke aufgeteilt und es existieren mehrere parallele Busse (Multibus), um gleichzeitig auf diese Speicherblöcke zuzugreifen. Die unterschiedlichen parallel ausgeführten Operationen führen auch dazu, dass die Instruktionen bei DSPs oft 256 Bit breit sind.
- Traditionelle DSPs sind auf hohe Rechenleistung optimiert. Das bedeutet auch, dass sie im Vergleich zu einer CPU relativ viel Energie brauchen.

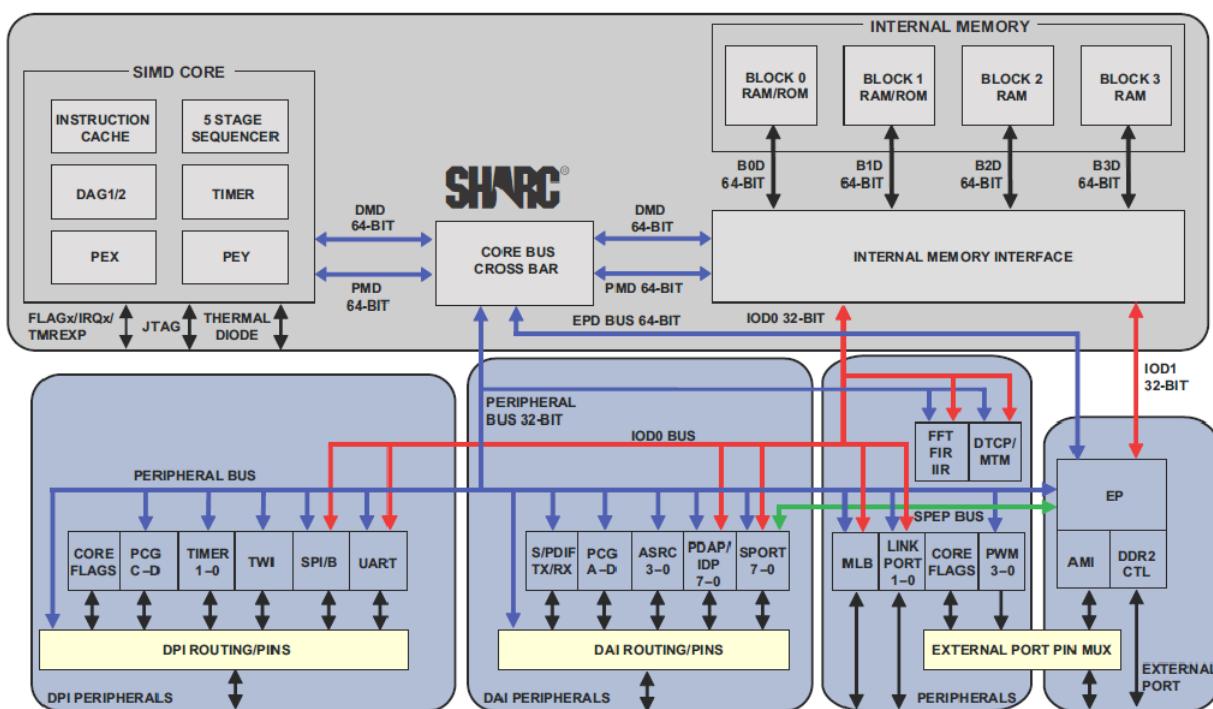


Abbildung 1.13.: Sharc ADSP21469 Blockdiagramm. Quelle: Datenblatt Analog Device ADSP-21467

Im Blockdiagramm des ADSP 21467 von Analog Device sind rechts oben die verschiedenen Memory-Blöcke für den parallelen Speicherzugriff gut zu erkennen.

1.5.3. Programmierung in C

C-Compiler sind nicht immer in der Lage, die Hardware eines DSPs effizient zu nutzen und erzeugen entsprechend nicht optimalen Maschinencode. Hier müssen dem Compiler Hilfestellungen programmiert werden. Generell sind folgende Varianten möglich:

1. Der Compiler ist selber in der Lage, die optimalen Instruktionen zu generieren.
2. Im C-Code werden Idiome verwendet, welche vom Compiler erkannt und in korrekte DSP-Instruktionen übersetzt werden.
3. Insbesondere für SIMD-Operationen müssen sogenannte Intrinsics verwendet werden. Bei Intrinsics (Intrinsic Functions) werden prozessorspezifische Instruktionen in Funktionen gekapselt. Die Verwendung von Intrinsics führt aber zu nicht oder schlecht portierbarem Code.
4. Implementation der Funktionalität in Assembler-Subroutinen.

Beispiele für C-Code für DSP-Operationen (aus [13]):

```

uint32_t x = 1;      // first operand
uint32_t y = 2;      // second operand
uint32_t z = 0;      // result
uint64_t acc = 0;    // result for accumulation

/* C-Compiler is able to choose the correct instructions without additional help */
/********************************************/

// SMLAL long signed multiply accumulate,
// multiplies two 32 bit integers and adds result to 64 bit accumulator
acc += (int64_t) x * y;

/* Using Idioms which map directly to DSP instructions *****/
/********************************************/

// SMMUL 32-bit multiply
// returning 32-most-significant-bits
z = (int32_t)((int64_t) x * y) >> 32;

/* Using Intrinsics to provide the compiler additional information *****/
/********************************************/

// QADD 32-bit saturating addition
// Adds two signed integers and saturates the result
z = __QADD(x, y);

// SADD8 quad 8-bit addition
// Adds four 8-bit values using SIMD
z = __SADD8(x, y);

// QADD16 dual 16-bit saturating addition
// Adds two 16-bit values using SIMD
z = __QADD16(x, y);

```

Listing 1.4: Programmierung von DSP-Operationen in der Programmiersprache C

1.6. Co-Prozessoren

Co-Prozessoren unterstützen die CPU und bieten zusätzliche Funktionalität. Sie sind entweder als zusätzlicher Chip auf einem Print zu finden, oder sie sind auf demselben Chip wie die CPU integriert.

Co-Prozessoren bieten folgende zusätzliche Funktionen:

- Floating Point Arithmetik (siehe FPU Kapitel 1.4)
- Digital Signal Processing (siehe DSP Kapitel 1.5)

- Grafikunterstützung
- Verschlüsselung
- Zugriff auf MMU, Cache usw.

1.7. Prozessor-Typen

Ein Prozessor ist ein Chip, auf welchem mindestens eine CPU integriert ist. Es gibt verschiedene Arten von Prozessoren: Mikroprozessoren, Microcontroller und Signalprozessoren (DSP). Oft wird der Ausdruck „Prozessor“ als Sammelbegriff für all diese Typen verwenden.

1.7.1. Prozessoren und Mikroprozessoren

Ein Mikroprozessor ist ein Chip, auf welchem eine CPU mit Rechenwerk und Steuerwerk integriert ist. Alle Speicher und Peripherie-Elemente sind extern und werden über den Bus angesprochen. Leistungsfähige Prozessoren, welche vor allem im PC-Bereich eingesetzt werden, haben zusätzlich eine MMU integriert (siehe Kapitel 3.3). In diesem Falle spricht man oft von Prozessoren, d.h. der Begriff „Mikro“ entfällt.

1.7.2. Microcontroller

Microcontroller sind für kostengünstige industrielle Anwendungen ausgelegt. Zusätzlich zur CPU beinhalten sie immer auch Speicher für Daten und Programminstruktionen sowie Peripherie-Elemente.

Typische Einsatzgebiete von Microcontrollern sind:

- Kommunikationssysteme
- Medizinaltechnik
- Energietechnik
- Sicherheitstechnik
- Mechatronics und Industrieautomation
- Automotive
- Consumer Electronics

Die Peripherie-Elemente sind je nach Microcontroller-Familie unterschiedlich und ermöglichen die Kommunikation mit dem Anwender, mit anderen Geräten oder mit Sensoren und Aktoren. Dies macht Microcontroller insbesondere für Embedded Applikationen zu einem wichtigen zentralen Element. Mögliche Peripherie-Elemente sind:

- Kommunikation (RS232, SPI, I2C, CAN, Ethernet, USB ...)
- Digitale Input / Output
- Analog/Digital und Digital/Analog Wandler
- Watchdog
- Timer
- usw.

Viele Microcontroller sind auf möglichst tiefen Energieverbrauch und tiefe Preise optimiert. Sie unterstützen nur teilweise Floating-Point und DSP-Funktionalität.

1.7.3. DSP

Signalprozessoren (DSP) sind für rechenintensive Aufgaben ausgelegt. Mehr dazu in Kapitel 1.5.

1.7.4. CISC / RISC

CISC und RISC sind zwei unterschiedliche Ansätze für die Definition des Instruktionssatzes. Sie sind sowohl bei Prozessoren als auch bei Microcontrollern anzutreffen.

Complex Instruction Set Computer (CISC): Die Befehle können relativ komplex sein. Eine einzelne Assembler-Instruktion wird in eine Folge von Mikrobefehlen zerlegt. Die Ausführung einer ganzen Instruktion benötigt mehrere Taktzyklen. Das entsprechende Mikroprogramm wird vom Chiphersteller im Mikroprogramm-Speicher auf dem Silizium integriert, der Anwender hat keinen Zugriff darauf.

Reduced Instruction Set Computer (RISC): Die Befehle sind sehr einfach. Es gibt keine Mikrobefehle. Eine Instruktion wird in der Regel in einem einzigen Taktzyklus ausgeführt.

1.8. Beispiele von Microcontroller-Familien

In den nachfolgenden Unterkapiteln sollen einige Microcontroller-Familien kurz vorgestellt werden. Die Microcontroller unterscheiden sich einerseits durch die Leistungsfähigkeit der CPU, andererseits durch die Integration von Speichern und Peripherie:

- Speicher (Flash, RAM, EEPROM)
- Anzahl GPIO
- Timer / PWM / Watchdog
- Serielle Schnittstellen (RS232, SPI, I2C, Ethernet, USB, CAN)
- A/D-Wandler, D/A-Wandler

Tabelle 1.2 zeigt eine Übersicht einiger Microcontroller-Familien. Sie erhebt keinen Anspruch auf Vollständigkeit.

Familie	Kurzbeschreibung
8051	Eine Familie aus der Steinzeit, aber immer noch eine der bekanntesten und weit verbreitetsten 8-Bit-Familien im Controller-Bereich überhaupt. Verschiedene Hersteller liefern Derivate mit unterschiedlicher integrierter Peripherie, auch heute noch. Als Urvater der Familie gilt der 8031 von INTEL.
PIC	8-Bit, 16-Bit und 32-Bit Controller-Familie von Microchip. Auch diese Familie besitzt Mitglieder aus den Anfangszeiten der Microcontroller, mittlerweile durch modernere Designs ergänzt. Wird vor allem im low-cost oder low-power Bereich eingesetzt.
Coldfire	Die Coldfire-Familie ist die Nachfolgerfamilie der MC683xx von Motorola. Coldfire sind reine 32-Bit Maschinen, d.h. haben 32 Bit Datenbus und 32 Bit Adressbus. Sie sind noch wesentlich leistungsfähiger wie die 683xx-Familie, sind von der Architektur her aber sehr ähnlich aufgebaut.
MSP430	16-Bit RISC Controller-Familie von TI, die vor allem in Low-Power Applikationen eingesetzt wird (z.B. batteriebetriebene Geräte).
AVR	Verschiedene Controller-Familien der Firma Atmel wie AVR (8-Bit RISC Prozessor), AVR32 (32-Bit RISC Architektur mit DSP und SIMD-Funktionalität) oder AT91SAM (sehr preisgünstige Microcontroller basierend auf der ARM7-Architektur). Diese Controller hatten in den letzten Jahren einen grossen Zuwachs im Embedded Bereich.
ARM Cortex-Mx	Core der Firma ARM für diverse Einsatzgebiete von einfacheren Embedded Systems. Heute sehr verbreitet sind ARM Cortex-M0, Cortex-M3 und Cortex-M4, sowie Cortex-M7. Diverse Chip-Hersteller haben diesen Core lizenziert und bieten Silizium mit unterschiedlichen Speicher und Peripherie-Elementen. Mehr dazu finden Sie im Kapitel 4.2.
ARM Cortex-Ax	Core der Firma ARM für diverse Einsatzgebiete von leistungsfähigen Embedded Systems wie Motor Control, Smartphones, Tablets usw. Auch hier gibt es viele Chip-Hersteller, welche diesen Core lizenziert haben und Silizium anbieten.

Tabelle 1.2.: Verschiedene Microcontroller-Familien

1.9. Auswahlkriterien

Für die Auswahl des geeigneten Prozessors / Controllers spielen viele Kriterien eine Rolle. Je nach Projekt werden diese Kriterien unterschiedlich gewichtet:

- Rechenleistung (MIPS, FLOPS, FPU, DSP)
- On-Chip Memory (Flash, RAM)
- Integrierte Peripherie und Interfaces (Timer, UART ...)
- Busbreite (Datenbus, Adressbus, CPU-Register)
- Energieverbrauch (mA/MHz, Standby-Modi, Sleep-Modi)
- Speisung und Umgebungsbedingungen (Betriebsspannung, Temperatur, EMV)
- Preis und Verfügbarkeit
- Gehäuse (Anzahl Pins, Grösse)
- Verfügbarkeit von Software-Libraries

- Verfügbarkeit von Entwicklungsumgebungen (IDE) und Evaluationsboards
- Qualität der Dokumentation (Datenblätter, Application Notes)
- Entwicklungsunterstützung durch den Hersteller
- Verfügbarkeit von Second-Source oder von ähnlichen Chips derselben Familie

Oft spielt bei der Auswahl auch das vorhandene Know-How der Entwickler*innen sowie bereits vorhandene Entwicklungs- und Testwerkzeuge eine nicht unbedeutende Rolle.

2. Bussysteme und Speicher

2.1. Bus-Systeme

2.1.1. Adress-, Daten- und Control-Bus

Verschiedene Busse verbinden die CPU mit dem Speicher und der Peripherie. Über diese Busse werden Adressen, Daten und Kontrollinformationen ausgetauscht („von-Neumann“ Architektur).

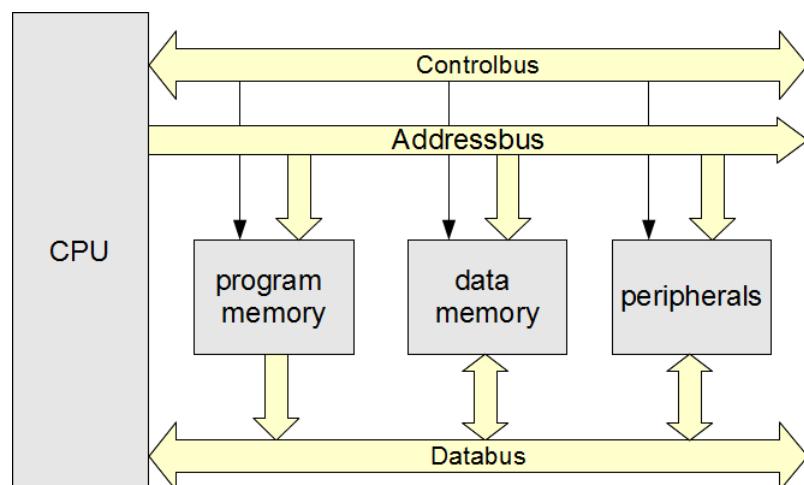


Abbildung 2.1.: Address-, Data- und Controlbus

Auf dem Adressbus werden Adressinformationen für die angeschlossenen Speicher oder die Peripherie übertragen (z.B. Adressierung einzelner Speicherzellen im Datenspeicher). Die Adressen werden vom Steuerwerk der CPU generiert. Oft werden zusätzliche Adressdecoder verwendet, um die einzelnen Einheiten zu selektieren (siehe Kapitel 2.1.3).

mit wem?

Auf dem Datenbus werden Programminstruktionen und Benutzerdaten übertragen. Diese können

was?

1. von der CPU aus dem Speicher oder der Peripherie gelesen werden.
2. von der CPU generiert und anschliessend im Speicher oder der Peripherie gespeichert werden.

Auf dem Controlbus (Steuerbus) werden zusätzliche Steuerinformationen übertragen (Read/Write, Reset, Interrupts, Requests und Acknowledge, Handshake ...).

wie?

Tabelle 2.1.: Adress- Daten- und Controlbus

Beispiel 1. Temperatur aus einem Sensor lesen.

Die Funktionsweise des Bussystems soll anhand eines Datentransfers von der Peripherie über die CPU in den Datenspeicher genauer betrachtet werden. Dazu soll die Temperatur von Sensor x (Peripherie) gelesen und anschliessend in der Variablen „temp“ im RAM gespeichert werden.

In C-Code würde man also schreiben:

```
int temp = Temperatur_Sensor_x;
```

Listing 2.1: Beispiel Temperatur eines Sensors auslesen

Zwischen CPU, Temperatursensor und Speicher gibt es nun folgende Busaktivitäten:

1. Die CPU legt die Adresse der nächsten auszuführenden Programminstruktionen auf den Adressbus. Diese Adresse ist im Program-Counter gespeichert.
2. Die CPU definiert über den Controlbus die Transferrichtung „lesen“.
3. Der Programmspeicher gibt die an der adressierten Speicherzelle liegende Instruktion auf den Databus.
4. Die CPU liest die auszuführende Instruktion ein.
5. Die CPU legt die Adresse der Peripherie auf den Adressbus, von der die Daten gelesen werden sollen.
6. Die CPU definiert über den Controlbus die Transferrichtung „lesen“.
7. Die Peripherie gibt die an der adressierten Zelle liegende Information auf den Databus.
8. Die CPU liest diese Daten ein.
9. Die CPU adressiert die Zieladresse im Datenspeicher über den Adressbus.
10. Die CPU legt die Daten auf den Databus
11. Die CPU definiert über den Controlbus die Transferrichtung „schreiben“.
12. Der Datenspeicher übernimmt die Daten vom Databus und speichert sie in die vom Adressbus adressierte Speicherzelle.

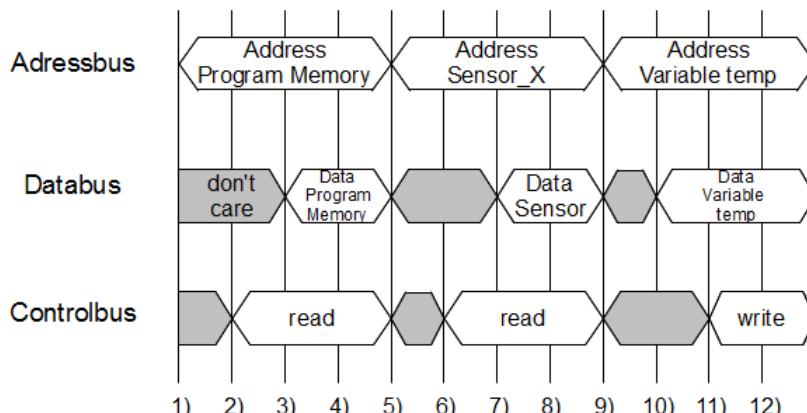


Abbildung 2.2.: Bustiming

In Abbildung 2.2 werden folgende Notationen für die Buspegel verwendet:

-
- Der Buspegel ist entweder high oder low, jedoch immer stabil und gültig
 - Der Buspegel ist irrelevant („don't care“)
 - Der Buspegel wechselt

2.1.2. Memory-Map

Die Memory-Map (Speicherbelegungsplan) gibt an, auf welchen Adressen Speicher und Peripherie liegen.

Beispiel 2. Einfache Memory-Map eines Systems mit 16-Bit Adressbus und 64 kByte Adressraum

	Address	A15	A14	A13	A12	A11	A0
Periphery	0xFFFF	1	1	1	1			
	0xE000	1	1	1	0			
Not used								
RAM2 (8 kByte)	0xBFFF	1	0	1	1			
RAM1 (8 kByte)	0xA000	1	0	1	0			
	0x9FFF	1	0	0	1			
Flash (32 kByte)	0x8000	1	0	0	0			
	0x7FFF	0	1	1	1			
	0	0	0	0	0			

Abbildung 2.3.: Memory-Map eines 16-Bit Adressbusses

8-Bit Microcontroller (8-Bit gibt hier die Breite des Datenbusses an) haben normalerweise einen 16-Bit breiten Adressbus (A0 bis A15) und können somit 2^{16} Adressen (64 kByte, Adressen 0 bis 0xFFFF) Speicher und Peripherie adressieren. 32-Bit Microcontroller haben einen 32-Bit breiten Adressbus und können somit 2^{32} Adressen (4 Giga) ansprechen. Die Grösse der verwendeten Speicher ist abhängig vom Typ des Microcontrollers. So werden bei 8-Bit Microcontrollern Speicher in der Größenordnung von einigen Kilobytes eingesetzt, was genügend ist, weil auf solchen Systemen in der Regel kein Betriebssystem und nur einfache Applikationen laufen. Bei 32-Bit Microcontrollern werden für Embedded Systems je nach Einsatzgebiet bis zu einigen MByte Speicher verwendet.

2.1.3. Adressdecoder

Die Aufgabe des Adressdecoders ist es, einzelne Bausteine oder Baugruppen zu selektieren. Sollen beispielsweise mehrere RAM am selben Adressbus betrieben werden, so stellt sich das Problem, dass die einzelnen Bausteine nicht nur durch die Adressleitungen selektiert werden können, sondern ein zusätzliches Selektierungs-Signal (Chip Select, CS) benötigen. Dieses CS-Signal wird vom Adressdecoder geliefert, indem er die höchstwertigen Adressleitungen logisch verknüpft und daraus die verschiedenen CS-Signale generiert.

Einfache Adressdecoder können mit diskreten Logikbausteinen aufgebaut werden (AND, OR, 1 of n Decoder). Die erforderlichen Bausteine sind sehr preiswert. Dafür braucht man relativ viel Platz auf dem Print und vor allem ist die diskrete Logik sehr unflexibel: Bei Fehlern oder bei Anpassungen der Memory-Map muss ein neuer Print hergestellt werden.

Wegen den oben beschriebenen Problemen der diskret aufgebauten Adressdecoder werden für die Adress-Decodierung häufig programmierbare Logikbausteine (GAL, PAL, CPLD ...) eingesetzt. Der Adressdecoder für die Memory-Map aus Abbildung 2.3 könnte wie folgt realisiert werden:

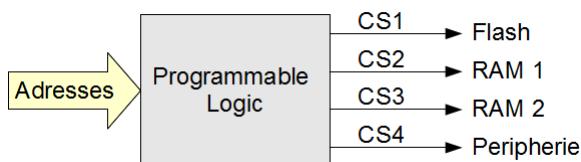


Abbildung 2.4.: Programmierbarer Adressdecoder

Für die Programmierung der Chip-Selects werden logische Gleichungen verwendet. Die CS-Signale für die Memory-Map aus Abbildung 2.3 werden wie folgt programmiert:

```
CS1 = /A15
CS2 = A15 * /A14 * /A13
CS3 = A15 * /A14 * A13
CS4 = A15 * A14 * A13
```

Listing 2.2: Beispiel Logische Gleichungen für die Generierung von Chip-Select Signalen

Beispiel 3. Blockschaltbild mit Adressdecoder.

Das Blockschaltbild in Abbildung 2.5 mit Adressdecoder, Speicher und Peripherie (32k Flash, 2 * 8k RAM sowie A/D-Wandler mit 1 Byte) für das Memory-Map aus Abbildung 2.3:

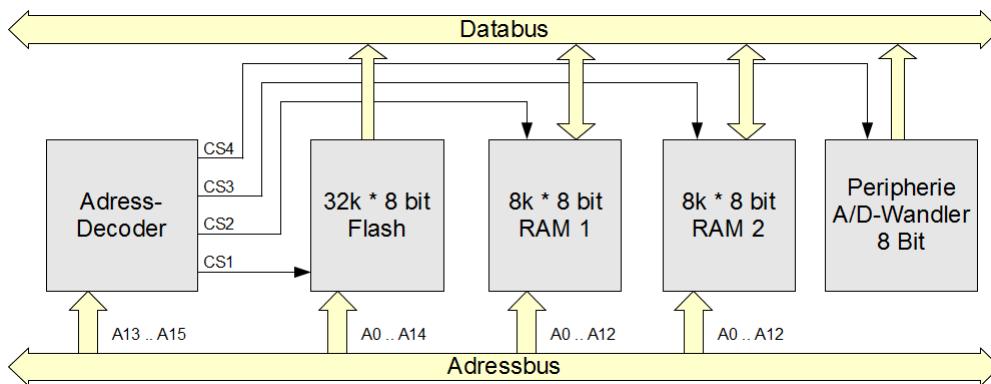


Abbildung 2.5.: Blockschaltbild mit Adressdecoder, Speicher und Peripherie

Bemerkungen:

Im Blockschaltbild sind die Steuerleitungen /RD und /WR nicht aufgeführt. Der A/D-Wandler wird in der obigen Schaltung über das Chip-Select CS4 angesprochen. Da er physikalisch über nur eine Adresse angesprochen werden kann, wird er somit im ganzen Memory-Bereich der Peripherie „gespiegelt“, das heisst er kann von der CPU auf irgend einer Adresse zwischen 0xE000 und 0xFFFF angesprochen werden.

2.2. Speicher

2.2.1. Technologien

Grundsätzlich wird zwischen nichtflüchtigen Festwertspeichern und flüchtigen Schreib-Lese-Speichern unterschieden.

Nichtflüchtige Speicher

Der Inhalt nichtflüchtiger Speicher bleibt auch nach dem Ausschalten der Versorgungsspannung erhalten. Speichertypen dieser Kategorie sind:

- ROM (Read Only Memory): Wird beim Chiphersteller maskenprogrammiert.
- EPROM (Erasable Programmable ROM): Kann vom Anwender mehrmals programmiert (spezielle Programiergeräte) und mit UV-Licht wieder gelöscht werden.
- OTP (oder OTP ROM, One Time Programmable ROM): Aufgebaut wie ein EPROM, jedoch ohne Fenster. Kann vom Anwender genau 1 mal programmiert werden.

- EEPROM (Electrically Erasable Programmable ROM): Kann während dem Betrieb zellenweise elektrisch gelöscht und neu programmiert werden. EEPROM werden beispielsweise für Produktionsdaten wie Seriennummern oder das Speichern von kleineren Datenmengen in batteriebetriebenen Datenloggern verwendet.
- Flash (genaue Bezeichnung FLASH EEPROM): Kann während dem Betrieb page-weise elektrisch gelöscht werden (es können nicht einzelne Zellen wie beim EEPROM gelöscht werden). Sehr grosse Speicherkapazität (einige MByte). Flash-Bausteine werden vor allem für Programmcode verwendet.
- FRAM (Ferroelectric Random Access Memory): Die einzelnen Informationen werden in Kristallen mit ferro-elektrischen Eigenschaften gespeichert. Der Aufbau entspricht etwa einer DRAM-Zelle, die Zugriffszeiten sind vergleichbar mit RAM-Technologien, die Informationen werden jedoch nichtflüchtig gespeichert.

Flüchtige Speicher

Der Inhalt flüchtiger Speicher geht nach dem Ausschalten der Versorgungsspannung verloren. Diese Speicher werden allgemein als RAM (Random Access Memory) bezeichnet. Random steht für den wahlfreien Zugriff auf den Inhalt von Speicherzellen, im Gegensatz zum sequenziellen Zugriff wie beispielsweise bei FIFOs. Man unterscheidet folgende Kategorien:

- DRAM (Dynamic Random Access Memory): Diese halten die Information in einer Kapazität und müssen periodisch aufgefrischt werden.
- SRAM (Static RAM): Sind aufwändiger aufgebaut, benötigen aber keinen Auffrisch-Zyklus. SRAM's sind schneller als DRAM's, brauchen weniger Strom, sind aber teurer.
- SDRAM (Synchronous Dynamic RAM): Sie sind eine getaktete Variante der DRAMs. Der Takt wird durch den Systembus vorgegeben. Bei DDR-SDRAM (Double Data Rate SDRAM) sind Speicherzugriffe sowohl bei der steigenden als auch bei der fallenden Flanke des Taktes möglich.

Heute wird teilweise auch mit nichtflüchtigen RAM-Technologien gearbeitet (FRAM, MRAM), diese haben jedoch eine kleine Speicherkapazität und sind relativ teuer.

2.2.2. Organisation

Speicher werden als Array von einzelnen Speicherwörtern organisiert. Speicherwörter sind je nach Speichertyp 1, 8 oder 16 Bit breit.

Die folgende Abbildung zeigt den Aufbau eines byte-organisierten Speichers:

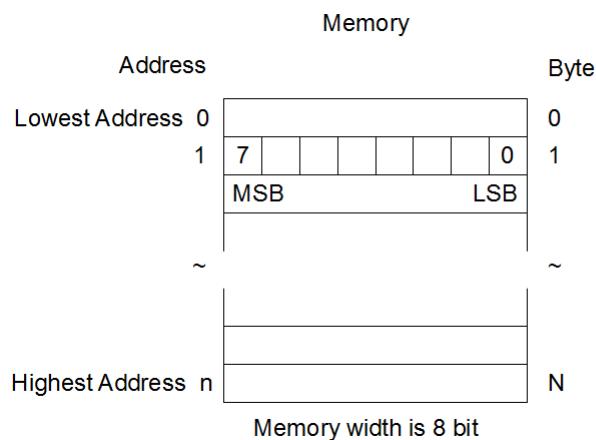


Abbildung 2.6.: Aufbau eines byte-organisierten Speichers

Die Anzahl der Adressleitungen, die gebraucht werden, um den gesamten Speicher zu adressieren, ist abhängig von der Speichergrösse. Mit „ m “ Adressleitungen können 2^m Speicherwörter adressiert werden. Mit 16 Adressleitungen kann somit ein Speicherbereich von 64k angesprochen werden (dies ist oft bei 8-Bit Microcontrollern der Fall).

2.2.3. Architekturen

Speicher sind intern so aufgebaut, dass sie die Information in einem Array von Speicherzellen (Memory-Matrix) ablegen. Die Adressierung der einzelnen Speicherzellen erfolgt über Adressdecoder. Steuerleitungen wie CS, OE, R/W usw. kontrollieren die Bustreiber.

Architektur eines ROM-Bausteins

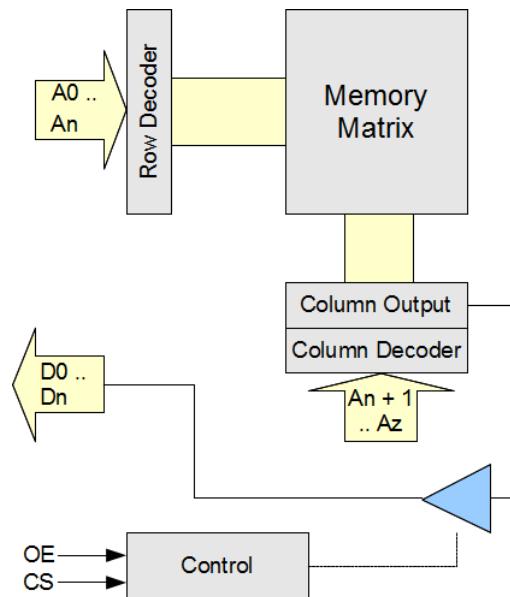


Abbildung 2.7.: Architektur eines ROM-Bausteins

ROM-Bausteine haben typischerweise einen Datenbus der Breite 8 oder 16 Bit. Die Breite des Adressbusses ist abhängig von der Speicherkapazität. Als Steuerleitungen wird ein CS (Chip Select) zur Selektion des Bausteins sowie ein OE (Output Enable) zum aktivieren der Ausgangstreiber für den Datenbus verwendet.

Abbildung 2.7 berücksichtigt nur den lesenden Zugriff auf den ROM-Baustein. Für die Programmierung werden zusätzliche Steuerleitungen, allenfalls eine separate Programmierspannung sowie der Datenfluss zur Memory-Matrix benötigt.

Architektur eines SRAM-Bausteins

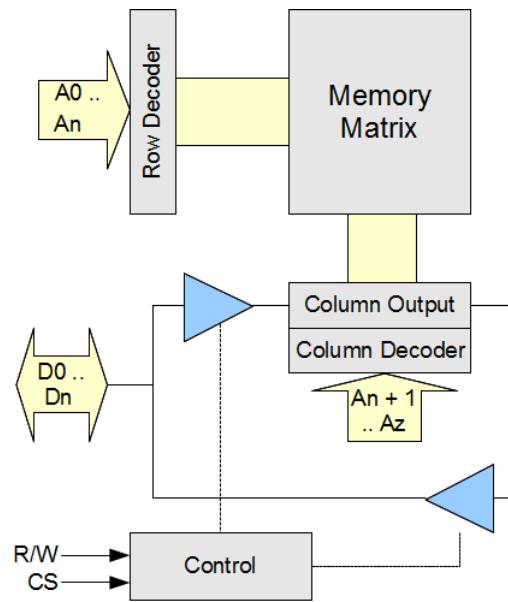


Abbildung 2.8.: Architektur eines SRAM-Bausteins

SRAM-Bausteine sind ähnlich aufgebaut wie ROM-Bausteine. Der Datenbus ist bidirektional, so dass die CPU lesen und schreiben kann. Als Steuerleitung wird zusätzlich ein R/W-Signal (Read/Write) verwendet. Bei einzelnen Herstellern wird dieses Signal in zwei unterschiedlichen Steuerleitungen (Read und Write) aufgeteilt.

Architektur eines SDRAM-Bausteins

Bei DRAM und SDRAM-Bausteinen wird nicht die gesamte Adresse gleichzeitig am Bus angelegt. Um die Anzahl der Pins zu verringern wird die Adresse in zwei Schritten übergeben, zuerst die row-address und anschliessend die column-address. Um dem Baustein zu signalisieren, welche der Adressen aktuell am Bus anliegt, werden zwei zusätzliche Steuersignale benötigt. Dies sind row-address-strobe (RAS) und column-address-strobe (CAS).

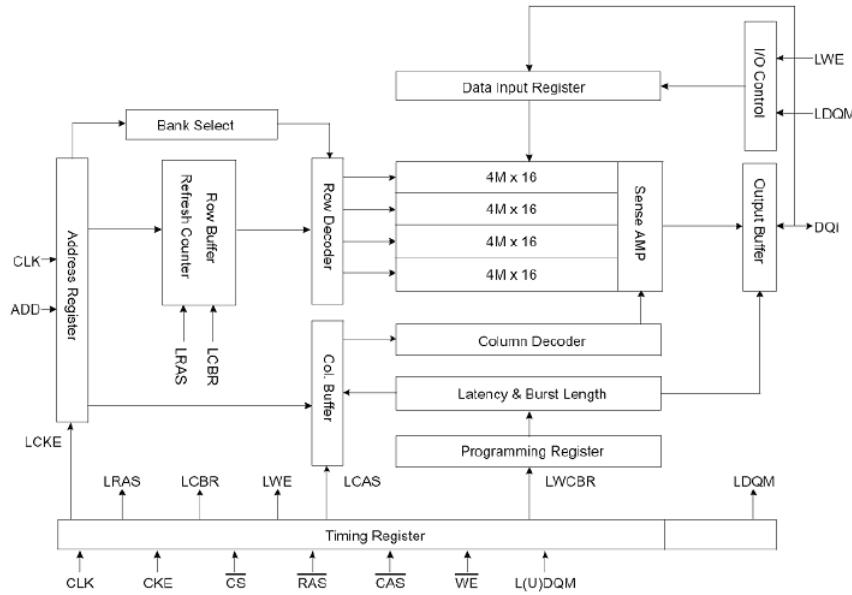


Abbildung 2.9.: Block-Diagramm des SDRAMs K4S561633F von Samsung (4M * 16 Bit * 4 Banks) Quelle: [Samsung SDRAM]

Der zeitliche Verlauf eines Lesezugriffs auf das SDRAM sieht vereinfacht wie folgt aus:

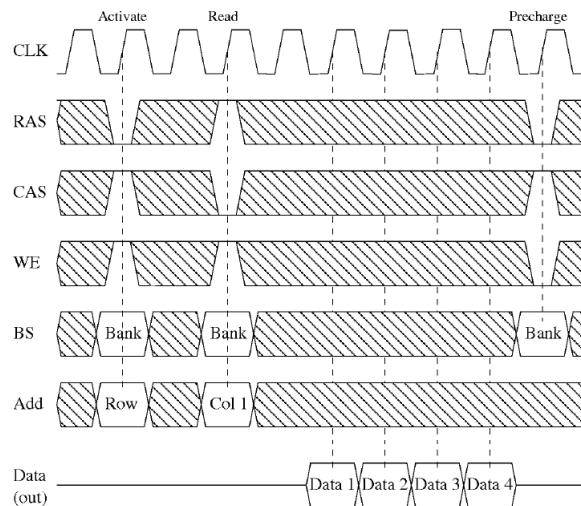


Abbildung 2.10.: Zeitlicher Verlauf eines Lesezugriffs auf das SDRAM

In Abbildung 2.10 ist zu erkennen, dass auf dem Bus zuerst die Row-Adresse angelegt wird (RAS-Signal ist aktiv tief). Danach wird die Column-Adresse angelegt, das CAS-Signal ist aktiv tief. Anschliessend können mehrere aufeinanderfolgende Daten gelesen oder geschrieben werden (Burst-Mode).

2.2.4. Datenblätter

Aus den Datenblättern der Speicherhersteller gehen die wichtigsten Charakteristiken der Speicher hervor. Dies sind insbesondere:

- Speichergrösse (in kBit/kByte oder Mbit/MByte)
- Pinbelegung

- Speicherorganisation (Bit, Byte oder Word)
- DC-Charakteristiken (Spannungen, Stromverbrauch ...)
- AC-Charakteristiken (Zugriffszeiten...)
- Wahrheitstabellen (Steuerlogik)

2.3. Speicher Hierarchy

Jede Speichertechnologie hat ihre Vor- und Nachteile. Speicher werden deshalb hierarchisch organisiert:

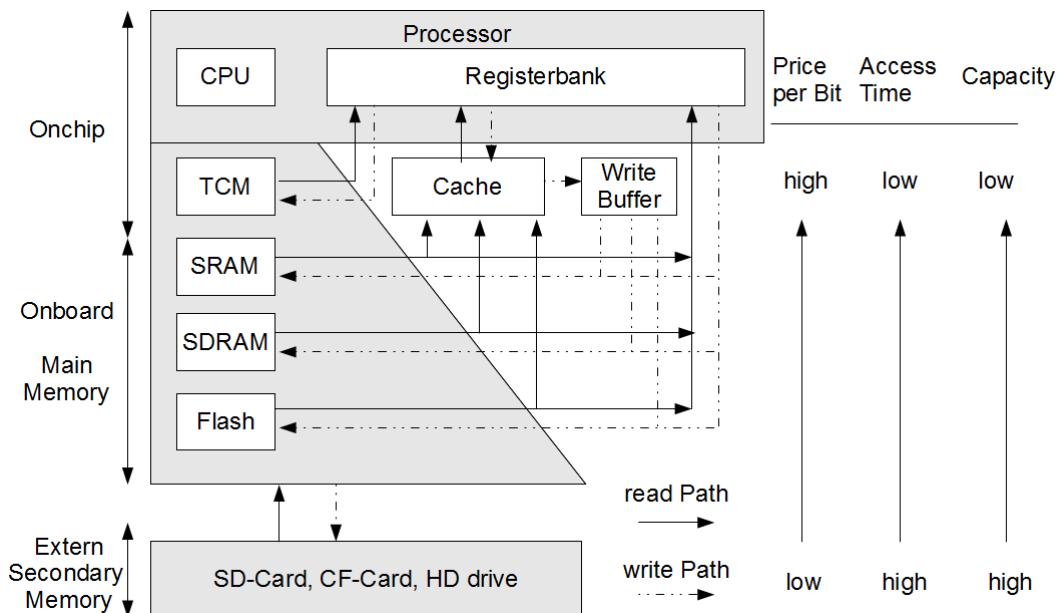


Abbildung 2.11.: Speicherhierarchie

Register

Sehr schneller Zwischenspeicher für Operanden und Resultate der ALU, direkter Zugriff von der CPU (siehe auch Kapitel 1.3). Pro CPU hat es in der Regel nur sehr wenige Register.

TCM

TCM (Tightly Coupled Memory): Zusätzlicher, schneller Speicher auf dem Silizium der CPU .

Cache

Ein Cache ist ein schneller Zwischenspeicher zwischen Registern und Main Memory für häufig benötigte Daten (Data-Cache) und Befehle (Instruction-Cache), siehe auch Kapitel 3.1.

Onboard Main Memory

Onboard Main Memory oder Arbeitsspeicher wird verwendet, um Programmcode (Flash) und Daten (SRAM, SDRAM) abzulegen, siehe auch Kapitel 2.2. In der PC-Welt wird der Programmcode vom Extern Memory (typisch Harddisk) ins RAM kopiert und aus dem RAM ausgeführt. In der Embedded-Welt wird der Programmcode entweder aus dem Flash ausgeführt oder beim Bootvorgang vom Flash ins RAM kopiert und aus dem RAM ausgeführt.

Extern Secondary Memory

Extern Memory oder Massenspeicher wie Festplatten oder Memory-Sticks werden in kleineren Embedded Systems selten eingesetzt. In Industrie-PCs (IPC) sind sie jedoch anzutreffen. Allerdings muss man die Nachteile der Mechanik in industriellen Umgebungen beachten (Temperatur, Erschütterungen, Feuchtigkeit, Staub usw.). Deshalb werden hier oft CF-Cards, SD-Cards oder Memory-Sticks als Speichermedium verwendet.

3. Cache, MPU, MMU, DMA

3.1. Cache

3.1.1. Einleitung

Leistungsfähige CPUs können mit wesentlich höheren Taktraten rechnen als es die Zugriffszeiten des Main Memories erlauben. Um Wartezyklen (wait cycles) der CPU beim Zugriff auf den Speicher zu vermeiden, wird der Cache verwendet. Der Cache erhöht dadurch die Performance des Systems. Ein Cache ist ein sehr schneller Speicher, welcher physikalisch zwischen der CPU und dem Main Memory liegt.

Abbildung 3.1 zeigt den Unterschied zwischen einem System ohne Cache und einem System mit Cache.

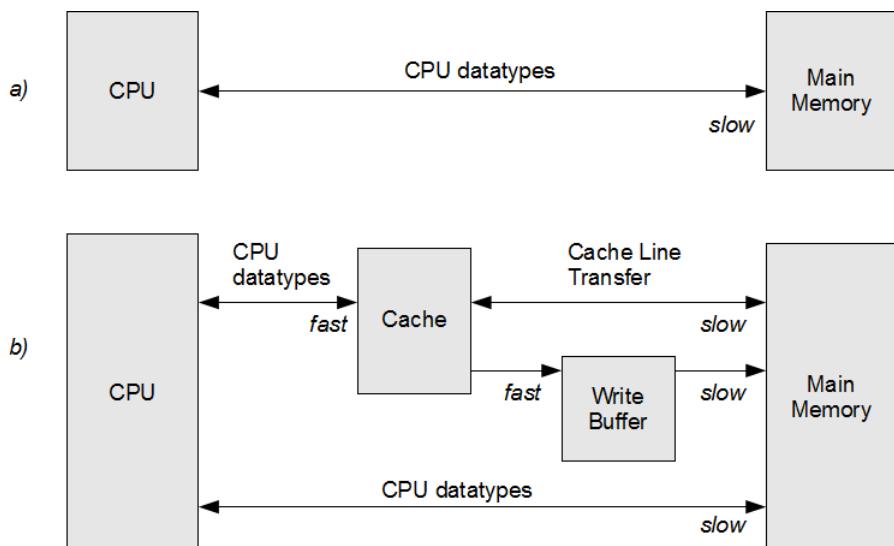


Abbildung 3.1.: a) System ohne Cache und b) System mit Cache

Bei Systemen ohne Cache wird das Main Memory direkt von der CPU angesprochen (siehe Abbildung 3.1 a). Dabei ist der Speicherzugriff der Flaschenhals: Die CPU muss warten, bis die Daten gelesen oder geschrieben sind.

Bei Systemen mit Cache werden Daten und Programmcode temporär im Cache gespeichert (siehe Abbildung 3.1 b). Wenn Daten oder Programmcode erstmals gelesen werden, werden sie vom Main Memory in den Cache kopiert. Wenn Daten oder Programmcode ein weiteres Mal gebraucht werden, erfolgt der Zugriff von der CPU auf den Cache.

Software-Programme führen sehr oft Programm-Loops aus (for, while ...) oder arbeiten mit Datenstrukturen (z.B. Array). Ein System mit Cache arbeitet deshalb sehr effizient bei der Ausführung von Schleifen oder bei der Verarbeitung von Datenstrukturen.

Der Cache-Vorgang ist für den Anwender transparent, d.h. die Hardware (Cache Controller) übernimmt die Steuerung des Transfers zwischen Main Memory, Cache Memory und CPU. Das heißt auch, dass Programmcode Cache-unabhängig ist, und somit für ein System mit Cache nicht neu geschrieben und übersetzt werden muss. Programmierer*innen können aber durch Kenntnisse der Details die Programmausführung optimieren, indem sie etwa Datenstrukturen und Programmschleifen auf die Cache-Größe anpassen.

Beim Cache-Vorgang (Caching) werden kleine Blöcke von Programmcode oder Daten zwischen Main Memory und Cache Memory transferiert. Diese Blöcke werden auch „Cache Lines“ genannt und haben typischerweise einige wenige Bytes (16, 32)

Sind Daten oder Programmcode, welche von der CPU gelesen werden, nicht im Cache vorhanden, so spricht man von einem „Cache Miss“. Diese Daten werden dann aus dem Main Memory gelesen und in einer „Cache Line“ abgespeichert. Ist der Cache voll, so müssen bestehende „Cache Lines“ überschrieben werden. Sind Daten, welche gelesen werden, im Cache vorhanden, so spricht man von einem „Cache Hit“. Der Zugriff auf diese Daten ist jetzt wesentlich schneller als wenn diese aus dem Main Memory gelesen werden müssten.

Die Ausführungszeit von Programmcode in einem System mit Cache hängt davon ab, ob der auszuführende Code im Cache schon vorhanden ist, oder ob er zuerst geladen werden muss. Dies macht die Ausführungszeit nicht-deterministisch, was insbesondere bei zeitkritischen Systemen ein Nachteil sein kann. Die gesamte System-Performance wird aber durch einen Cache immer erhöht.

Die Speichergrösse eines Caches (Cache Memory) beträgt typischerweise zwischen einigen kByte bis einige MByte. Microcontroller mit tieferen Taktraten (< 200 MHz) arbeiten ohne Cache (z.B. ARM Cortex-M3 / M4). Microcontroller mit höheren Taktraten (z.B. ARM Cortex-M7 oder ARM Cortex-Ax) verwenden einen Cache.

Caches werden oft in mehreren Levels organisiert (1, 2 oder 3 Level). Auf dem Prozessor ist ein sehr schneller aber kleiner Level 1 Cache integriert (L1), in einem externen aber sehr schnellen SRAM der zweite Level L2. L1 wird auch „Primary Cache“ genannt, L2 „Secondary Cache“.

3.1.2. Cache Architekturen

Die „von-Neumann“ Architektur (siehe auch Kapitel 1.2.1) verwendet einen gemeinsamen Bus um Programmcode und Daten zu adressieren. Die „Harvard“ Architektur (siehe auch Kapitel 1.2.2) verwendet unterschiedliche Busse für Programmcode und Daten. Es gibt zwei unterschiedliche Cache-Designs, welche diese Architekturen unterstützen.

Prozessoren, welche auf der „von-Neumann“ Architektur basieren, verwenden einen gemeinsamen Cache für Programmcode und Daten. Dieser Typ von Cache wird auch „Unified Cache“ genannt. Prozessoren, welche die „Harvard“ Architektur unterstützen, verwenden zwei Caches: Einen Instruktions-Cache (I-Cache) und einen Data-Cache (D-Cache). Dieser Typ von Cache wird auch „Split Cache“ genannt.

Ein Cache besteht aus zwei Elementen: Einem Cache Memory und einem Cache Controller. Beide werden nachfolgend beschrieben.

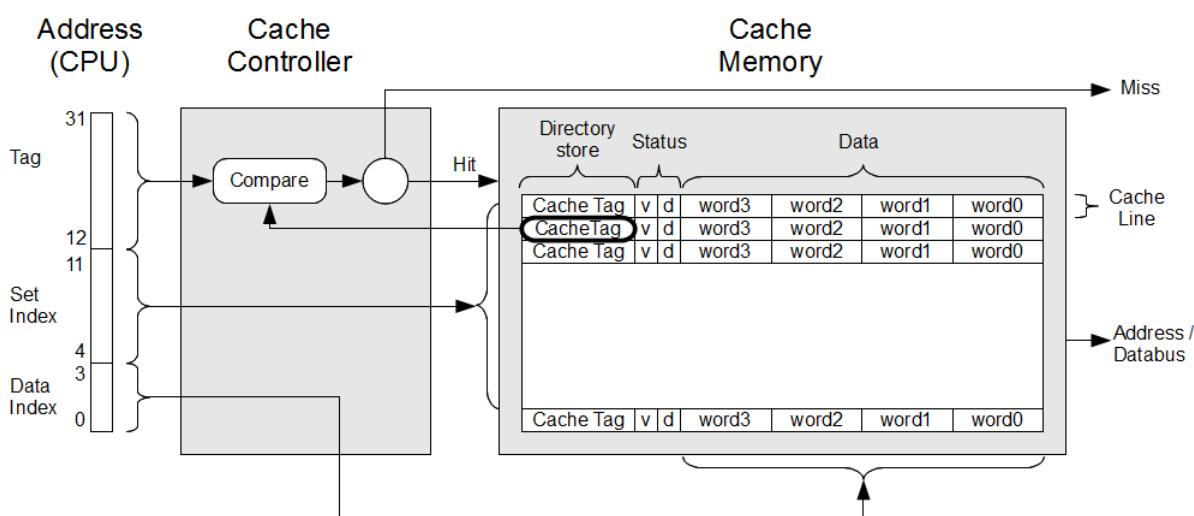


Abbildung 3.2.: 4KB Cache bestehend aus 256 Cache Lines zu je vier 32-bit Words

3.1.3. Cache Memory

Das Cache Memory besteht aus den folgenden Elementen:

- Directory Store
- Status Information
- Data Section

Im Directory Store wird eine Adresse abgelegt, welche angibt woher die Information in der Cache Line aus dem Main Memory kopiert wurde. Diese Adressen werden auch Cache Tags genannt. Auf diese Weise kann der Cache kontrollieren, woher die Informationen in einer Cache Line ursprünglich stammen.

Die Status Information Bits zeigen den Zustand jeder Cache Line an. Es gibt zwei übliche Statusbits:

- Das Valid Bit zeigt, ob die Cache Line aktiv ist, d.h. ob die Information in dieser Cache Line gültig ist oder nicht.
- Das Dirty Bit zeigt, ob Daten in einer Cache Line von der CPU modifiziert wurden, und somit nicht identisch sind mit dem Inhalt im Main Memory.

In der Data Section werden Programmcode und Daten aus dem Main Memory abgelegt. Die Cachegröße definiert die Anzahl Bytes Programmcode und Daten, welche in der Data Section des Caches gespeichert werden können. Directory Store und Status Information werden nicht zur Cachegröße gerechnet.

3.1.4. Cache Controller

Der Cache Controller ist in Hardware implementiert. Er kopiert Programmcode und Daten vom Main Memory ins Cache Memory falls erforderlich. Der Cache Controller führt seine Arbeit automatisch im Hintergrund aus, d.h. er ist transparent für den Anwender und die Programmausführung.

Wenn die CPU auf Programmcode oder Daten zugreifen will, prüft der Cache Controller, ob die erforderliche Information in einer Cache Line vorhanden ist. Dazu wird die von der CPU angeforderte Adresse in folgende drei Felder aufgeteilt:

- Tag Field
- Set Index Field
- Data Index Field

Das Set Index Field der Adresse wird verwendet, um die Cache Line innerhalb des Cache Memories zu lokalisieren, welche die erforderliche Information enthalten könnte. Der Cache Controller wird das Valid Bit prüfen um zu entscheiden, ob die Cache Line aktiv ist. Weiter prüft der Cache Controller, ob das Cache Tag im Cache Memory dem Tag Field der angeforderten Adresse entspricht. Wenn diese Tests erfolgreich sind, liegt die von der CPU angeforderte Information im Cache Memory. In diesem Falle spricht man von einem Cache Hit. Wenn einer dieser Tests negativ sind, liegt die Information nicht im Cache Memory, dies wird auch als Cache Miss bezeichnet.

Im Falle eines Cache Miss wird der Cache Controller den Programmcode oder die Daten vom Main Memory ins Cache Memory kopieren. Dabei wird eine ganze Cache Line kopiert, nicht nur ein paar Bytes. Dieser Vorgang wird Cache Line Fill genannt. Der kopierte Programmcode oder die Daten werden anschliessend an die CPU weitergegeben. Das gleichzeitige Weiterleiten der Information an die CPU sowie das Speichern im Cache Memory wird auch Data Streaming genannt.

Im Falle eines Cache Hit wird die Information direkt vom Cache Memory zur CPU kopiert.

3.1.5. Beziehungen zwischen Cash und Main Memory

Die einfachste Beziehung zwischen Cache Memory und Main Memroy wird Direct Mapped Cache genannt. Im Direct Mapped Cache wird jede Adresse des Main Memory auf eine spezifische Cache Line gemappt. Weil aber das Main Memory wesentlich grösser ist als das Cache Memory gibt es sehr viele Adressen im Main Memory, welche auf dieselbe Cache Line gemappt werden. Abbildung 3.3 zeigt diese Beziehung.

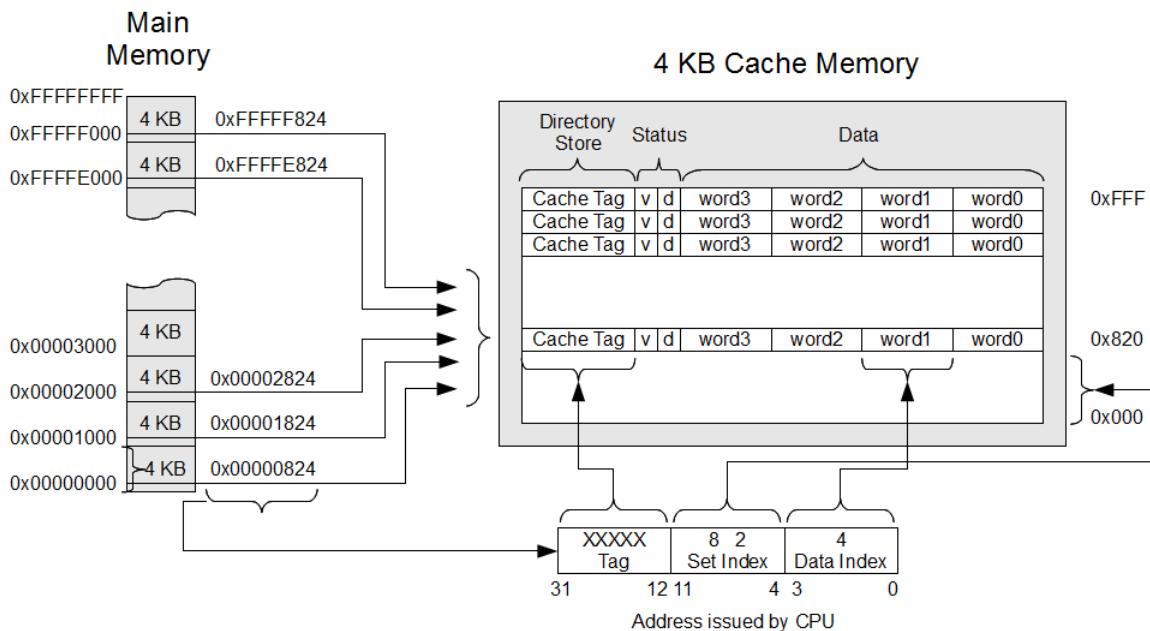


Abbildung 3.3.: Direct Mapped Cache

Abbildung 3.3 zeigt, dass alle Adressen, welche auf 0x824 enden (Set Index ist 0x82, Data Index ist 4), dasselbe Word im Cache Memory ansprechen.

Wenn die CPU Programmcode oder Daten anfordert, die nicht im Cache Memory vorhanden sind, und das Cache Memory schon voll ist, muss der Cache Controller eine Cache Line löschen, auch wenn diese aktuelle Information enthält. Das Löschen einer aktiven Cache Line aufgrund eines Cache Miss wird auch „eviction“ genannt.

Der Nachteil von Direct Mapped Cache ist, dass es im Cache Memory jeweils nur eine Möglichkeit gibt, um eine Adresse aus dem Main Memory zu speichern. Dies kann zu so genanntem „Trashing“ führen, d.h. es gibt einen Wettbewerb um Einträge im Cache Memory. Das Resultat ist ein fortlaufendes laden und löschen (eviction) von Cache Lines, was die System Performance reduziert.

Um das Problem des Trashing zu minimieren, unterteilen einige Cache-Architekturen das Cache Memory in kleinere Blöcke, so genannte Ways, siehe auch Abbildung 3.4.

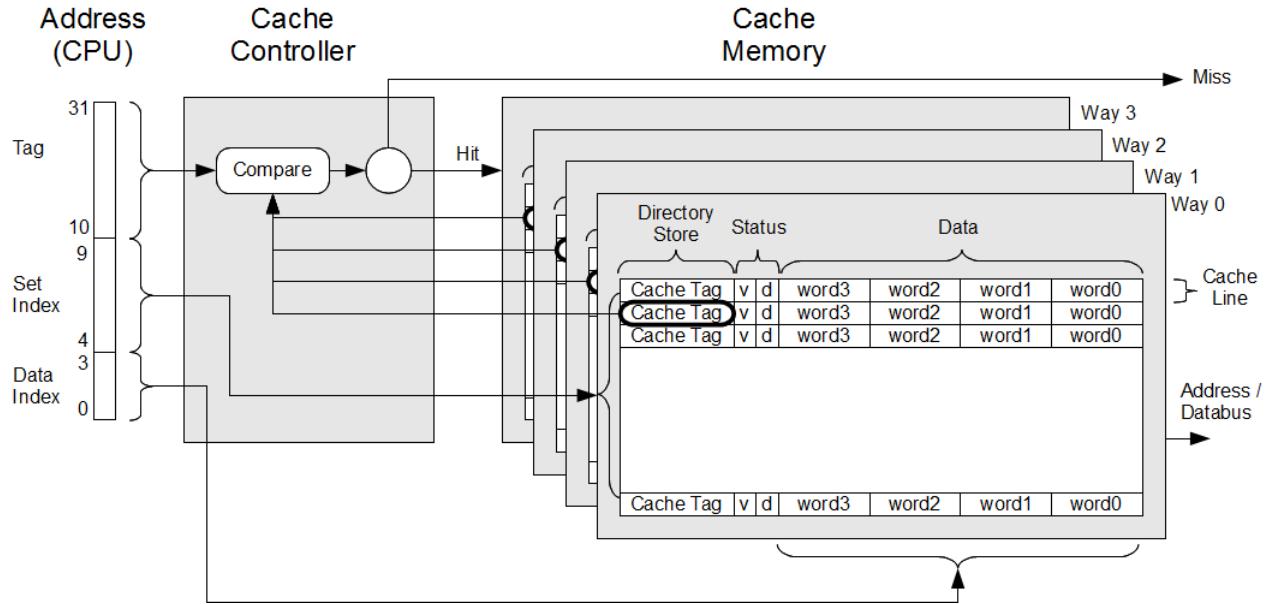


Abbildung 3.4.: 4KB Four-Way Set Associative Cache, bestehend aus 4*64 Cache Lines zu je vier 32-bit Words.

Das Set Index Field adressiert in dieser Architektur mehrere Cache Lines und zeigt auf eine Cache Line in jedem Way. Cache Lines mit demselben Set Index werden auch Set Associative genannt. Programmcode oder Daten aus dem Main Memory können in einer beliebigen Cache Line desselben Sets gespeichert werden.

Zu beachten ist, dass das Index Field aus Abbildung 3.4 aufgrund der vier Ways zwei Bits kleiner ist als dasjenige aus Abbildung 3.3, dass dafür aber das Tag Field zwei Bits grösser ist.

3.1.6. Caches und MMUs

MMUs werden im Kapitel 3.3 ausführlich beschrieben. Caches können zwischen CPU und MMU oder zwischen MMU und Main Memory liegen. Abbildung 3.5 zeigt diese beiden Varianten.

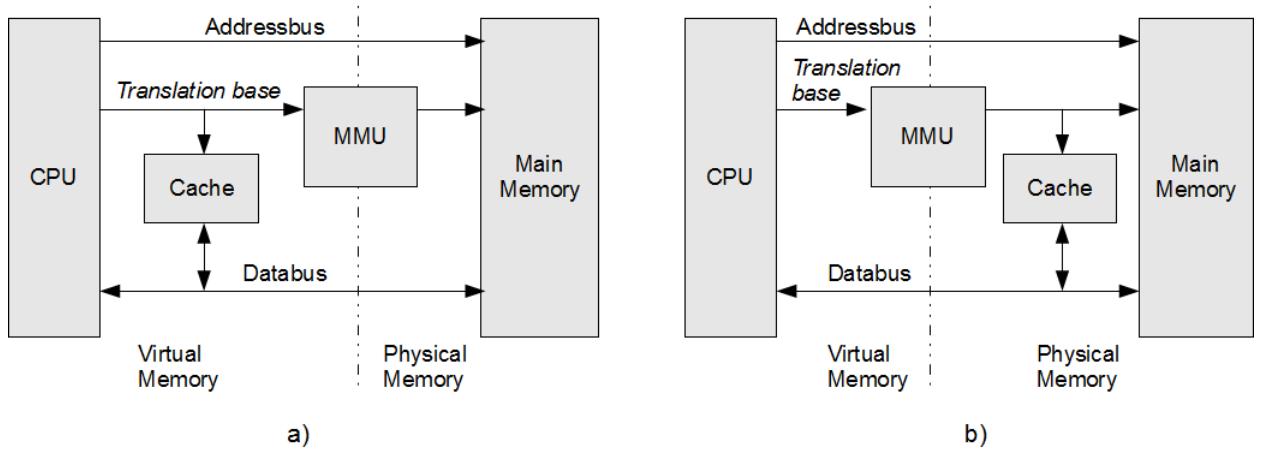


Abbildung 3.5.: a) Virtual Cache und b) Physical Cache

Ein virtueller (oder logischer) Cache befindet sich zwischen CPU und MMU. Programmcode und Daten werden im virtuellen Adressraum abgelegt. Die CPU kann den virtuellen Cache direkt ansprechen, ohne den Weg über die MMU.

Ein physikalischer Cache befindet sich zwischen MMU und Main Memory. Programmcode und Daten werden im physikalischen Adressraum abgelegt. Wenn die CPU Informationen aus dem Cache braucht, muss die MMU die virtuellen Adressen der CPU immer zuerst in die physikalischen Adressen des Caches übersetzen.

3.1.7. Write Buffers

Der Write Buffer ist ein zusätzlicher FIFO-Speicher, in welchem Daten, die ins Main Memory geschrieben werden müssen, zwischengespeichert werden.

In einem System ohne Write Buffer schreibt die CPU die Daten direkt ins Main Memory. Dies verursacht zusätzliche Wartezyklen (wait cycles) aufgrund der langsamen Schreibzugriffe im Main Memory.

In einem System mit Write Buffer befreit dieser die CPU und das Cache Memory vom langsamen Schreibzugriff auf das Main Memory. Eine Cache Line kann mit hoher Geschwindigkeit in den Write Buffer kopiert werden. Der Write Buffer kopiert die Daten anschliessend mit tiefer Geschwindigkeit ins Main Memory. Dadurch erhöht der Write Buffer die Performance des Caches: Wenn der Cache Controller Daten einer dirty Cache Line löschen muss (eviction), wird diese zuerst in den Write Buffer kopiert. Dabei ist allerdings zu beachten, dass Daten im Write Buffer noch nicht zum Lesen zugänglich sind, diese müssen zuerst ins Main Memory kopiert werden. Write Buffer sind aus diesem Grund relativ klein.

3.1.8. Cache Policies

Es gibt zwei Arten von Cache Policies:

- Writeback
- Writethrough

Bei der Writeback Policy schreibt der Cache Controller die Daten nur ins Cache Memory, nicht aber ins Main Memory. Dadurch kann eine aktive Cache Line andere Daten enthalten als das Main Memory. Im Cache Memory werden nur die zuletzt verwendeten Daten gespeichert. Wenn der Cache Controller eine veränderte (dirty) Cache Line löschen will, müssen die Daten in den Write Buffer oder direkt ins Main Memory kopiert werden, bevor die Cache Line gelöscht werden darf.

Der Vorteil der Writeback Policy ist die höhere System Performance: Funktionen im Programm verwenden häufig temporäre Variablen. Diese Variablen müssen nicht im Main Memory abgelegt werden. Der Nachteil der Writeback Policy ist, dass die Ausführungszeit nicht deterministisch (nicht berechenbar) ist. Das ist insbesondere für Systeme mit hohen Echtzeitanforderungen ein wichtiger Nachteil.

Bei der Writethrough Policy schreibt der Cache Controller die Daten sowohl ins Cache Memory als auch ins Main Memory. Dadurch stimmen die Inhalte von Cache Memory und Main Memory überein, was auch Coherent Cache genannt wird. Weil die Daten auch ins Main Memory geschrieben werden ist die Performance der Writethrough Policy schlechter als diejenige der Writeback Policy. Dafür sind Systeme mit Writethrough Policy deterministisch.

3.2. MPU

3.2.1. Einleitung

Die MPU (Memory Protection Unit) ist eine Hardware-Einheit für den Zugriffsschutz von Speicherbereichen (Memory Regions). Im Vergleich zur MMU (siehe Kapitel 3.3) bietet sie jedoch keine Umrechnung von virtuellen in physikalische Adressen. Die MPU erhöht insbesondere die Zuverlässigkeit und Sicherheit eines Embedded Systems, indem sie:

- Daten und Stack einer Applikation vor dem Zugriff anderer Applikationen schützt,
- den Zugriff auf die Peripherie für nicht privilegierte Applikationen unterbindet,
- die Codeausführung aus dem RAM für gewisse Bereiche verhindert.

Für einfache Anwendungen ohne Betriebssystem bietet eine MPU folgende Funktionalitäten:

1. Datenbereiche im RAM können als „Read-Only“ definiert werden, sodass diese Daten nicht überschrieben werden können.
2. Datenbereiche im Anschluss an den Stack können als „Read-Only“ oder „Inaccessible“ definiert werden, womit Stack-Überläufe detektiert werden können.
3. RAM-Bereiche können als „Not-Executable“ definiert werden, wodurch das Einfügen von Code durch Programmfehler oder Viren usw. erschwert wird.

Für Anwendungen mit einfachen Betriebssystemen, welche die Funktionalität einer MPU unterstützen (siehe Vorlesung „Kernmodul Embedded Systems“ sowie Vorlesung „Echtzeit-Betriebssysteme“), bietet eine MPU zusätzlich folgende Funktionalitäten:

1. Zugriffsschutz von Daten- und Programmspeicher, sodass eine Applikation nicht auf Bereiche anderer Applikationen zugreifen kann.
2. Zugriffsschutz der Stacks einzelner Applikationen, sodass eine Applikation nur auf ihren eigenen Stack zugreifen kann, bei Fehlern aber den Stack anderer Applikationen nicht verändert.
3. Zugriffsschutz der Peripherie, sodass nur privilegierte Applikationen auf die Peripherie zugreifen können.

3.2.2. Architektur

Die MPU befindet sich zwischen CPU und Speicher (siehe Abbildung 3.6). Sie prüft die Adressen, welche die CPU generiert und erzeugt zusätzliche Steuersignale (Memory Access Attributes). Falls die MPU eine Verletzung der Zugriffsrechte detektiert, wird eine Exception (siehe Kapitel 14.1) ausgelöst.

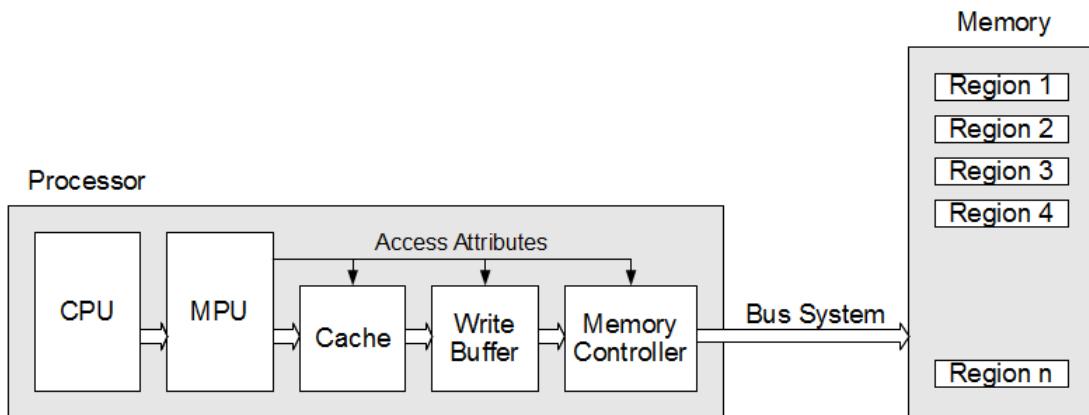


Abbildung 3.6.: Architektur MPU, Quelle: [13]

Der Speicher und auch die Peripherie werden in Memory Regions aufgeteilt. Diese Bereiche können unterschiedlichen Applikationen zugewiesen werden. Memory Regions können durch den Anwender oder das Betriebssystem dynamisch konfiguriert werden (Vergabe der Startadresse, der Grösse und der Access Attribute). Zur Definition von Memory Regions siehe auch Kapitel 3.2.3.

Die „Memory Access Attributes“ definieren Eigenschaften einzelner Memory Regions sowie von zusätzlichen Elementen wie Write Buffer oder Cache. Einige Möglichkeiten sind:

- Read-Only, Daten dürfen nicht überschrieben werden.
- Not-Executable, Information darf nicht ausgeführt werden.
- Inaccessible, auf den Bereich darf nicht zugegriffen werden.
- Bufferable, Daten werden im Write Buffer zwischengespeichert.

- Cacheable, Daten werden im Cache abgelegt.

3.2.3. Empfehlungen für die Definition von Memory Regions

Bevor die MPU konfiguriert werden kann, müssen die Memory Regions definiert werden. Die Einteilung kann wie folgt durchgeführt werden:

- Privilegierte Applikationen sowie das Betriebssystem erhalten eigene Regions für den Programmcode.
- Privilegierte Applikationen sowie das Betriebssystem erhalten eigene Regions für den Datenspeicher und den Stack.
- Peripherie-Bausteine, welche nur von privilegierten Applikationen oder vom Betriebssystem verwendet werden dürfen, erhalten eigene Regions.
- Nicht privilegierte Applikationen erhalten eine gemeinsame Region für den Programmcode sowie eine Region für die Daten und den Stack.
- Peripherie-Bausteine, welche von nicht privilegierten Applikationen verwendet werden können, werden in einer weiteren Region zusammengefasst.

Bei der Definition der Memory Regions gibt es aber einige Probleme, welche die Aufteilung gemäss obenstehender Liste erschweren: Programmcode von Applikationen, welche gemeinsame Libraries oder Gerätetreiber verwenden, kann nur schlecht in unterschiedliche Memory Regions aufgeteilt werden. Auch die Abgrenzung zwischen den Applikationen ist oft schwierig und muss dem Linker durch zusätzliche Angaben mitgeteilt werden (Linkerscript, #pragma). Deshalb wird oft auch der gesamte Programmcode in eine gemeinsame Memory Region abgelegt, welche als Read Only markiert wird.

Für die Abgrenzung der Datenbereiche ist es notwendig, dem Linker Angaben zu machen, welche Daten er in welchen Memory Regions ablegen soll. Der Speicherbereich des Heap, welcher von allen Applikationen für die Allokierung von dynamischem Speicher verwendet wird (malloc und free), wird von den Applikationen gemeinsam verwendet und kann deshalb nicht durch die MPU geschützt werden.

3.3. MMU

3.3.1. Einleitung

Die MMU (Memory-Management Unit) ist eine Hardware-Einheit zur Speicherverwaltung. Sie bietet insbesondere beim Einsatz von Betriebssystemen Vorteile. Die MMU ist häufig auf demselben Chip wie die CPU integriert, allerdings ist dies nur bei leistungsfähigen 32-Bit oder 64-Bit Prozessoren der Fall.

Eine MMU bietet folgende Funktionalität:

1. Schutz der Speicherbereiche einzelner Programme: Auf einem Betriebssystem können mehrere Programme gleichzeitig laufen. Mit Hilfe der MMU kann verhindert werden, dass ein Programm auf den Speicherbereich eines anderen Programms zugreift. Diese Funktionalität ist für Systeme sehr wichtig, weil dadurch die Sicherheit erhöht wird.
2. Es kann mehr Programmcode im Secondary Memory abgelegt werden, als dass Code im Main Memory gespeichert werden kann. Der aktuell auszuführende Code wird vom Secondary Memory ins Main Memory kopiert, Code den man aktuell nicht braucht wird überschrieben. Daten, die temporär nicht verwendet werden, müssen ebenfalls ins Secondary Memory ausgelagert werden („swapping“). Diese Funktion ist im Bürobereich wichtig, wo wesentlich mehr Programme auf dem Secondary Memory vorhanden sind, als dass aktuell ausgeführt werden. Im Embedded Bereich spielt diese Funktionalität aber eine sehr untergeordnete Rolle.

3.3.2. Virtuelle und physikalische Adressen

Als virtuelle oder auch logische Adressen werden diejenigen Adressen bezeichnet, welche der Compiler / Linker einem Programm zuweist und welche das Programm auch während der Ausführung auf der CPU erhält. So können mehrere Programme parallel ausgeführt werden, welche alle auf der virtuellen Adresse 0 starten. Als physikalische Adressen werden jene bezeichnet, welche bei der Ausführung des Programms auf den Adressbus ausgegeben werden (physikalische Adressen der Speicherbausteine).

→ Programme und CPU arbeiten mit virtuellen Adressen, der Zugriff auf die Speicherbausteine erfolgt über die physikalischen Adressen.

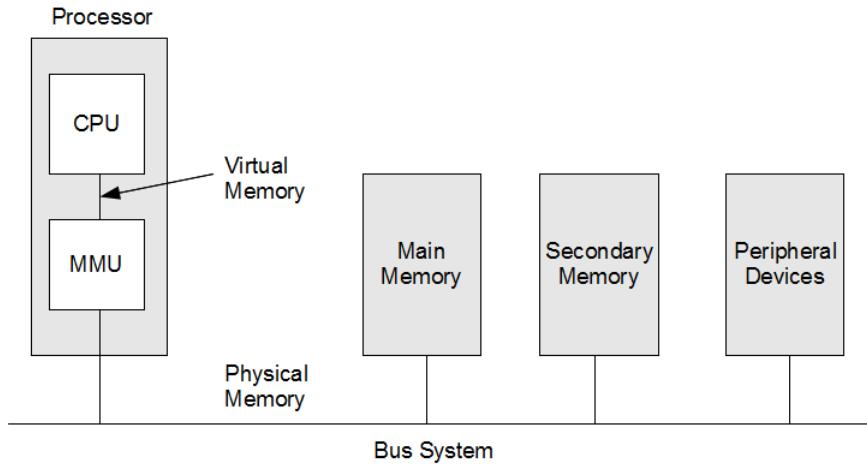


Abbildung 3.7.: MMU zwischen CPU und Speicher

Die Aufgabe der MMU besteht darin, die virtuellen Adressen auf die physikalischen Adressen umzurechnen (relocation). Dazu addiert die MMU zur virtuellen Adresse den Wert im so genannten Relocation-Register. Das Resultat ist die physikalische Adresse. Gleichzeitig prüft die MMU, ob sich der Zugriff in einem erlaubten Bereich befindet. Dazu werden zusätzliche Limit-Register verwendet.

Der Aufbau einer MMU kann vereinfacht wie folgt dargestellt werden:

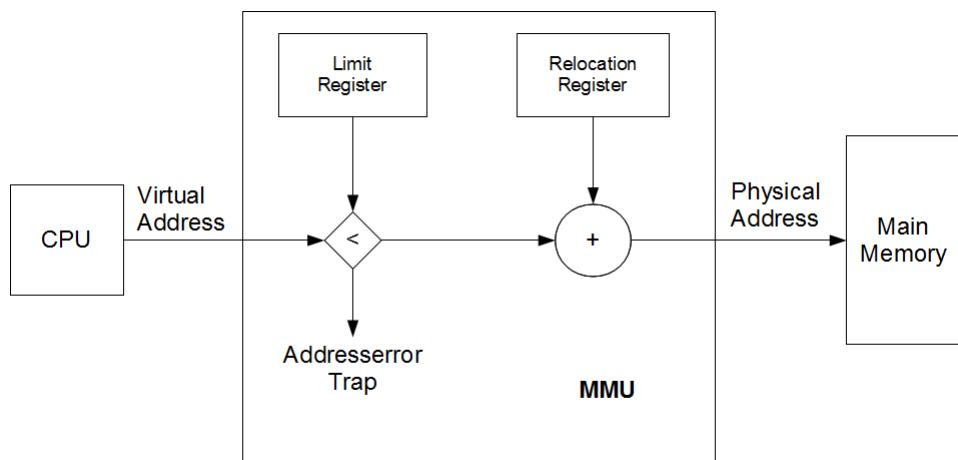


Abbildung 3.8.: Vereinfachtes Blockschaltbild einer MMU

Um von einem Programm zum anderen zu wechseln müssen die MMU-Register (Limit, Relocation) neu geladen werden. Das ist eine der Aufgaben eines Betriebssystems. Bei Prozessoren ohne MMU entspricht die virtuelle Adresse der physikalischen Adresse.

3.3.3. Pages und Frames, Page Table

Der Virtuelle Adressbereich wird in Blöcke gleicher Grösse unterteilt, die so genannten „Pages“. Auch der physikalische Adressbereich wird in gleich grosse Blöcke eingeteilt, diese werden „Frames“ genannt. Die Grösse der Pages und Frames liegt je nach System zwischen 512 Bytes und 16 MBytes.

Für die Umrechnung der virtuellen in die physikalische Adresse wird eine so genannte „Page Table“ verwendet. Die virtuelle Adresse wird wie folgt aufgeteilt:

1. Eine Page-Number, welche den Index in die Page-Table angibt. Die Page-Table enthält als Eintrag die Basis-Adresse jedes Frames (physikalische Adresse).
2. Einen Page-Offset, welcher addiert zum Inhalt der Page-Table (Basis-Adresse) die physikalische Adresse ergibt.

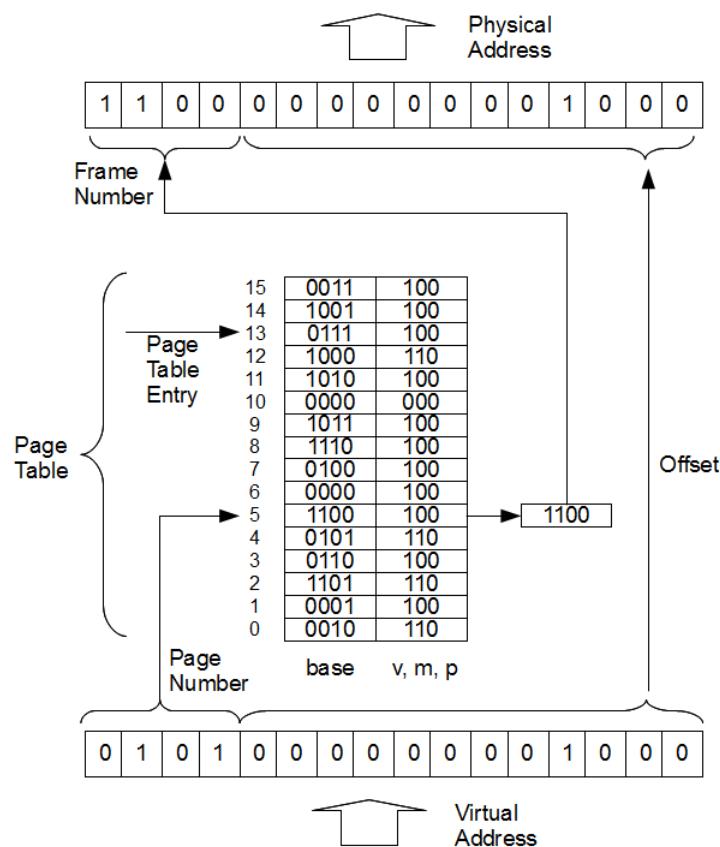


Abbildung 3.9.: Page Table für 16 * 4kByte Pages

Der virtuelle Adressraum kann grösser als der physikalische Adressraum sein. Das valid bit gibt an, ob die physikalische Adresse, auf welche der Inhalt der Page Table zeigt, auch gültig ist. Falls nicht, müssen die entsprechenden Daten zuerst vom Secondary Memory ins Main Memory kopiert werden.

3.3.4. Translation Lookaside Buffer

Page Tables werden je nach Grösse der Pages und des Speicherraumes, den sie abbilden, sehr gross. Andererseits arbeiten Programme während ihrer Ausführung normalerweise nur mit wenigen Pages (Programmschleifen, Daten in Arrays usw.). D.h. es werden jeweils nur wenige Einträge aus der gesamten Page-Table verwendet. Die Berechnung der physikalischen Adresse kann optimiert werden indem TLBs (Translation Lookaside Buffer) verwendet werden. Diese sind ebenfalls in der MMU integriert. Der TLB ist eine Tabelle mit 8 bis 64 Einträgen, welche den Einträgen in der Page Table entsprechen.

Page Number (virtual)	Frame (physical)	valid	modified	protection
36	22	1	0	r w x
18	54	1	1	r x
632	60	1	0	r w
320	52	1	0	r w

Tabelle 3.1.: Mögliche Einträge im TLB

Im TLB werden pro Eintrag Informationen über eine Page abgelegt: Page Number, Frame, valid bit, modified bit (zeigt ob der Inhalt der Page modifiziert wurde und somit vor dem Verwerfen noch gespeichert werden muss), protection (Zugriffsrechte auf ein Frame).

Der Ablauf für die Umrechnung der virtuellen in die physikalische Adresse ist nun wie folgt: Zuerst prüft die MMU, ob die Page Number im TLB gespeichert ist. Dazu werden sämtliche Einträge des TLB durch die MMU-Hardware gleichzeitig mit der anliegenden Page Number verglichen. Falls eine Übereinstimmung gefunden wird („TLB hit“), wird die Frame Number direkt aus dem TLB gelesen. Falls die Page Number nicht im TLB gefunden wird („TLB miss“), prüft die MMU die gesamte Page Table. Anschliessend wird ein Eintrag im TLB ausgewählt und mit der aktuellsten Page Number aus der Page Table ersetzt. Im TLB stehen somit immer die zuletzt verwendeten Page Number.

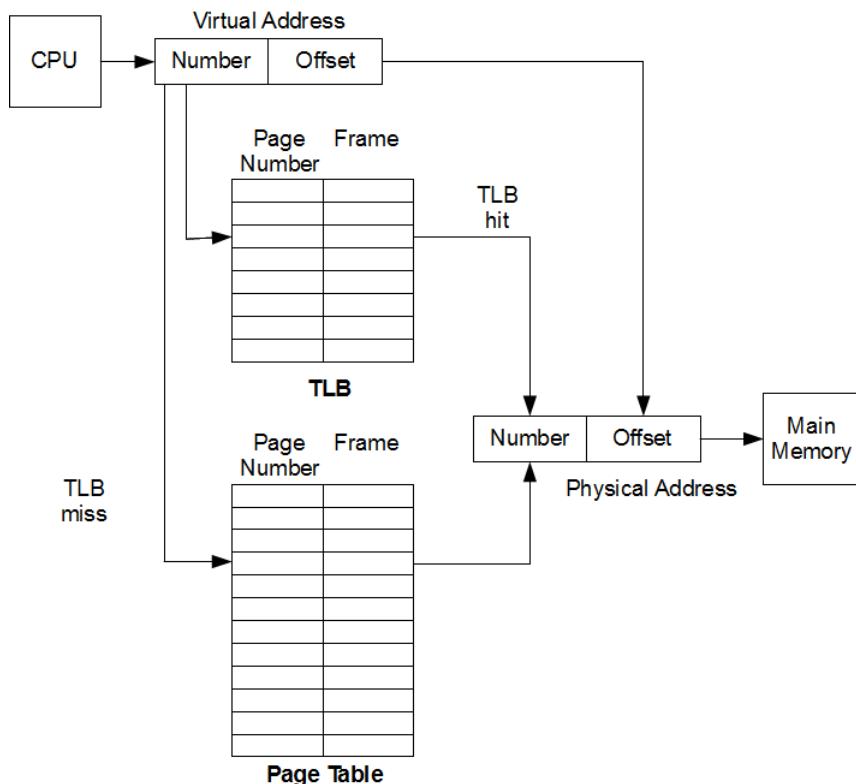


Abbildung 3.10.: MMU mit Page Table und TLB

3.4. DMA

3.4.1. Einleitung

DMA (Direct Memory Access) ist ein Verfahren um möglichst schnell grössere Datenmengen von einer Quelle zu einem Ziel zu kopieren (beispielsweise von einer Kommunikations-Schnittstelle ins RAM). Auch Kopiervorgänge zwischen zwei Speicherbereichen sind möglich. Für diesen Kopiervorgang wird zusätzliche Hardware, der sogenannte DMA-Controller, verwendet.

Ohne DMA-Controller müssen Daten von der Peripherie via CPU ins RAM kopiert werden. Die CPU verwaltet Quelladresse, Zieladresse und verwendet ein oder mehrere Register als Zwischenspeicher für die Daten. Der Transfer muss programmiert werden. Um ein Datenwort zu kopieren, müssen mehrere Buszyklen ausgeführt werden, was den Transfer sehr langsam macht. Die CPU ist für den Kopiervorgang zu 100% ausgelastet und kann während dieser Zeit keine anderen Aufgaben ausführen.

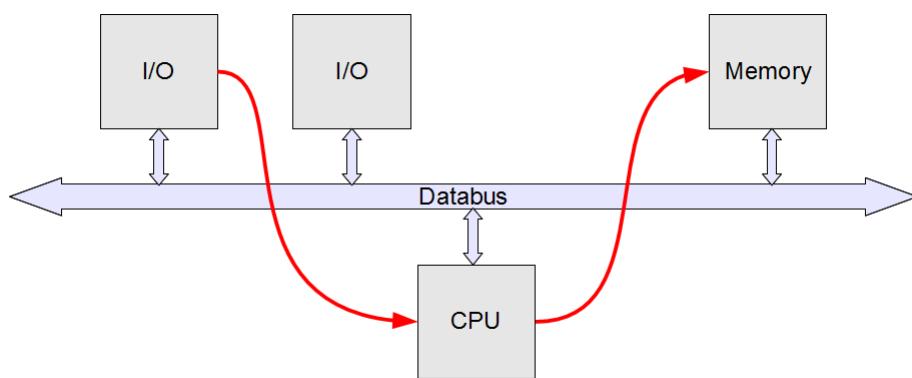


Abbildung 3.11.: Datentransfer ohne DMA, über die CPU

Mit DMA-Controller können Daten von der Quelle zum Ziel ohne Verwendung der CPU kopiert werden. Die CPU wird dazu für eine gewisse Zeit vom Bus abgekoppelt und der DMA-Controller übernimmt die Buskontrolle. Die Aufgabe der CPU besteht lediglich darin, den DMA-Controller zu Beginn des Transfers korrekt zu initialisieren (Quelladresse, Zieladresse, Datenmenge). Die CPU kann während des Datentransfers andere Aufgaben übernehmen.

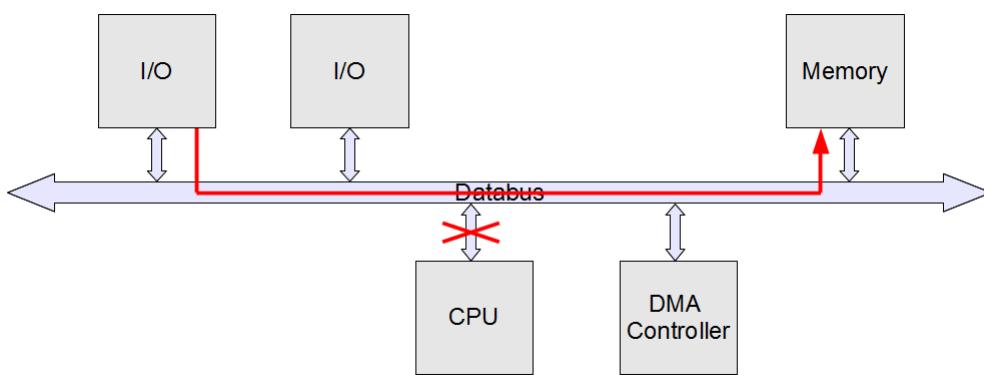


Abbildung 3.12.: Datentransfer mit DMA

3.4.2. Funktionsweise des DMA-Controllers

Ein DMA-Datentransfer muss von der CPU initiiert werden. Dazu übergibt die CPU dem DMA-Controller Quell- und Zieladresse sowie die Anzahl der zu übertragenden Bytes.

Da sowohl die CPU als auch der DMA-Controller auf den Bus zugreifen können, werden Arbitrierungsverfahren eingesetzt, um Buskonflikte zu vermeiden. Oft werden zusätzliche Steuerleitungen verwendet, wie etwa das Bus-Request-Signal, mit dem der DMA-Controller den Bus anfordern kann, sowie das Bus-Grant-Signal, mit welchem die CPU den Bus frei gibt.

Der DMA-Controller kann unterschiedliche Adressierungsverfahren anwenden:

- Beim **Explicit Addressing** Verfahren legt der DMA-Controller zuerst die Quelladresse auf den Bus, kopiert die Daten in ein internes Register, legt anschliessend die Zieladresse auf den Bus und kopiert die Daten in die Zieladresse. In diesem Falle werden zwei Buszyklen ausgeführt. Dieses Adressierungsverfahren wird für den Transfer von Daten zwischen zwei Speicherbereichen verwendet.
- Beim **Implicit Addressing** Verfahren kopiert der DMA-Controller die Daten direkt von der Quelle zum Ziel, ohne diese in einem internen Register zwischenzuspeichern. Dieses Verfahren benötigt nur einen Buszyklus und ist ideal um Daten von der Peripherie zum Speicher zu kopieren, oder umgekehrt. Ein Kopiervorgang Speicher - Speicher ist nicht möglich, da gleichzeitig nur eine Speicheradresse angelegt werden kann.

4. ARM und Cortex-Mx Prozessoren

Im Unterricht wird das Leguan-Board mit dem Microcontroller STM32H743 von ST Microelectronics eingesetzt. Dies ist ein Cortex-M7 Microcontroller mit einem Kern von ARM. In diesem Kapitel wird deshalb die Philosophie und Geschichte von ARM erklärt und anschliessend eine Übersicht über die Cortex-Mx Familie gegeben.

4.1. ARM Prozessoren

4.1.1. Einleitung

ARM-Prozessoren wurden ursprünglich (zwischen 1983 und 1985) von der Firma Acorn Computers Limited in Cambridge (England) entwickelt. Es war der erste kommerziell genutzte RISC-Prozessor. 1990 wurde die Firma ARM Limited (Advanced RISC Machines Limited) gegründet, welche die ARM-Technologie weiter entwickelte.

Microcontroller basierend auf der ARM-Architektur sind heute weit verbreitet. Dies hat mehrere Gründe:

- Gleiche Architektur des Kerns (Core), was zu hoher Wiederverwendbarkeit von Code und Tools führt.
- hohe Rechenleistung
- geringe Leistungsaufnahme (typischerweise kleiner 1 Watt)
- low cost

Die hohe Verbreitung führt auch zu einer Vielzahl von Einsatzgebieten, insbesondere für Embedded Systems jeder Art. ARM Prozessoren dienen auch als Plattformen für verschiedene Betriebssysteme, vom kleinen Echtzeit-Kernel bis hin zu ausgewachsenen OS wie Linux.

Die wichtigsten Eigenschaften aller ARM-Architekturen sind:

- Typische RISC-Architektur Eigenschaften wie:
 - Grosse Registerbank mit universellen 32-Bit Registern.
 - Load/Store Architektur, die Datenverarbeitung erfolgt mit Hilfe von Registern, kein direkter Zugriff der CPU auf den Speicher für die Datenverarbeitung.
 - Orthogonaler Befehlssatz
 - Einfache Adressierungsarten über die Standard-Register.
- Weitere ARM-spezifische Eigenschaften:
 - Verwendung der ALU und eines zusätzlichen „barrel shifters“ (shift left / right) in einer Instruktion.
 - Auto-Inkrement und Auto-Dekrement für die Adressierungsarten.
 - Load/Store Operationen für mehrere Register gleichzeitig.
 - Bedingte Ausführung von Instruktionen (conditional execution)
 - Datentypen Byte (8 Bit), Halfword (16 Bit) und Word (32 Bit)
 - 32-Bit ARM Instruktionen, 16-Bit Thumb Instruktionen, 16-Bit / 32-Bit Thumb-2 Instruktionen, Jazelle für Java Bytecode

ARM selber stellt keine Chips her, sondern verkauft das Design für die Cores. Heute gibt es weltweit viele Halbleiterhersteller, welche den ARM-Core lizenzierten und in ihre Prozessoren und Microcontroller integrieren. Einige dieser Hersteller sind: Analog Devices, Atmel, Freescale, Infineon, Marvell, NEC, NXP, Philips, Samsung, ST Microelectronics, TI und Toshiba.

4.1.2. ARM-Architekturen

ARM bietet historisch bedingt verschiedene Architektur-Versionen an. Neuere Versionen sind softwaremäßig abwärtskompatibel zu ihren Vorgängern und bieten neue Instruktionen sowie neue Hardware-Features. Zu jeder Architektur-Version werden verschiedene Familien und Kerne (Core) angeboten. Nachfolgende Tabelle zeigt eine Übersicht der Architektur-Versionen:

ARM-Architektur	Kurzbeschreibung	Family	Core
v1 1985	Version 1 der ARM-Architektur von Acorn Computers, basierend auf der CPU 6502. 26-Bit-Adressierung keine Multiplikation, kein Koprozessor	ARM1	ARM1
v2 1986	Erster kommerzieller Chip, 26-Bit-Adressierung, 32-Bit Datenbus, Multiplikation mit 32-Bit-Ergebnissen, einfach: 30'000 Transistoren, low Power v2 ohne Cache, v2a erstmals mit Cache Koprozessor-Unterstützung	ARM2	ARM2, ARM250
v3 1991	Erste Version von ARM Limited. Als Makrozelle, eingeständiger Prozessor oder integriert in einer CPU. On-Chip-Cache, MMU, 32-Bit-Adressierung, getrennte CPS- und SPS-Register. 35'000 Transistoren, Die Version V3M unterstützte erstmals die Multiply-Accumulate-Befehle mit 64-Bit-Ergebnis.	ARM6 ARM7	ARM600, ARM610 ARM700, ARM710
v4 1995	Die älteste ARM-Architektur, welche heute noch unterstützt wird. Neue Befehle zum Laden und Speichern von signed und unsigned Halbwörtern und signed Bytes. Fünfstufige Pipeline. Einführung des System-Modus. V4 verfügt erstmals über eine formale Definition.	ARM8	StrongARM, ARM810
v4T	Neue komprimierte 16-Bit-Thumb-Form des Befehlssatzes.	ARM7TDMI ARM9TDMI	ARM710T, ARM720T, ARM740T ARM920T, ARM940T
v5TE 2002	Verbesserte Thumb-Befehle. Neue Befehle wie BLX, CLZ und BRK. Erweiterung des Befehlssatzes für die Signalverarbeitung (DSP-Unterstützung) der E-Version.	ARM9E ARM10E	XScale, ARM946 ARM1020

Tabelle 4.1.: Übersicht der ARM-Architekturen, Teil I

ARM-Architektur	Kurzbeschreibung	Family	Core
v5TEJ 2002	Neu eingeführte Jazelle-Technologie zur optimalen Ausführung von Java Bytecode. Dies ist wesentlich schneller als eine rein in Software implementierte Java Virtual Machine (JVM).	ARM9E ARM10E	ARM926EJ ARM1026EJ
v6 2002	Unterstützung von Multiprozessor-Umgebungen. Thumb-2 Multi-Media Instruktionen welche SIMD unterstützen, z.B. für MPEG4. TrustZone: Aufteilung des physikalischen Adressraums in einen sicheren und einen unsicheren Teil. 6 neue Statusbits (GE[3:0], E-Bit, A-Bit).	ARM11	ARM1136J, ARM1156T2
v6-M	Microcontroller Profile Thumb und reduzierter Thumb-2 Befehlssatz Hardware-Multiplikation	Cortex-M	Cortex-M0, Cortex-M0+, Cortex-M1
v7-A 2004	Application Profile 1 - 4 SMP Cores Hohe Rechenleistung Betriebssysteme: iOS, Android, Linux, Windows ARM, Thumb und Thumb-2 Befehlssatz Hardware-Multiplikation und optional Division DSP, FPU, MMU, Jazelle	Cortex-A	Cortex-A5, Cortex-A7, Cortex-A8, Cortex-A9, Cortex-A12, Cortex-A15, Cortex-A17
v7-M 2005	Microcontroller Profile low-cost, energieeffizient Thumb und Thumb-2 Befehlssatz Hardware-Multiplikation und Division Optional DSP und FPU (Cortex-M4, Cortex-M7)	Cortex-M	Cortex-M3, Cortex-M4, Cortex-M7
v7-R 2011	Real-time Profile Hohe Rechenleistung Einsatzgebiete: high end Real-time Thumb und Thumb-2 Befehlssatz Hardware-Multiplikation und optional Division DSP und optional FPU Parity check, Cache, TCM	Cortex-R	Cortex-R4, Cortex-R5, Cortex-R7, Cortex-R8
v8-A 2012	Application Profile 1 - 4 SMP Cores A32 und A64 Befehlssatz SIMD Hardware-Visualisierung	Cortex-A	Cortex-A32, Cortex-A57, Cortex-A..., Cortex-A78
v8-M	Microcontroller Profile Security Extension TrustZone	Cortex-M	Cortex-A23, Cortex-A33
v8-R 2016	Real-time Profile “Bare-Metal“ Hypervisor-Modus Safety Isolation einzelner SW-Systeme	Cortex-R	Cortex-R52

Tabelle 4.2.: Übersicht der ARM-Architekturen, Teil II

Basierend auf den verschiedenen Architekturen werden diverse ARM-Familien angeboten (ARM11, Cortex-M usw.). Innerhalb einer Familie sind mehrere Cores erhältlich (z.B. Cortex-M3, Cortex-M7). Die Nomenklatur für die Cores

lautet wie folgt:

ARM{x}{y}{z}{T}{D}{M}{I}{E}{J}{F}{S}

x: Family

y: Memory management / protection unit

z: Cache

T: Thumb 16-bit decoder

D: JTAG Debug

M: Fast multiplier

I: Embedded ICE Macrocell

E: Enhanced instructions

J: Jazelle (Java Bytecode)

F: Vector floating-point unit

S: Synthesizable version

Alle Cores welche nach dem ARM7TDMI entwickelt wurden, enthalten die TDMI Features.

4.2. Cortex-Mx-Prozessoren

4.2.1. Übersicht

Die Cortex-Mx-Familie besteht aus diversen Cores. Cortex-M0 und Cortex-M0+ basieren auf der ARMv6-M Architektur. Diese Cores bestehen aus nur etwa 12K Gates und sind somit ideal für low-cost Produkte. Die Instruktionen basieren auf dem Thumb und teilweise auf einem reduzierten Thumb-2 Instruktionssatz. Durch die geringe Anzahl Gates sind diese Cores sehr energieeffizient und dadurch ideal für den Einsatz in low-power Applikationen. Der Cortex-M1 basiert ebenfalls auf der ARMv6-M Architektur und kann in FPGAs eingesetzt werden. Cortex-M3, Cortex-M4 und Cortex-M7 basieren auf der ARMv7-M Architektur und bieten mehr Rechenleistung sowie verschiedene Optionen wie DSP, FPU oder MPU. Sie verwenden den Thumb-2 Instruktionssatz.

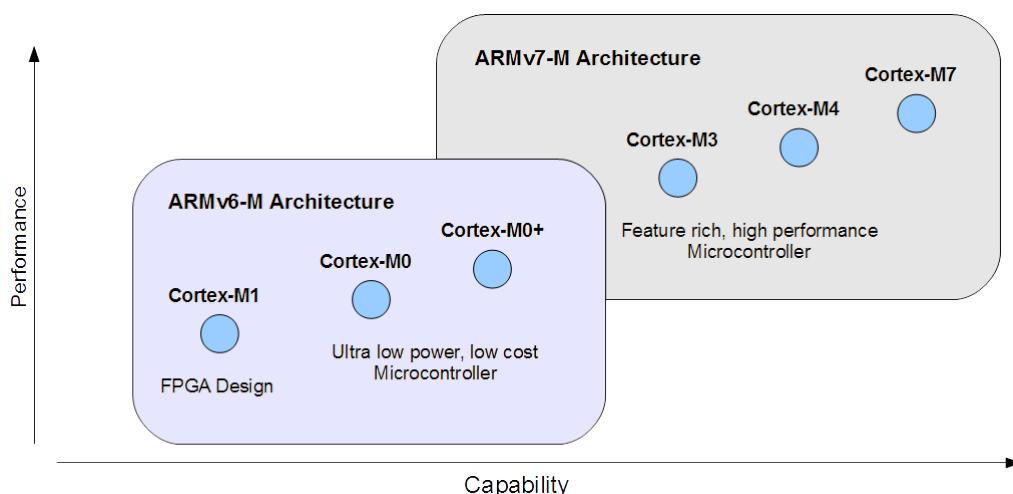


Abbildung 4.1.: Die ARM Cortex-Mx-Prozessor-Familie. Quelle: [13]

4.2.2. Vorteile

Cortex-Mx-Prozessoren bieten diverse Vorteile. Die wichtigsten sind:

- **Rechenleistung:** Im Vergleich zu Preis und Energieverbrauch liefern die Cortex-Mx-Prozessoren eine hohe Rechenleistung.

- **Low Power:** Das Core Design ist auf Low-Power optimiert und die Gate-Zahl ist gering. Dadurch erreichen die Cortex-Mx-Prozessoren einen Stromverbrauch von weniger als 200 μ A/MHz. Zudem werden diverse Sleep-Modi unterstützt. Durch Reduktion der Taktfrequenz kann der Energieverbrauch zusätzlich reduziert werden.
- **Energieeffizienz:** Durch die Kombination von Low-Power und hoher Rechenleistung erreichen die Prozessoren eine ausgezeichnete Energieeffizienz.
- **Interrupts:** Der Interrupt-Controller von Cortex-Mx-Prozessoren ist konfigurierbar (Maskierung, Priorisierung) und unterstützt viele Interrupt-Quellen. Interrupts können verschachtelt werden.
- **C friendly:** Cortex-Mx-Prozessoren sind für die Programmiersprache C optimiert. Sogar Interrupt-Handler können in C programmiert werden. Der lineare Adressbereich erleichtert die Programmierung ebenfalls.
- **Skalierbarkeit:** Die Cortex-Mx Prozessor Familie erlaubt eine einfache Skalierung, weil Prozessoren ohne grossem Aufwand durch eine leistungsfähigere Version ersetzt werden können. Die Firmware muss beim Wechsel auf einen anderen Prozessor nur geringfügig modifiziert werden.
- **Debug Support:** Cortex-Mx Prozessoren unterstützen eine Vielzahl von Debug-Features wie Traces für die Rückverfolgung des Programmablaufs oder Profiling Information.
- **OS Support:** Cortex-Mx-Prozessoren bieten verschiedene Funktionalitäten, um die Implementation eines OS möglichst einfach zu gestalten (beispielsweise System Tick).
- **Portabilität und Wiederverwendbarkeit:** Die gute C-Unterstützung und CMSIS (siehe Kapitel 12.4) ermöglichen eine sehr gute Portabilität und Wiederverwendbarkeit von Code.
- **Angebot an Chips und Tools:** Auf dem Markt existiert eine Vielzahl an Microcontrollern verschiedener Hersteller, die alle auf Cortex-Mx basieren. Zudem sind sehr viele Tools wie Entwicklungsumgebungen erhältlich.

4.2.3. Architektur

Cortex-Mx-Prozessoren basieren auf einer 32-Bit RISC-Architektur. Die Breite der Befehle ist 16-Bit und 32-Bit gemischt, was eine hohe Codedichte bewirkt.

Für Cortex-M3, Cortex-M4 und Cortex-M7 Prozessoren gelten folgende Eigenschaften:

- 3-stufige Pipeline
- Harvard-Architektur on-chip
- Linearer Adressraum, 32-Bit Adressen (4GB Memory Space)
- NVIC Interrupt Controller (siehe Kapitel 14.6)
- Sleep Mode und verschiedene Stromspar-Funktionen
- optionaler MPU-Support (siehe Kapitel 3.2)
- Bit-Adressierung wird durch „Bit Band“ unterstützt.

Cortex-M4 und Cortex-M7 Prozessoren unterstützen zusätzlich folgende Funktionen:

- SIMD (siehe Kapitel 1.1.1)
- MAC (siehe Kapitel 1.5.1)
- Saturating Arithmetics (siehe Kapitel 1.5.1)
- Optional single precision FPU (siehe Kapitel 1.4 und Anhang B.2)

Cortex-M7 Prozessoren unterstützen zusätzlich folgende Funktionen:

- Höhere Taktraten
- längere Pipeline und L1-Cache bis 64 kB
- TCM für Befehle und Daten, je bis 16 MB
- höhere DSP-Performance

4.2.4. Dokumentation

Für die Entwicklung eines Cortex-Mx basierten Projektes sind mindestens folgende Dokumente notwendig:

1. Das Technische Reference Manual von ARM zum entsprechenden Cortex-Mx Core.
2. Das Datenblatt des Microcontroller-Herstellers (Pinout, Peripherie, Memory Map, Registerbeschreibung usw.)
3. Das Reference Manual des Microcontroller-Herstellers
4. Application Notes des Chipsetellers oder von ARM.

Für den STM32H743 Cortex-M7 auf dem Leguan-Board sind folgende Dokumente hilfreich:

- ARMv7-M Architecture Reference Manual [5]
- ARM Cortex-M7 Devices Generic User Guide [3]
- **Cortex-M7 Technical Reference Manual** [2]
- **STM32H743 Datasheet** [12]
- **STM32H7xx Reference Manual** [11]
- Procedure Call Standard for the ARM Architecture [4]
- STM32F7 Series and STM32H7 Series Cortex-M7 processor programming manual [9]

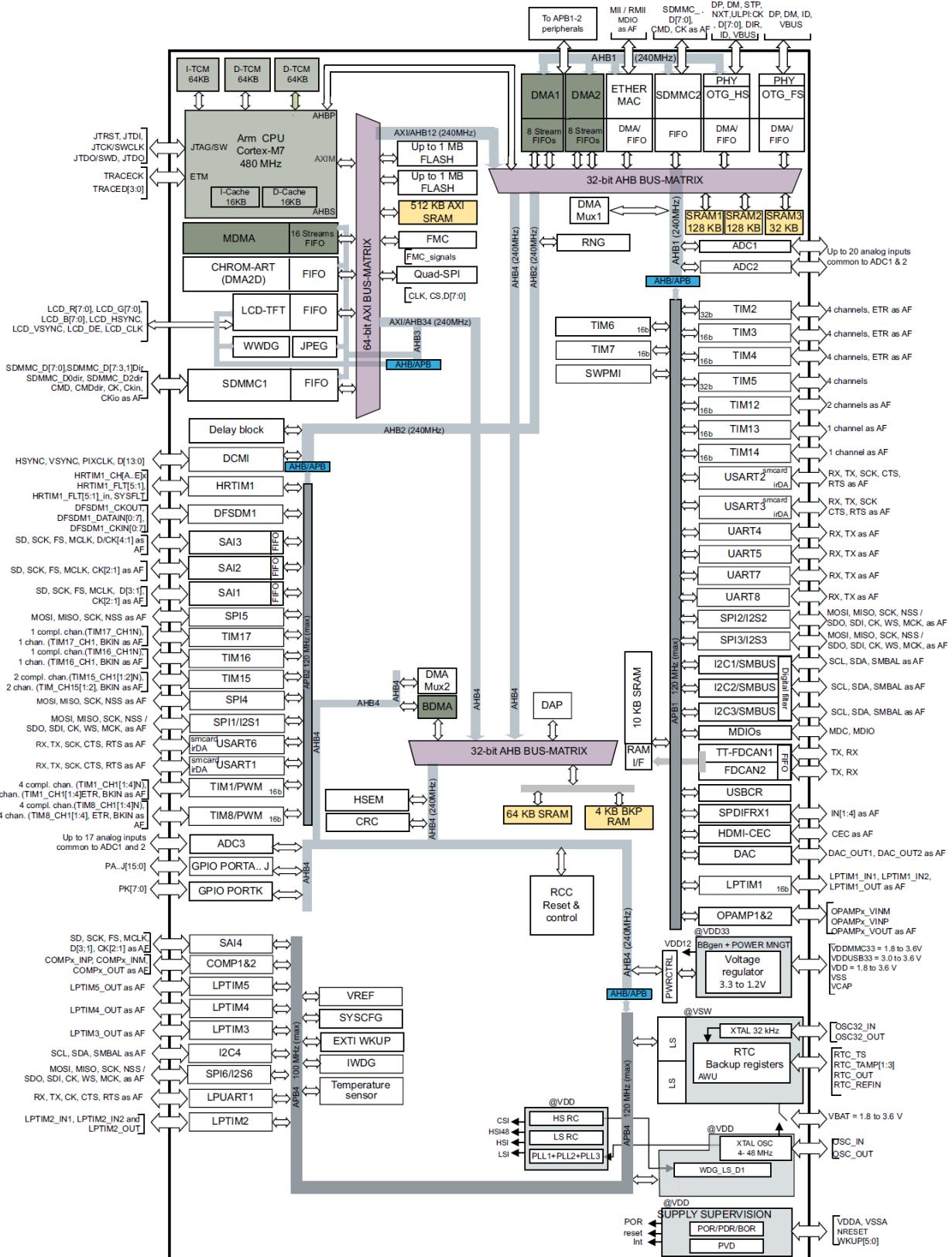
5. STM32H743

5.1. Eigenschaften

Der STM32H743 ist ein Vertreter der STM32H7xx-Familie von ST. Er besitzt eine Vielzahl interessanter Eigenschaften, welche im Datenblatt [12] ausführlich beschrieben sind. Die wichtigsten Eigenschaften sind:

- Core: ARM Cortex-M7 32-bit CPU mit FPU, MPU, L1 cache, DSP-Funktionalität, Clock bis 480 MHz
- Memories
 - Bis 2 MByte Flash
 - Bis 1 MByte SRAM
 - Memory Controller für externe Speicher wie NOR/NAND Flash, SRAM, PSRAM und SDRAM
- Debugging:
 - Serial Wire Debug (SWD) & JTAG interfaces
 - Embedded Trace Buffer™
- 4 General-Purpose DMA
- Bis zu 168 I/O Ports (GPIO) mit Interrupt-Funktionalität
- LCD-TFT Controller bis XGA-Auflösung
- 3x16-bit, 3.6 MSPS A/D-Wandler
- 2x12-bit, 1 MHz D/A-Wandler
- 22 Timers: bis 32 bit und 240 MHz, verschiedene Funktionalitäten wie PWM, Zähler oder inkrementaler Quadrature Encoder
- Kommunikations-Schnittstellen
 - 4 * USART / 4 UART (12.5 Mbit/s)
 - 4 * I2C interfaces (SMBus/PMBus)
 - 6 * SPI (bis 150 Mbits/s), 3 mit muxed duplex I2S für audio class
 - 2 * CAN interfaces (2.0B)
 - 2 * SD/SDIOMMC interface
 - 2 * USB OTG
 - Ethernet MAC: IEEE 1588v2
 - HDMI Interface
- 8- bis 14-bit Kamera-Interface bis 54 Mbytes/s
- True random number generator
- RTC mit Kalender

5.2. Blockdiagramm



5.3. Betriebsmodi

Cortex-M7 Prozessoren haben zwei Operation States und zwei Operation Modes. Des weiteren haben sie zwei Access Levels, den Privileged und den Unprivileged Access Level. Im Privileged Access Level können alle Ressourcen angesprochen werden, während im Unprivileged Access Level nicht alle Memory-Regionen verfügbar sind und ein paar wenige Instruktionen nicht genutzt werden können.

Operation States:

- Thumb State: Wenn der Prozessor Instruktionen ausführt, ist er im Thumb State. Prozessoren wie der STM32H743 haben keinen ARM State und können daher nur Thumb Instruktionen ausführen.
- Debug State: Wenn der Prozessor angehalten wird (z.B. durch einen Breakpoint oder den Debugger) geht er in den Debug State und stoppt die Ausführung weiterer Instruktionen.

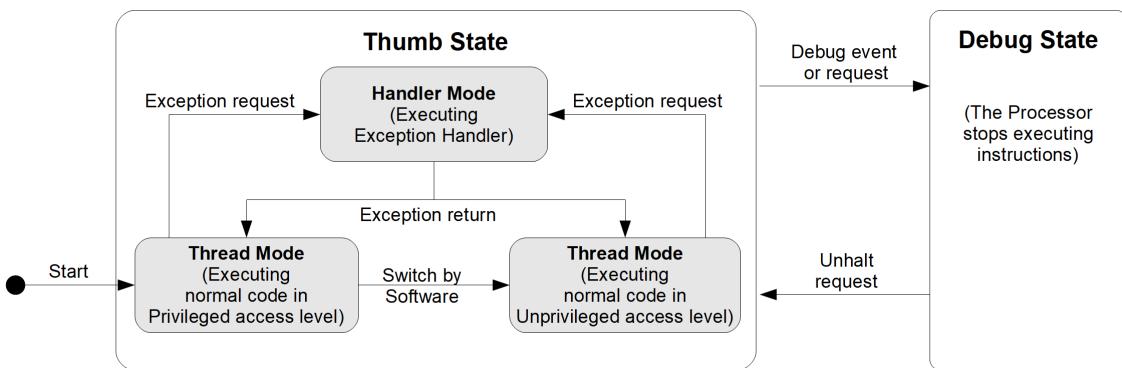


Abbildung 5.2.: Operation States und Modes. Quelle: [2] und [13]

Operation Modes:

- Thread Mode: Während der Ausführung von Applikationen (nicht in einer ISR) arbeitet der Prozessor im Thread Mode. Im Thread Mode kann sich der Prozessor entweder im Privileged oder Unprivileged Access Level befinden, dies wird über das Register „CONTROL“ gesteuert.
- Handler Mode: Im Handler Mode ist der Prozessor immer dann, wenn er Exceptions bearbeitet, beispielsweise eine Interrupt Service Routine (ISR). Im Handler Mode ist der Prozessor immer im Privileged Access Level.

Die Software kann vom Privileged in den Unprivileged Access Level wechseln, nicht aber umgekehrt. Soll vom Unprivileged in den Privileged Access Level gewechselt werden, muss dies über eine Exception erfolgen. Beispielweise kann der Kernel eines Embedded Operating Systems im Privileged Mode laufen, während die Tasks der Applikation im Unprivileged Mode laufen. Dadurch können die Tasks mit Hilfe der Memory Protection Unit (MPU) voreinander geschützt werden.

5.4. Program Status Register

Mit Hilfe des PSR (Program Status Register) kann der aktuelle Zustand der CPU ausgelesen oder verändert werden.

Das Program Status Register wird in drei Ansichten aufgeteilt:

- APSR (Application Program Status Register), kann von der Applikations-Software (unprivilegiert) angesprochen werden.
- IPSR (Interrupt Program Status Register), Zugriff wenn die CPU einen Exception-Handler ausführt.
- EPSR (Execution Program Status Register), enthält die Statusbits für die Ausführung.

Diese drei Ansichten können auch kombiniert als xPSR angesprochen werden.

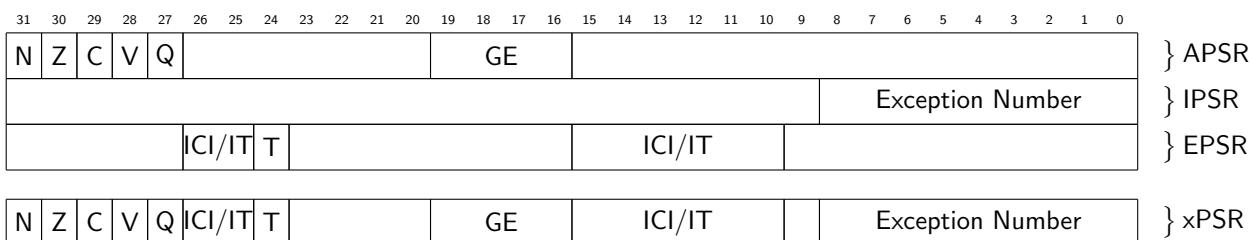


Abbildung 5.3.: APSR, IPSR und EPSR. Quelle: [5]

Einzelne Bits / Flags sind wie folgt definiert:

- Statusbits (Condition Flags): N, Z, C, V, Q, siehe auch Kapitel 5.5.
 - Greater-Than or Equal Flags (GE): Dies sind 4 Bits, wobei für die SIMD-Verarbeitung für jedes Byte ein GE-Flag existiert.
 - Interrupt-Continuable Instruction bits sowie If-Then Instruction Status bits (ICI/IT): Für die Verarbeitung von verschachtelten Interrupts (siehe Kapitel 14.5.1) sowie für bedingte Ausführung von Instruktionen (siehe Kapitel 7.1.4).
 - Thumb-State (T) (ist immer 1, durch löschen wird eine Fault exception ausgelöst).
 - Exception Number: Stellt dar, welche Exception der Prozessor gerade bearbeitet.

Für den Zugriff aus Assembler auf die PSR-Register werden die Instruktionen MRS und MSR verwendet.

5.5. Statusbits

Die Statusbits (Condition Flags) werden jeweils bei Vergleichen oder durch ALU-Operationen mit S-Suffix aktualisiert. So wird beispielsweise bei einer Subtraktion (SUBS-Instruktion) das Z-Bit (Zero) gesetzt, wenn das Resultat Null ist.

Bedingte Sprungbefehle werden immer abhangig von den Statusbits durchgefuhrt. Wenn beispielsweise in der Programmiersprache C eine Anweisung wie `if (a == 0)` programmiert wird, erfolgt der Sprung abhangig vom Z-Bit. Speziell bei ARM-Prozessoren ist, dass einzelne Instruktionen abhangig von den Statusbits ausgefuhrt werden konnen (Conditional Execution, siehe Kapitel 7.1.4).

Die einzelnen Statusbits werden in Tabelle 5.1 beschrieben.

Flag	Flag Name	Beschreibung
N	Negative	Bei vorzeichenbehafteten Zahlen (signed) wird dieses Bit als Vorzeichen interpretiert. Bei negativen Zahlen ist das N-Bit gesetzt (1), bei positiven Zahlen ist es gelöscht (0).
Z	Zero	Das Z-Bit ist gesetzt, wenn das Resultat einer Operation Null war. Dies gilt ebenfalls bei Vergleichen von zwei Werten: Wenn diese gleich sind, wird das Z-Bit gesetzt.
C	Carry	Bei Additionen wird das C-Bit bei einem Überlauf gesetzt, sonst ist es 0. Bei einer Subtraktion wird das C-Bit bei einem Unterlauf gelöscht, sonst ist es 1. Siehe auch Abbildung 5.4. Bei Shift-Operationen wird das letzte Bit der Operation in das C-Bit kopiert.
V	Overflow	Das V-Bit wird bei einem Überlauf von vorzeichenbehafteten Zahlen gesetzt (Addition oder Subtraktion). Siehe auch Abbildung 5.5.
Q	Saturation	Das Q-Bit wird durch Instruktionen zur Überlaufbegrenzung (Saturation) gesetzt. Es ist nur bei einigen ARM-Cores mit entsprechender DSP-Funktionalität vorhanden.

Tabelle 5.1.: Statusbits

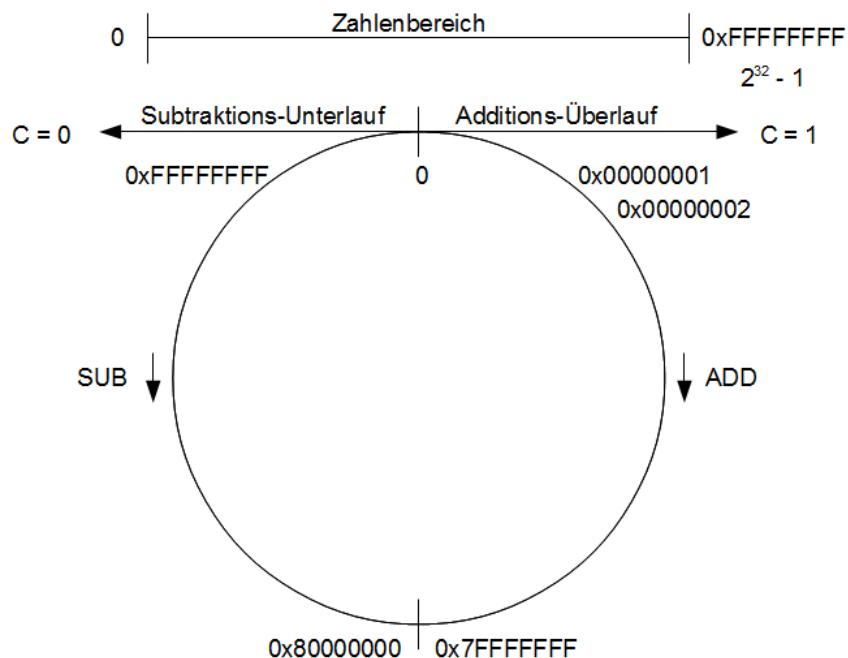


Abbildung 5.4.: Zahlenkreis vorzeichenloser Dualzahlen

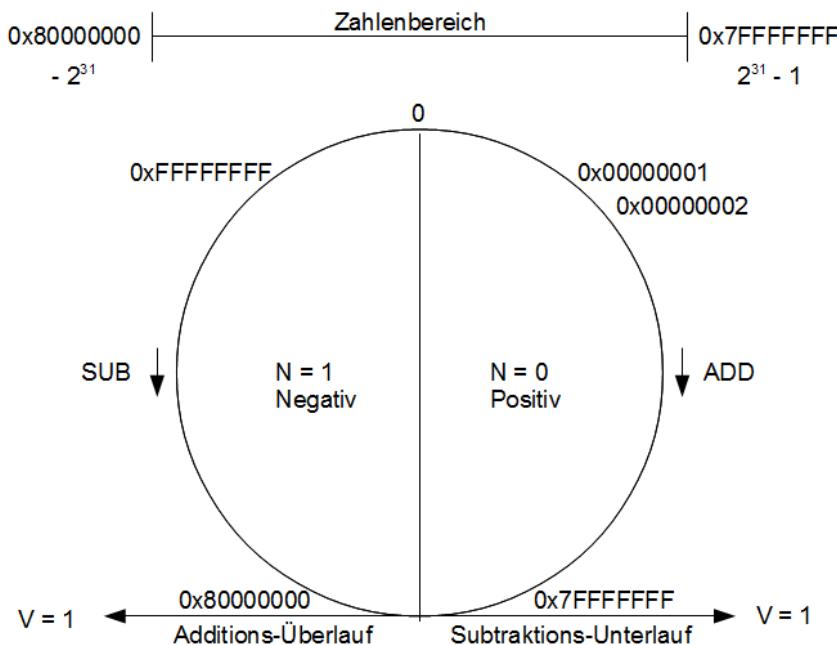


Abbildung 5.5.: Zahlenkreis vorzeichenbehafteter Dualzahlen

5.6. Speichermodell, Datenformat

Bei allen Prozessoren der ARM-Familie werden die Daten standardmäßig im „little-endian Format“ abgelegt, d.h. die tieferwertigen Bytes werden auf den tieferen Adressen gespeichert. Durch Konfiguration können die meisten ARM-Prozessoren aber auf das „big-endian Format“ umgeschaltet werden. Wir werden uns nachfolgend auf das „little-endian Format“ beschränken.

ARM-Prozessoren ab Architektur Version v4 unterstützen folgende Datenformate:

- signed und unsigned Byte (8 Bit, char)
- signed und unsigned Halfword (16 Bit, short int)
- signed und unsigned Word (32 Bit, long int)

Die ARM-Prozessoren verwenden eine Load/Store Architektur, d.h. die Operanden von ALU-Operationen sind immer in CPU-Register abgelegt. Diese ALU-Operationen verarbeiten somit Word-Formate (32-Bit). Die einzigen Instruktionen, welche Daten zwischen Memory und Registern transferieren, sind die Datentransfer-Befehle (Load und Store, PUSH und POP). Diese können mit den Datenformaten Byte, Halfword und Word umgehen. Wenn ein Byte oder ein Halfword vom Speicher geladen wird, wird es für die interne Verarbeitung durch Nullen oder das Vorzeichen auf 32-Bit erweitert (Barrel Shifter, siehe Kapitel 7.1.3).

Für die Ausrichtung der Daten im Speicher sind gewisse Bedingungen einzuhalten. Die folgenden Tabellen zeigen, wie Bytes, Halfword und Word im Adressraum abgelegt werden:

Speichermodell für Byte-Adressen:

Adresse	Byte
n	Byte 0
n + 1	Byte 1
n + 2	Byte 2
n + 3	Byte 3

Tabelle 5.2.: Byte-Speichermodell

Speichermodell für Halfword-Adressen:

Adresse	Halfword	Byte
n	Halfword 0	low Byte Halfword 0
		high Byte Halfword 0
n + 2	Halfword 1	low Byte Halfword 1
		high Byte Halfword 1

Tabelle 5.3.: Halfword-Speichermodell

Ein Halfword wird immer auf einer geraden Adresse (2-Byte Grenze) ausgerichtet. Auf der unteren Adresse (gerade) wird das tieferwertige Byte gespeichert, auf der ungeraden Adresse das höherwertige Byte. Das niederwertigste Adressbit A0 des Adressbusses ist Null.

Speichermodell für Word-Adressen:

Adresse	Word	Byte
n	Word 0	lowest Byte Word 0
		...
n + 4	Word 1	highest Byte Word 0
		lowest Byte Word 1
		...
		highest Byte Word 1

Tabelle 5.4.: Word-Speichermodell

Ein Word wird immer an einer 4-Byte Grenze ausgerichtet. Die zwei niederwertigsten Adressbits (A0 und A1) sind Null. Ein Word hat das tiefstwertige Byte auf der Adresse n und das höchstwertige Byte auf der Adresse (n + 3).

5.7. Memorymap STM32H7xx

Auf die CPU-Register (Core-Register) kann direkt zugegriffen werden, d.h. bei Assembler-Instruktionen können diese Register als Operanden angegeben werden. Beispiel: *MOV r0, r3*. Alle anderen Register des Prozessors (Hardware-Peripherie, z.B. GPIO, USB usw.) werden über den internen Adress-Databus angesprochen (Memory-Mapped). Jedem dieser Register ist eine Adresse zugeordnet, über die es angesprochen wird. Die Memory-Map (Speicherbelegungsplan) des Prozessors gibt an, welche Gruppen von Registern wo im Adressraum zu finden sind.

Die Memory-Map des STM32H7xx ist wie folgt definiert:

0xFFFF FFFF	512 Mbyte Block 7 Reserved
0xE000 0000	
0xDFFF FFFF	512 Mbyte Block 6
0xC000 0000	FMC SDRAM
0xBFFF FFFF	512 Mbyte Block 5
0xA000 0000	Reserved
0x9FFF FFFF	512 Mbyte Block 4
0x8000 0000	FMC NAND Flash, QUADSPI
0x7FFF FFFF	512 Mbyte Block 3
0x6000 0000	Ext. Memory FMC NOR/PSRAM
0x5FFF FFFF	512 Mbyte Block 2
0x4000 0000	Peripherals
0x3FFF FFFF	512 Mbyte Block 1
0x2000 0000	RAM
0x1FFF FFFF	512 Mbyte Block 0
0x0000 0000	Code

Abbildung 5.6.: Memory Map STM32H7xx

Abbildung 5.6 zeigt den Adressbereich eines STM32H7xx, aufgeteilt in 8 Blöcke zu je 512 MByte. Die Peripherie des STM32H7xx (USART, Timer, usw) wird über unterschiedliche interne Bussysteme (APB1, APB2, AHB1, AHB2) angesprochen. Die Adressen aller Peripherie-Bausteine können dem Reference Manual [11] entnommen werden.

5.8. Memorymap Leguan

Die Speicher auf dem Microcontroller sowie die zusätzlichen Speicherbausteine (externes SDSRAM) des Leguan Boards sind in die Memory-Map eingebunden. Die Peripherie des Leguan-Boards ist über das FPGA und den FMC-Bus ebenfalls eingebunden.

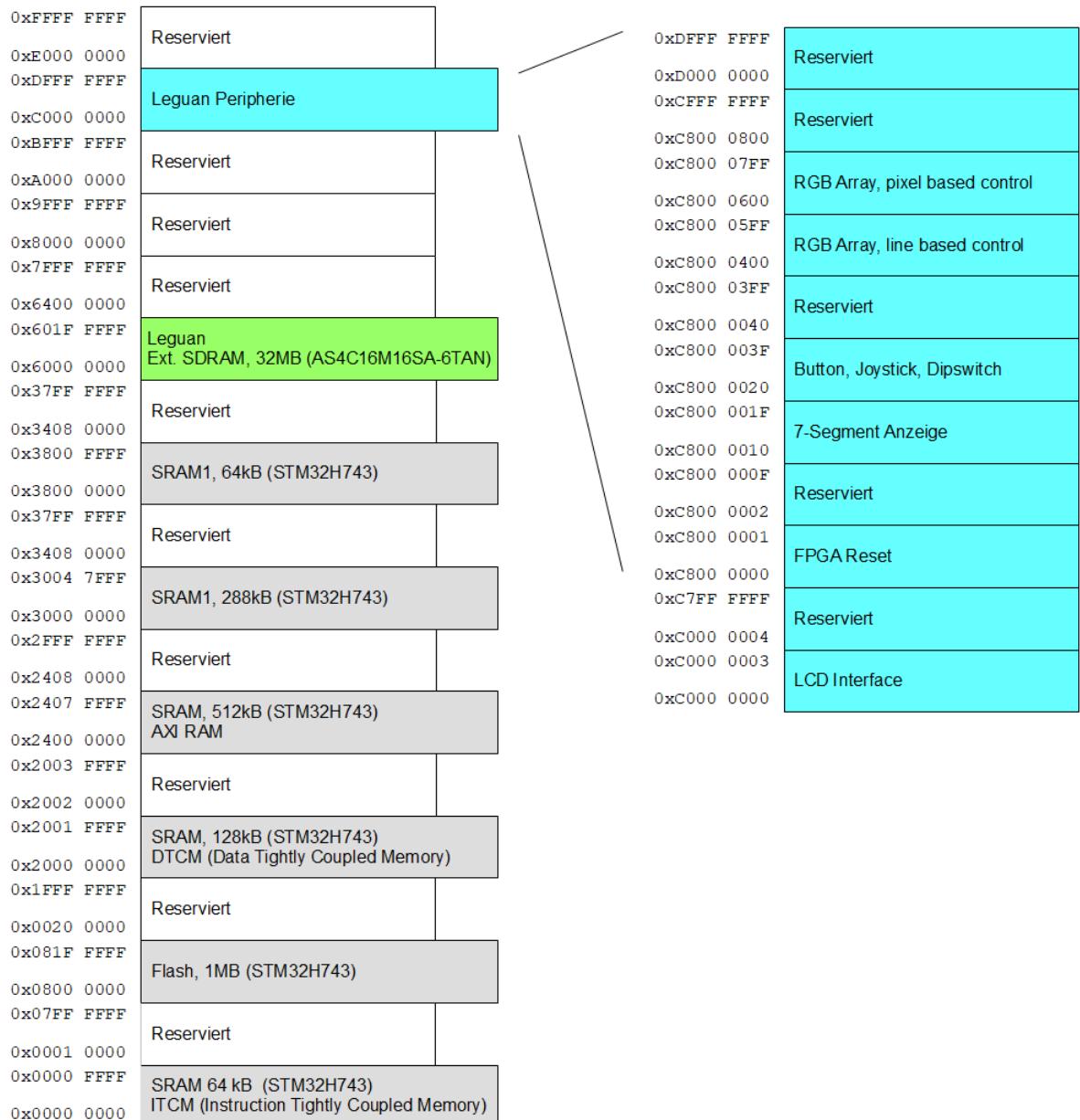


Abbildung 5.7.: Memory Map Leguan-Board, Peripherie und SDRAM

5.9. Pinbelegung STM32H743

Der STM32H743 ist in verschiedenen Gehäusen mit unterschiedlicher Grösse erhältlich. Je nach Gehäuse sind die Anschlüsse unterschiedlich. Ein Beispiel für eine Pinbelegung zeigt Abbildung 5.8.

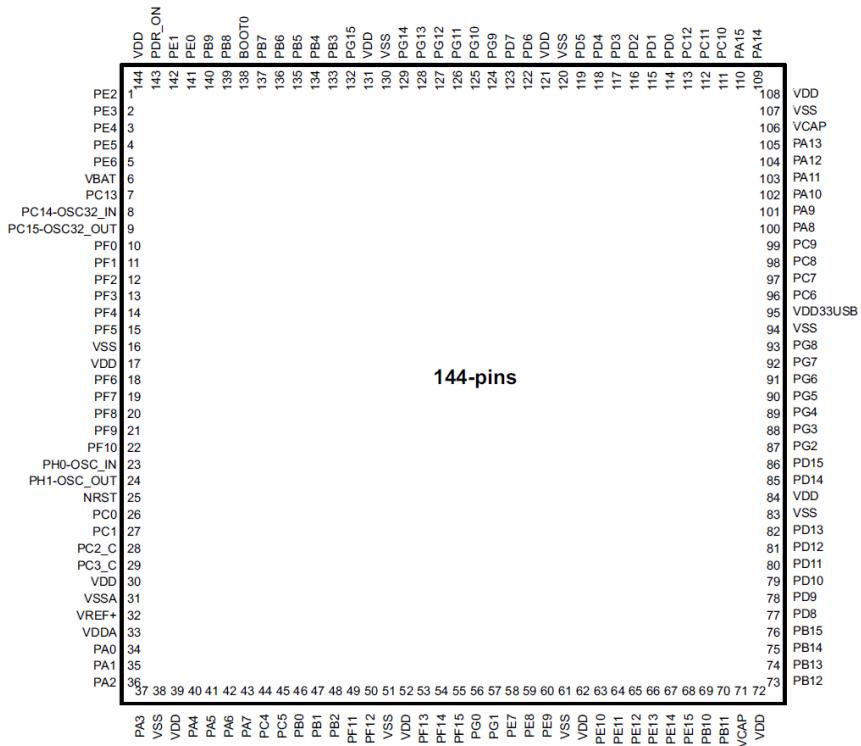


Abbildung 5.8.: STM32H743 Pinout, Quelle: [12]

5.10. Anschluss SDRAM

Der STM32H7xx besitzt einen Memory-Controller, welcher auch SDRAM ansteuern kann. Als externes SDRAM wird der Typ AS4C16M16SA-6TAN der Firma Alliance Memory mit 256Mb eingesetzt. Dieses SDRAM hat 16 Datenleitungen (D0 bis D15, 16-Bit Organisation) und 13 Adressleitungen (A0 bis A12 für Row und sowie A0 bis A9 für Column). Der Schaltplan sieht wie folgt aus:

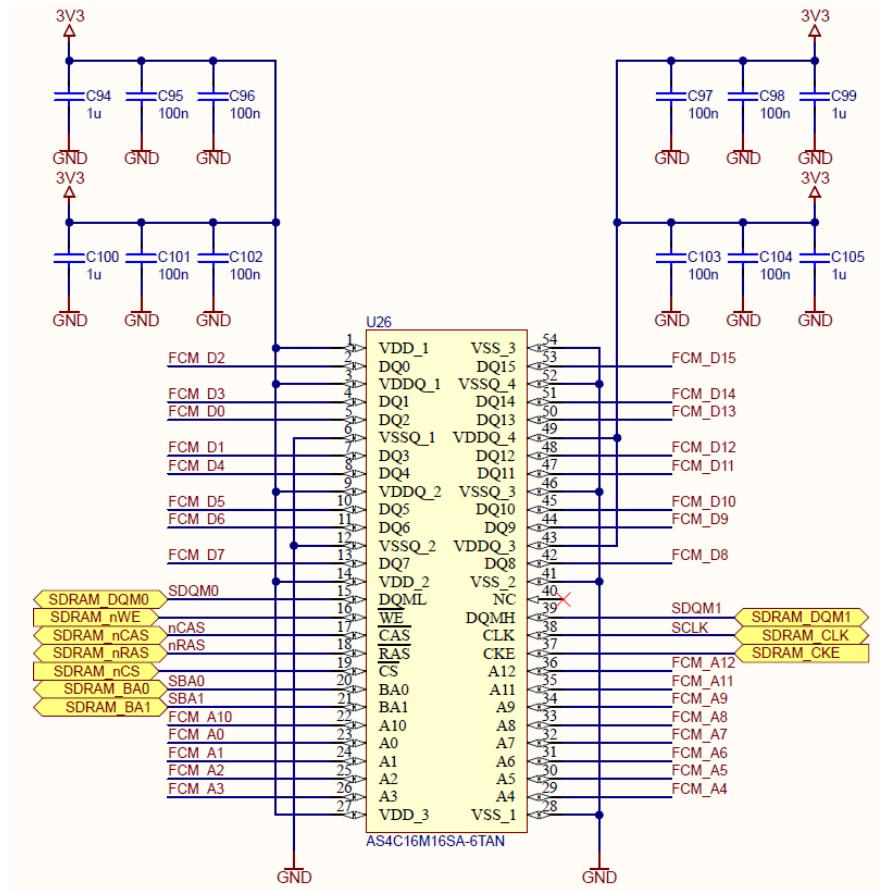


Abbildung 5.9.: Ansteuerung eines externen 16-Bit SDRAM-Bausteins am Beispiel IS42S16320D. Quelle: Schaltplan Leguan

In Tabelle 5.5 werden die Signale zur Ansteuerung des SDRAM-Bausteins beschrieben.

Signal	Kurzbeschreibung
/CS	Chip-Select (Input): Aktiv „low“ um den SDRAM-Baustein zu aktivieren.
/WE	Write Enable (Input): Aktiv „low“ wenn Daten ins SDRAM geschrieben werden.
/RAS	Row Adress Strobe (Input), auf dem Adressbus befindet sich die Row-Address.
/CAS	Column Adress Strobe (Input), auf dem Adressbus befindet sich die Column-Address.
CLK	Clock (Input)
CKE	Clock Enable (Input)
DQx	Datenleitungen (Input / Output)
Ax	Adressleitungen (Input)
BAx	Bank Select (Input)
DQML	Lower Byte, Input/Output Mask
DQMH	Upper Byte, Input/Output Mask

Tabelle 5.5.: Signale zur Ansteuerung des PSRAM-Bausteins

5.11. Schaltplan

Auf der Webseite des Leguan-Boards finden Sie die Schaltpläne für den Aufbau mit dem STM32H743. Die Schaltpläne des Leguan-Board sind von der Berner Fachhochschule entwickelt worden und dementsprechend verfügbar.

Teil II.

Assembler- und C-Programmierung

6. Einstieg Assembler-Programmierung

6.1. Einleitung

Dieses Kapitel befasst sich mit der Assembler-Programmierung im Allgemeinen, die Beispiele zur Assembler-Syntax sind jedoch auf den Cortex-Mx Microcontroller mit Thumb-2 Instruktionssatz bezogen. Die Assembler-Syntax orientiert sich am GNU-Assembler. Diese unterscheidet sich in verschiedenen Punkten von Produkten anderer Hersteller.

Die Assemblersprache ist eine hardwarenahe Programmiersprache. Assemblersprachen sind deshalb immer hardwareabhängig, d.h. sie sind für die jeweilige CPU-Familie entwickelt worden.

- Vorteil: Die vorhandene Hardware kann optimal ausgenutzt werden (Register, Architektur usw.).
- Nachteil: Assembler-Programme sind nicht portierbar (können nicht auf einer anderen CPU ausgeführt werden).

6.2. Sprachebenen

Eine CPU versteht nur den binären Maschinencode, welcher aber für den Menschen nicht verständlich ist. Der Mensch verwendet deshalb Programmiersprachen, die für ihn besser lesbar sind. Diese werden anschliessend von Assemblern oder Compilern in Maschinencode übersetzt.

Abbildung 6.1 zeigt die verschiedenen Sprachebenen:



Das Diagramm zeigt eine Tabelle mit sechs Zeilen, die die verschiedenen Sprachebenen von Maschinennähe bis Abstraktion darstellen. Die Spalten sind 'Maschinennähe' (links) und 'Sprachebene' (rechts). Ein vertikaler Pfeil auf der linken Seite zeigt nach unten und ist mit 'Maschinennähe' beschriftet. Ein vertikaler Pfeil auf der rechten Seite zeigt nach oben und ist mit 'Abstraktion' beschriftet.

	Nimm eine Flasche Bier aus dem Kühlschrank	Natürliche Sprache
	Bier - 1	Pseudocode
	Bier = Bier - 1;	Hochsprache
Maschinennähe	LDR r0,=bier LDR r1,[r0] SUB r1,r1,#1 STR r1,[r0]	Assembler
	0x4802 0100'1000'0000'0010 0x6801 0110'1000'0000'0001 0x3901 0011'1001'0000'0001 0x6001 0110'0000'0000'0001	Maschinencode

Abbildung 6.1.: Verschiedene Sprachebenen

1. Die natürliche Sprache ist für den Menschen am besten verständlich. Sie ist aber sehr unpräzise und redundant, sodass sie sich für die Programmierung nicht eignet.
2. In der Designphase wird oft Pseudocode verwendet. Die Abstraktionsebene ist hier sehr hoch. Pseudocode ist nicht normiert.

3. Hochsprachen bieten ebenfalls ein recht hohes Abstraktionsniveau. Compiler übersetzen die Hochsprache in Maschinencode, wobei die Effizienz des erzeugten Maschinencodes von der Programmiersprache und vom verwendeten Compiler abhängt. Gängige Hochsprachen für Embedded Systems sind C und C++. Hochsprachen sind normiert (z.B. ANSI-C). Dadurch lassen sich Programme mit vertretbarem Aufwand von einer Prozessor-Familie auf eine andere portieren.
4. Assemblersprachen sind für den Menschen bereits schwer verständlich. Andererseits sind gute Assembler-Programme bezüglich Performance und Ressourcen-Verbrauch einem Hochsprachen-Programm überlegen.
5. Der Maschinencode wird von der CPU ausgeführt und ist für den Menschen nicht oder nur sehr schwer verständlich.

6.3. Aufbau einer Assembler-Datei

Folgende Punkte beeinflussen den Aufbau einer Assembler-Datei:

- Assembler-Syntax, abhängig vom gewählten Prozessor.
- Assembler-Direktiven, abhängig von der Entwicklungsumgebung (je nach Hersteller).
- Programmierrichtlinien, abhängig von der Firma, die ein Produkt entwickelt.

Abbildung 6.2 zeigt den möglichen Aufbau einer Assembler-Datei:

```

/*
* Project      : Project name
*
* Program/Module : Module name
* File name    : File.s
* Version       : 1.00
* Created       : dd.mm.yyyy
* Author        : My name
*
* Description   : A short description of the module
*
* Modifications / History:
* Author date   description
* m.n dd.mm.yyyy ...
*/
/* PUBLIC DECLARATIONS */
.global mysub      @exported symbol
/* EXTERNAL DECLARATIONS */
.extern extsub     @imported symbol
/* INCLUDE DEFINITIONS */
.include "registers.h" @include controller regs
/* EQUATE DEFINITIONS */
.equ MAX,0x1000    @maximum allowable size
/* VARIABLE AND CONSTANT DEFINITIONS */
.data
myvar:.space 4      @space for 1 Word (32-Bit)
/* CODE */
.text
mysub: LDR r0,=myvar    @ load address of variable myvar
      MOV r1,#MAX      @ load value MAX to r1
      STR r1,[r0]       @ store value in variable myvar
      BL extsub        @ call subroutine extsub
      MOV pc,lr         @ return from subroutine
.end

```

The diagram illustrates the structure of an assembly source file with the following sections and their descriptions:

- Header (optional):** Contains project information like name, module name, file name, version, creation date, author, and a short description of the module.
- Exported Symbols (optional):** Contains declarations for symbols that are exported (e.g., `.global mysub @exported symbol`).
- Imported Symbols (optional):** Contains declarations for symbols that are imported (e.g., `.extern extsub @imported symbol`).
- Included Files (optional):** Contains include statements for header files (e.g., `.include "registers.h" @include controller regs`).
- EQU (optional):** Contains equate definitions (e.g., `.equ MAX,0x1000 @maximum allowable size`).
- Variables & Constants (optional):** Contains variable and constant definitions (e.g., `.data myvar:.space 4 @space for 1 Word (32-Bit)`).
- Code:** Contains the assembly code itself, including labels, instructions, and comments.

Abbildung 6.2.: Möglicher Aufbau einer Assembler-Datei

Header sind firmenabhängig. Normalerweise existieren in einer Firma Programmierrichtlinien, welche definieren, wie ein Header aussehen muss. Es sollten mindestens folgende Informationen enthalten sein:

- Projektname, zu welchem dieses Modul gehört
- Kurzbeschreibung des Moduls
- Autor, Datum und Version
- History mit einer Kurzbeschreibung der durchgeführten Änderungen

Im „PUBLIC DECLARATIONS“-Teil werden alle Subroutinen und Variablen aufgeführt, welche im vorliegenden Modul definiert sind und exportiert werden.

Im „EXTERNAL DECLARATIONS“-Teil werden alle Subroutinen und Variablen aufgeführt, welche von extern importiert werden.

Im „INCLUDE DEFINITIONS“-Teil können Files importiert werden, welche z.B. die Register des Controllers beschreiben. Mit „include“ wird das spezifizierte File komplett in das aktuelle File hineinkopiert.

Im „EQUATE DEFINITIONS“-Teil können Definitionen aufgelistet werden („.equ“ entspricht dem „#define“ in C).

Im „VARIABLE AND CONSTANT DEFINITIONS“-Teil wird Speicherplatz für Variablen und Konstanten reserviert.

Im „CODE“-Teil folgen die Programminstruktionen. Der Codeteil wird in 4 Spalten unterteilt, siehe dazu Kapitel 6.4.1.

6.4. Syntax

6.4.1. Aufbau einer Assemblerzeile

Die Assembler-Syntax wird auch Mnemonics genannt. Sie ist zeilenorientiert, deshalb entspricht jeder Zeile der Quelldatei eine Assembler-Operation. Beispiel für eine gültige Assemblerzeile:

Labelfield	Operation / Instruction	Operands	Comment
main:	LDR	r0, #MAX	@ load maximal value

Abbildung 6.3.: Aufbau einer Assemblerzeile

Labelfeld:

Das Labelfeld beginnt in der ersten Spalte und endet mit „:“. Im Labelfeld wird entweder der Name einer Subroutine oder ein Label eingetragen. Ein Label ist eine symbolische Adresse (Marke) und wird oft bei Verzweigungen als Sprungziel verwendet.

Operation / Instruktion:

Die Operation im Operationsfeld definiert die CPU-Instruktion oder die Assembler-Direktive. In Abbildung 6.3 ist „LDR“ eine Instruktion zum Laden eines Wertes in ein CPU-Register. Hier wird der konstante Wert „MAX“ in das Prozessorregister „r0“ kopiert.

Die möglichen Instruktionen einer CPU werden durch ihren Instruktionssatz beschrieben (siehe Kapitel 7 für den Thumb-2 Instruktionssatz).

Operanden:

Die Operanden im Operandenfeld definieren den oder die Operanden von Instruktionen oder Assembler-Direktiven. Im obigen Beispiel sind dies die Konstante „MAX“ und das Prozessorregister „r0“ für die Instruktion „LDR“.

ARM-CPPUs verarbeiten Drei-Operanden-Befehle. Viele Instruktionen beschränken sich auf einen oder zwei Operanden. Der Thumb-2-Befehlssatz kennt keine Instruktionen ohne Operanden.

Ein-Operand-Befehle: Dieser Instruktionstypus benötigt nur einen Operanden. Bei Verzweigungs- und Sprungbefehlen wird ein Label (symbolischer Name für eine Adresse) notiert. Beispiel:

```
B      start
SWI    0x10
```

Listing 6.1: Ein-Operand-Befehle

Zwei-Operanden-Befehle: Darunter fallen die meisten Datentransportbefehle. Sie kopieren Daten von einer Quelle zu einem Ziel (z.B. vom Arbeitsspeicher in ein CPU-Register). Beispiel:

```
MOV    pc,lr
LDR    r0,=0x20000000
LDR    r0,[r1]
```

Listing 6.2: Zwei-Operanden-Befehle

Drei-Operanden-Befehle: Datenverarbeitende Befehle werden mit drei Operanden notiert. Beispiel:

```
ADD    r1,r0,#1
```

Listing 6.3: Drei-Operanden-Befehle

Im obigen Listing wird das Register r0 (source-Register) um 1 inkrementiert, das Resultat wird in r1 (destination-Register) abgelegt.

Wie bei RISC-Prozessoren üblich, sind datenverarbeitend nur Register-Register-Operationen möglich. Register-Speicher-Operationen beschränken sich ausschliesslich auf den Datentransport.

Kommentar:

Der Kommentar im Kommentarfeld wird zum besseren Verständnis des Codes geschrieben. Der GNU-Assembler kennt drei Arten von Kommentaren: Blockkommentare, Zeilenkommentare und Zeilenendkommentare. Blockkommentare werden innerhalb von „/* ... */“ formuliert, wie bei ANSI-C. Zeilenkommentare stehen exklusiv auf einer Zeile und werden mit einem Nummernzeichen „#“ eingeleitet. Kommentare am Schluss einer Zeile, nach dem Code, werden mit einem Klammeraffen „@“ eingeleitet. Bei der Assembler-Programmierung ist es besonders wichtig den Code sinnvoll zu kommentieren. Der Kommentar soll aber nicht die Instruktion beschreiben, sondern was aus Sicht des Ablaufes bezweckt wird.

Beispiel:

```
/*
  Blockkommentar
*/
# Zeilenkommentar
LDR r0,=operand1          @ Zeilenendkommentar
```

Listing 6.4: Drei Formen der Assemblerkommentare

6.4.2. Symbole

Symbole sind Namen für Variablen, Subroutinen, Label usw. Folgende Zeichen sind für Symbole zugelassen:

- Buchstaben a..z, A..Z (keine Umlaute)
- Ziffern 0..9
- Unterstrich „_“
- Punkt „.“
- Dollarzeichen „\$“

Für Symbole gelten folgende Regeln:

- Das erste Zeichen darf keine Zahl sein.
- Symbole können beliebig lang sein, alle Zeichen sind signifikant.
- Es wird zwischen Gross- und Kleinschreibung unterschieden.

Der GNU-Assembler unterstützt auch lokale Symbole. Der alleinstehende Punkt hat als vordefiniertes Symbol eine spezielle Bedeutung: Er verkörpert als Wert den Programmzähler des aktuellen Bereiches (section). So ist beispielsweise „.=.+4“ äquivalent zu „space 4“.

6.4.3. Konstanten

Für Konstanten gelten folgende Regeln:

Typ	Kurzbezeichnung	Präfix	Beispiel
Zeichenkette	String	“	“Hallo“
Buchstabe	Char	'	'a'
Numerisch	Hex	0x	0x7FF
Numerisch	Bin	0b	0b0111011
Numerisch	Oct	0	0123
Numerisch	Dez	kein Präfix	123

Tabelle 6.1.: Regel für Konstanten

Konstanten können einfach mit dem „=“-Zeichen spezifiziert werden. Dies gilt für numerische Zahlenwerte wie auch für Speicherplatzadressen, die durch Symbole verkörpert werden.

Um Konstanten in ein Register zu laden, wird die Instruktion LDR verwendet. Die Instruktion LDR ist auch eine Pseudo-Instruktion. Je nach Wert, der geladen werden muss, wird der Assembler diese durch ein MOV ersetzen, so beispielsweise wenn die Zahl 1 geladen werden soll (MOV r0,#1), siehe Code Beispiel unten. Die Zahl 0x12345678 kann nicht innerhalb einer Assembler-Instruktion abgespeichert werden. Sie wird deshalb am Schluss des Programms, im folgenden Beispiel im Speicher auf den Adressen 0x080001e4 bis 0x080001e7 abgelegt. Dies kann in der Entwicklungsumgebung im Memory-Fenster geprüft werden (siehe Abbildung 6.4). Hinweis: ARM Cortex-Mx arbeiten im Little-Endian Format, d.h. dass das tiefstwertigste Byte auf der untersten Adresse zu liegen kommt. Somit steht auf Adresse 0x080001e4 der Wert 0x78. Um diese Konstante 0x1234567 zu laden, wird durch den Assembler automatisch eine PC-relative Adressierung verwendet (hier [pc,#4]).

Im folgenden Beispiel wird die Konstante 0x12345678 PC-relativ aus dem Speicher geladen, während der Wert Eins direkt über eine MOV-Instruktion geladen wird.

```

main:      MOV    r0 ,#1          @ Konstanter Wert 1 laden
loop:      LDR    r1 ,=0x12345678   @ Konstanter Wert 0x12345678 laden
           ADD    r0,r0,r1        @ Beide Werte addieren
           B     loop            @ Endlosschleife

```

Listing 6.5: Laden von Konstanten in ein Register

Der Assembler wird daraus Maschinencode erstellen. Dieser kann im Disassembly-Fenster der Entwicklungsumgebung eingesehen werden:

```

main:      MOV    r0 ,#1          @ Konstanter Wert 1 laden
080001dc:  mov   r0, #1
loop:      LDR    r1 ,=0x12345678   @ Wert 0x12345678 laden
080001de:  ldr   r1, [pc, #4]    ; (0x80001e4 <loop+6>)
           ADD    r0,r0,r1        @ Beide Werte addieren
080001e0:  add   r0, r0, r1
           B     loop            @ Endlosschleife
080001e2:  b.n   0x80001de <loop>

```

Listing 6.6: Darstellung des Codes im Disassembly-Fenster

Bemerkungen zum Disassembly-Fenster:

- In diesem Beispiel sind die Zeilen mit Endkommentar (schwarz) identisch mit denjenigen des Source-Codes.
- In den jeweils folgenden Zeilen (rot) ist der Code ersichtlich, welcher von der Entwicklungsumgebung generiert wurde. Normalerweise sind dies dieselben Assembler-Instruktionen, bei der Instruktion LDR sind sie unterschiedlich (Begründung siehe oben).
- In der ersten Spalte sind die Adressen ersichtlich, auf welchen dieser Code gespeichert wird. Das Hauptprogramm (main) beginnt somit auf Adresse 0x080001dc. Das Label „loop“ steht auf Adresse 0x080001de. Am Schluss des Programms wird wieder zum Label „loop“ gesprungen, d.h. ein Sprung (branch) auf die Adresse 0x080001de.
- Die Instruktion „LDR r1, [pc, #4]“ liegt auf der Adresse 0x080001de. Wenn diese Instruktion ausgeführt wird, zeigt der PC bereits auf die nächste auszuführende Instruktion „ADD r0,r0,r1“ auf Adresse 0x080001e0. Da die Konstante 0x12345678 ab Adresse 0x080001e4 gespeichert wird, ergibt sich ein Offset relativ zum PC von 0x080001e4 - 0x080001e0 = 4. Schön ist, dass die Entwicklungsumgebung diesen Offset berechnet, das müssen nicht die Programmierer*innen machen.

Address	0 - 3	4 - 7	8 - B	C - F
080001D0	A4000020	A0000020	00000000	01200149
080001E0	4018FCE7	78563412	002100F0	04B80E4B
080001F0	5B584350	04310D48	0DAB4218	9A42FFF4
08000200	F6AF0C4A	00F003B8	002342F8	043B0A4B
08000210	9A42FFF4	F9AF00F0	CBF800F0	E5F9FFF7
08000220	DDFFFFF7	FEBF0000	BC130008	00000020
08000230	A0000020	A0000020	BC000020	07480138

Abbildung 6.4.: Ausschnitt aus dem Memory mit der Konstanten 0x12345678 auf Adresse 0x080001e4 bis 0x080001e7

6.4.4. Operatoren und Operanden

Zusätzlich zum Instruktionssatz (Ausführung zur Laufzeit auf der CPU) können auch während des Assembliervorganges auf dem PC Brechungen durch den Assembler durchgeführt werden. Bedingung ist natürlich, dass die Operanden zur Zeit des Assembliervorganges bekannt sind. Häufig werden solche Rechnungen in Form von Ausdrücken zur Bestimmung von Speichergrößen oder Sprungdistanzen im Programm durchgeführt.

Innerhalb von Ausdrücken sind die nachfolgend aufgeführten Operatoren zulässig. Vor und nach den Operatoren sind **keine** Zwischenräume erlaubt!

Operator	Kurzbezeichnung
+	Addition
-	Subtraktion
*	Vorzeichenbehaftete (signed) Multiplikation
/	Vorzeichenbehaftete Division
%	Vorzeichenbehafteter Divisionsrest (Modulus)
&	Bitweise Und-Verknüpfung
	Bitweise Oder-Verknüpfung
^	Bitweise Exklusiv-Oder-Verknüpfung
&&	Logische Und-Verknüpfung
	Logische Oder-Verknüpfung
==, <>	Gleich, ungleich
>, >=	Grösser als, grösser als oder gleich wie
<, <=	Kleiner als, kleiner als oder gleich wie
<<	Logisches Schieben nach links
>>	Logisches Schieben nach rechts
~	Bitweise Negation, Einerkomplement

Tabelle 6.2.: Operatoren für die Assembler-Programmierung

Die Reihenfolge beim Auswerten von Ausdrücken ist wie folgt definiert:

1. Negation, Komplement (**Höchste Priorität**)
2. Multiplikation, Division, Modulus
3. Schieben links, Schieben rechts
4. Bitweise Und, Oder, Exklusiv-Oder
5. Addition, Subtraktion
6. Gleich, Ungleich
7. Relationalvergleiche
8. Logisch Und, Oder (**Tiefste Priorität**)

Das Resultat der Vergleichsoperationen ist beim GNU-Assembler immer ein numerischer Wert. 0 wenn falsch, -1 wenn wahr. Die logischen Operatoren **&&** und **||** erzeugen aber für wahr einen Wert von +1. Für eine logische Auswertung spielt dieser Unterschied aber keine Rolle, weil alles was ungleich Null ist, als wahr interpretiert wird.

Beispiel: Bit 3 und 11 des Registers r0 sollen gelöscht werden.

```
BIC r0, r0, #(1 << 3) | (1 << 11)
```

Listing 6.7: Beispiel Löschen einzelner Bits

Die Instruktion BIC löscht die Bits in Register r0, welche durch den Ausdruck **#(1 << 3) | (1 << 11)** angegeben werden. Die Postion der zu löschen Bits wird mit dem „<<“ Parameter definiert. Der Parameter „|“ verknüpft die zu löschen Bits.

6.5. Entwicklungsumgebungen

6.5.1. Übersicht

Hardwarenahe Programme (Firmware) werden üblicherweise auf einem PC mit Hilfe einer IDE (**I**ntegrated **D**evelopment **E**nvironment) entwickelt und anschliessend auf der Zielhardware (engl. Target) getestet (debug). Entwicklungsumgebungen bieten folgende Funktionalitäten an:

- Editor
- Assembler, Compiler und Linker/Locator
- Debugger
- Projektverwaltung

Abbildung 6.5 zeigt eine Übersicht der Dateien, die während der Entwicklungsphase entstehen.

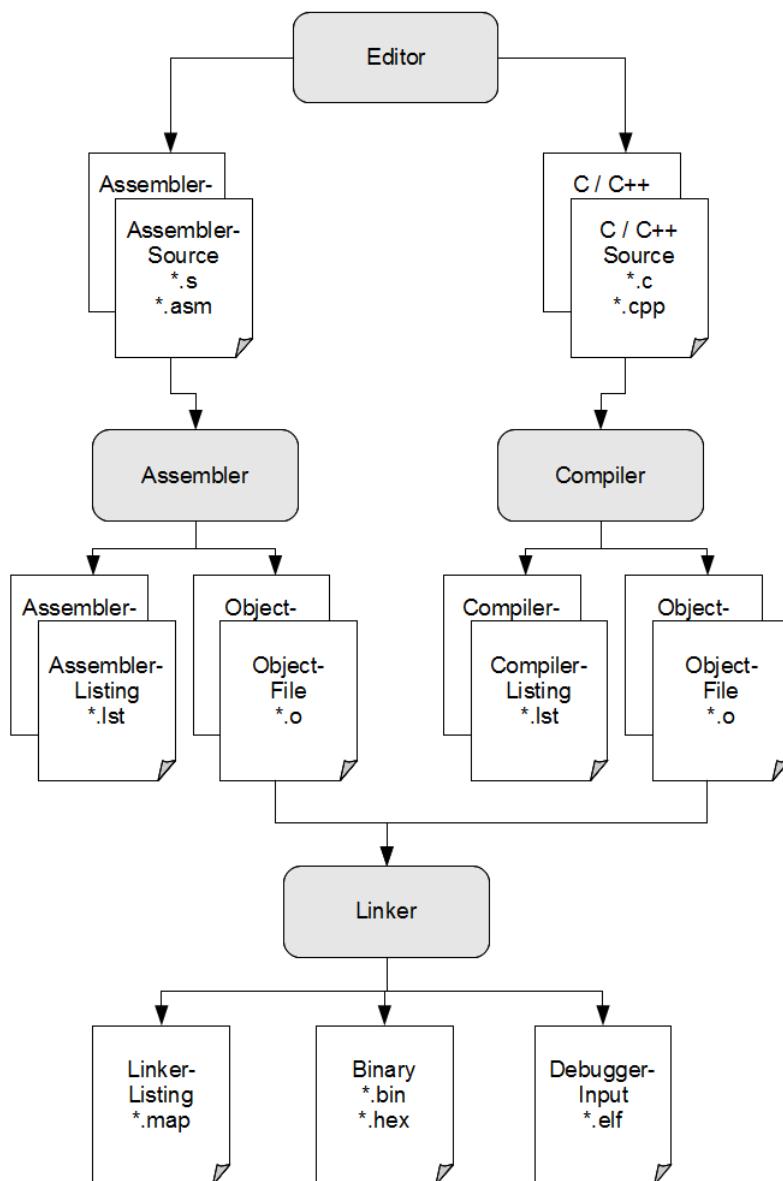


Abbildung 6.5.: Entwicklungsablauf

6.5.2. Editor

Mit Hilfe des Editors wird der Quellcode (Assembler oder Hochsprache) geschrieben. Viele Editoren sind syntax-sensitiv und färben Schlüsselwörter, Kommentare usw. farbig ein. Beispiel: Mit dem Editor wurde folgender Assembler-Code erstellt:

```
.global main
.thumb
.text

main:    LDR   r0,=1          @ Konstanter Wert 1 laden
loop:    LDR   r1,=0x12345678  @ Konstanter Wert 0x12345678 laden
        ADD   r0,r0,r1          @ Beide Werte addieren
        B     loop              @ Endlosschleife

.end
```

Listing 6.8: Assembler-Instruktionen im Editor erstellt

6.5.3. Assembler

Die Aufgabe des Assemblers ist es, den im Editor erstellten Assembler-Sourcecode bestehend aus Assembler-Instruktionen und Assembler-Direktiven in Maschinencode zu übersetzen. Ein Assembler ist immer CPU-spezifisch. Wenn Software für zwei unterschiedliche CPU-Familien entwickelt wird, benötigt man auch zwei verschiedene Assembler. Selbst Assembler für die gleiche CPU, aber von unterschiedlichen Herstellern, sind untereinander meist nicht vollständig kompatibel, da sie unterschiedliche Assembler-Direktiven verwenden (siehe Kapitel 8). Läuft der Assembler auf einem anderen Computer als das erzeugte Programm (Bsp.: IDE läuft auf einem PC, erzeugter Code läuft in einer Kaffeemaschine), so spricht man von einem Cross-Assembler.

Der Assembler übersetzt den Assembler-Sourcecode einer Quelldatei mit der Endung „.s“ oder „.asm“ in ein Object-File (.o). Je nach Einstellungen der Assembler-Optionen können auch zusätzliche Files (z.B. List-File, „.lst“) generiert werden. Das Assembler-Listing des obigen Beispieles ist nachfolgend dargestellt:

```
.global      main
.thumb
.text

main:    MOV   r0,#1          @ Konstanter Wert 1 laden
80001dc:  2001             mov    r0, #1

loop:    LDR   r1,=0x12345678  @ Konstanter Wert 0x12345678 laden
80001de:  4901             ldr    r1, [pc, #4]
        ADD   r0,r0,r1          @ Beide Werte addieren
80001e0:  1840             add    r0, r0, r1
        B     loop              @ Endlosschleife
80001e2:  e7fc             b.n   80001de <loop>
80001e4:  12345678         .word  0x12345678
```

Listing 6.9: Assembler-Listing

Jede Zeile mit Zeilenendkommentar im obigen Listing entspricht einer Source-Zeile (schwarz), wie sie im Editor eingegeben wurde. Bei den übrigen Zeilen sind in den Spalten des Assembler-Listings die folgende Informationen angegeben:

Spalten-Nr.	Inhalt
1	Adresse des Programmspeichers der Instruktion
2	Maschinencode, beispielsweise heisst „2001“ für die CPU, dass das Register r0 mit dem Wert 1 geladen werden soll. „1840“ heisst, dass zum Register r0 der Inhalt von Register r1 addiert werden soll, das Resultat wird in Register r0 gespeichert. Die Instruktionen im obigen Beispiel benötigen 2 Bytes Speicherplatz, was typisch ist für den ARM Thumb-2 Befehlssatz. Die nächste Instruktion beginnt immer an der nächsten Halfword-Adresse.
3	Assembler-Instruktion

Tabelle 6.3.: Spalten des Assembler-Listings

6.5.4. Compiler

Der Compiler übersetzt ein in einer Hochsprache geschriebenes Source-File und erzeugt wie schon der Assembler ein Object-File. Embedded-Anwendungen werden üblicherweise in den Hochsprachen C / C++ geschrieben. Entsprechend werden C oder C++ Compiler verwendet. Den C-Compiler kennen Sie bereits aus den unteren Semestern des Studiums.

6.5.5. Linker / Locator

Die Aufgabe des Linker/Locator ist es, alle Object-Files eines Projektes zusammenzubinden (Linker). Er weist aber auch dem Code und den Daten die definitiven, absoluten Adressen zu (Locator). Der Linker/Locator erzeugt Dateien

1. für den Download in den Programmspeicher (.bin oder .hex),
2. für den Debugger (.elf),
3. List- und Map-Files.

Das elf-Format ist ein Standard zum Einbinden von direkt ausführbarem Code, Objektcode, Bibliotheken, Speicherabbilder, aber auch Debuginformationen. ELF steht für „Executable and Linking Format“. Das elf-Format ist mittlerweile weit verbreitet und hat konzeptionell COFF und viele andere Debug-Exec-Formate abgelöst.

In der Regel ist der Dateiinhalt für die Programmierer*innen nicht interessant. Er ist mit dem Editor nicht lesbar, kann aber mit objdump.exe oder elfdump.exe ausgewertet werden.

```

C:\Program Files (x86)\Atollic\TrueSTUDIO for ARM Pro 4.3.1\PCTools\bin>objdump -rfh ASM_Konstanten.elf

ASM_Konstanten.elf:      file format elf32-little
architecture: UNKNOWN!, flags 0x000000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x080001e9

Sections:
Idx Name      Size    UMA      LMA     File off  Align
 0 .isr_vector 00000188 08000000 08000000 00008000 2**0
                CONTENTS, ALLOC, LOAD, READONLY, DATA
 1 .text       0000122c 08000188 08000188 00008188 2**2
                CONTENTS, ALLOC, LOAD, READONLY, CODE
 2 .init_array 00000004 080013b4 080013b4 000093b4 2**2
                CONTENTS, ALLOC, LOAD, DATA
 3 .fini_array 00000004 080013b8 080013b8 000093b8 2**2
                CONTENTS, ALLOC, LOAD, DATA
 4 .data        000000a0 20000000 080013bc 00010000 2**2
                CONTENTS, ALLOC, LOAD, DATA
 5 .bss         0000001c 200000a0 200000a0 000100a0 2**2
                ALLOC
 6 .user_heap_stack 00002000 200000bc 200000bc 000100a0 2**0
                ALLOC
 7 .ARM.attributes 00000032 00000000 00000000 000100a0 2**0
                CONTENTS, READONLY
 8 .debug_line   000015cd 00000000 00000000 000100d2 2**0
                CONTENTS, READONLY, DEBUGGING
 9 .debug_info   00002d45 00000000 00000000 0001169f 2**0
                CONTENTS, READONLY, DEBUGGING

```

Abbildung 6.6.: Informationen aus dem elf-File mit OJDUMP.EXE

6.5.6. Debugger

Mit Hilfe des Debuggers kann der erstellte Code getestet werden. So können Variablen, Register, Memory usw. der Hardware auf dem PC dargestellt werden. Ferner gehören das Setzen von Haltepunkten (Breakpoints) und die schrittweise Instruktionsausführung im Quellcode (Code-Stepping) zum Minimalumfang eines Debuggers. Findet das Debugging nicht auf dem Entwicklungsrechner, sondern auf dem Zielsystem statt, spricht man von Target-Debugging. Der Debugger auf dem PC agiert als Steuersoftware für den Download und die Kommunikation mit der Zielhardware. Im Zielsystem wird das Debugging durch die Prozessor-Hardware selbst oder durch zusätzliche Debuggingsoftware (Debug-Monitor) unterstützt. Der Debugger lädt das Programm über eine Debug-Schnittstelle auf das Zielsystem (heute üblicherweise über JTAG oder SWI) und ermöglicht ein kontrolliertes Testen auf dem Zielsystem.

6.5.7. Projektverwaltung

Innerhalb der Projektverwaltung können die Projektdaten zentral verwaltet werden. Das sind:

- Alle Files, die zum Projekt gehören (Source-Files, aber auch Bibliotheken und Listings)
- Setzen der Projektpfade (Library usw.)
- Mapping der verschiedenen Programm- und Datensegmente auf absolute Adressen
- Optionen wie Assembler-, Compiler- und Linker-Einstellungen

Moderne integrierte Entwicklungsumgebungen (IDE (Integrated Development Environment)) wie Eclipse, STM32CubeIDE, Segger Embedded Studio, MS Visual Studio, Keil oder Embedded Studio von Segger bieten diese Funktionalitäten in grossem Umfang inklusive Versionsverwaltung.

7. Instruktionssatz ARM V7M

In diesem Kapitel wird der Instruktionssatz von Cortex-Mx CPUs (d. h. die Befehle, welche die CPU ausführen kann) beschrieben. Die Instruktionen werden dabei in Gruppen ähnlicher Funktionalität aufgeteilt. Eine ausführliche Beschreibung aller Instruktionen finden Sie in [9].

Für weiterführende Angaben zur Architektur und zu Registern sei auf die Kapitel 5.3 und 5.4 verwiesen.

7.1. Grundlagen

7.1.1. Einleitung

Die Notation der Befehle erfolgt mit Hilfe von „Mnemonics“ (Befehlskürzel). Sie werden systematisch aus den Grossbuchstaben der ausgeschriebenen englischen Befehlsnamen abgeleitet, beispielsweise ADD, AAdd with Carry (ADC), Reverse SuBtract (RSB), Exclusive OR (EOR) oder Blt Clear (BIC).

Die Cortex-Mx Prozessoren verwenden den Thumb-2 Instruktionssatz. Thumb-2 basiert auf dem 16-Bit Thumb Instruktionssatz, fügt aber zusätzlich 32-Bit Instruktionen hinzu. Somit beinhaltet Thumb-2 sowohl 16-Bit als auch 32-Bit Instruktionen. Dies ermöglicht einerseits eine hohe Codedichte mit den 16-Bit Instruktionen, andererseits wird aber auch die hohe Performance der 32-Bit ARM-Instruktionen nachgebildet. Einige Instruktionen existieren sowohl als 16-Bit als auch als 32-Bit Version. Diese Instruktionen können durch die Suffix „.n“ auf 16-Bit oder „.w“ auf 32-Bit festgelegt werden.

Die UAL (Unified Assembler Language) ist ein Superset der Instruktionssätze ARM, Thumb und Thumb-2. Code, welcher in UAL geschrieben wurde, kann sowohl für ARM als auch für Thumb und Thumb-2 basierte Prozessoren übersetzt werden. Siehe auch Kapitel 8.3.

7.1.2. Syntax

Die Syntax einer Assemblerinstruktion ist wie folgt definiert:

```
Opcode Rd ,Rn {,N}
```

Listing 7.1: Syntax Assemblerinstruktion

Der Opcode ist eine Instruktion gemäss Kapitel 7.2.

Für Rd und Rn können alle Register von r0 bis r15 angegeben werden. N kann ein einfaches Register, ein konstanter Wert oder ein geshiftetes Register sein (siehe Kapitel 7.1.3).

7.1.3. Operanden

Instruktionen enthalten je nach Funktionalität einen, zwei oder drei Operanden (siehe auch Kapitel 6.4.1). Der erste Operand ist bei datenverarbeitenden Befehlen das Zielregister, der zweite und optional der dritte Operand sind die Quellen. „N“ kann entweder eine Konstante oder ein Register mit optionalem Shift sein.

Konstanten

- Ein 8-Bit Wert, welcher um eine variable Anzahl Bits nach links, in einem 32-Bit Wert, geschoben werden kann.
- Eine Konstante der Form 0x00XY00XY
- Eine Konstante der Form 0xXY00XY00
- Eine Konstante der Form 0xXYXYXYXY

In den oben beschrieben Konstanten sind X und Y hexadezimale Ziffern.

Register mit optionalem Shift

Bei ALU-Operationen kann der zweite Operand der ALU (Rm) vor der Operation durch den Barrel-Shifter geleitet werden. Der Datenwert kann dadurch, je nach Instruktion, um 1 bis 32 Bit nach links oder rechts geschoben werden (Multiplikation oder Division mit Zweierpotenzen). Das Schieben erfolgt immer in einem Taktzyklus, unabhängig vom Schiebewert. Der Wert im Ursprungsregister Rm wird durch den Barrel-Shifter nicht verändert.

In der ALU wird der Wert aus dem Register Rn und das Resultat des Barrel-Shifters N verarbeitet. Beispiel:

```
Vorher:
r5 = 5
r7 = 8
ADD r7, r7, r5, LSL #2      @ r7 = r7 + (r5 * 4)
                             @ Rd is r7, Rn is r7, N is R5 shifted

Nachher:
r5 = 5
r7 = 28
```

Listing 7.2: Beispiel Register mit optionalem Shift laden

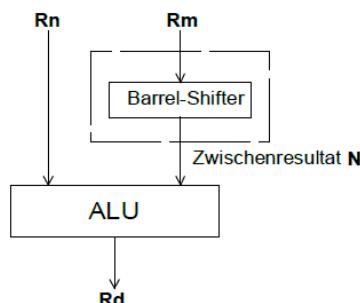


Abbildung 7.1.: ALU mit Barrel-Shifter

Befehl	Wirkung	Operation	Resultat	Schiebebereich
LSL	Logisches Schieben nach links	x LSL y	$x \ll y$	#0-31 oder Rs
LSR	Logisches Schieben nach rechts	x LSR y	$(unsigned)x \gg y$	#1-31 oder Rs
ASR	Arithmetisches Schieben nach rechts	x ASR y	$(signed)x \gg y$	#1-31 oder Rs
ROR	Rotieren nach rechts	x ROR y	$((unsigned)x \gg y) (x \ll (32 - y))$	#1-31 oder Rs
RRX	Erweitertes Rotieren nach rechts	x RRX	$(cflag \ll 31) ((unsigned) \gg 1)$	1 (fest gegeben)

Tabelle 7.1.: Barrel-Shifter

7.1.4. Bedingte Ausführung

Die meisten datenverarbeitenden Instruktionen können die Statusbits (Condition Flags) im PSR (Program Status Register, siehe Kapitel 5.4) aktualisieren, indem an den Befehlsnamen das Suffix „S“ angehängt wird. Einige Instruktionen wie „CMP“ verändern die Statusbits immer.

Condition code suffixes

Die Statusbits können verwendet werden, um Instruktionen abhängig von Bedingungen auszuführen (Conditional execution). Dies wird vor allem für die bedingten Sprungbefehle „Bxx“ verwendet (siehe Kapitel 7.6.3).

Suffix	Flags	Beschreibung
EQ	Z = 1	Gleich
NE	Z = 0	Nicht gleich
CS oder HS	C = 1	Grösser oder gleich, unsigned
CC oder LO	C = 0	Kleiner, unsigned
MI	N = 1	Negativ
PL	N = 0	Positiv oder Null
VS	V = 1	Overflow
VC	V = 0	Kein overflow
HI	C = 1 und Z = 0	Grösser, unsigned
LS	C = 0 oder Z = 1	Kleiner oder gleich, unsigned
GE	N = V	Grösser oder gleich, signed
LT	N != V	Kleiner, signed
GT	Z = 0 and N = V	Grösser, signed
LE	Z = 1 and N != V	Kleiner oder gleich, signed
AL	Can have any value	Immer, dies ist der default, wenn sonst nichts angegeben wird.

Tabelle 7.2.: Condition code suffixes

7.2. Befehlsübersicht

Die Tabellen 7.3 und 7.4 zeigen eine kurze Zusammenfassung der wichtigsten Thumb-2 Instruktionen. Eine Vollständige Liste der Befehle befindet sich im Anhang E.

Instruktion	Funktion
ADC	Add with carry
ADD	Add
AND	Logical AND
ASR	Arithmetic shift right
B	Branch
BFC	Bit field clear
BFI	Bit field insert
BIC	Bit clear
BKPT	Breakpoint
BL	Branch with link
BLX	Branch indirect with link
BX	Branch indirect
CBNZ	Compare and branch if non zero
CBZ	Compare and branch if zero
CLREX	Clear exclusive
CLZ	Count leading zeros
CMN	Compare negative
CMP	Compare
CPSID	Change processor state, disable interrupts
CPSIE	Change processor state, enable interrupts
EOR	Exclusive OR
LDM	Load multiple registers
LDR	Load register with word
LSL	Logical shift left
LSR	Logical shift right
MLA	Multiply with accumulate, 32-bit result
MLS	Multiply and subtract, 32-bit result
MOV	Move
MRS	Move from special register to general register
MSR	Move from general register to special register
MUL	Multiply, 32-bit result
MVN	Move NOT
NOP	No operation

Tabelle 7.3.: Insruktionen Thumb-2, Zusammenfassung Teil I

Instruktion	Funktion
ORN	Logical OR NOT
ORR	Logical OR
POP	Pop registers from stack
PUSH	Push registers onto stack
RBIT	Reverse bits
ROR	Rotate right
RSB	Reverse subtract
SADD<8 16>	Signed add 8 / 16
SBC	Subtract with carry
SDIV	Signed divide
STM	Store multiple registers
STR	Store register word
SUB	Subtract
SVC	Supervisor call
TEQ	Test equivalence
TST	Test
UADD<8 16>	Unsigned add 8 / 16
UHADD<8 16>	Unsigned halving add 8 / 16
UDIV	Unsigned divide
UMULL	Unsigned multiply (32 x 32), 64-bit result
USUB<8 16>	Unsigned subtract 8 / 16
WFE	Wait for event
WFI	Wait for interrupt

Tabelle 7.4.: Insruktionen Thumb-2, Zusammenfassung Teil II

7.3. Syntaxnotation

Für Instruktions-Beschreibungen wird die Notation gemäss Tabelle 7.5 benutzt.

Syntax	Kurzbeschreibung
[]	Indirekter, indizierter Zugriff
{ }	Fakultatives Element
< >	Element aus einer Menge, Aufzählung
	Auswahl aus Aufzählung (oder)

Tabelle 7.5.: Syntaxelemente für formale Befehlsbeschreibungen

7.3.1. Indirektion, Indexierung []

Es wird auf denjenigen Wert zugegriffen, auf den in der Klammer verwiesen wird. Der Verweis kann eine Speicherplatzadresse aber auch eine Bitposition sein. So verkörpert [Rn] einen Wert, der durch die Adresse in Rn referenziert wird.

7.3.2. Auswahl aus Aufzählung |

Mit dieser Notation wird exakt ein Element aus der Aufzählung gewählt. LDR | LDRB ist eine andere Formulierung für LDR{B}.

7.3.3. Element aus einer Aufzählung oder Menge <>

Mit dieser Notation wird exakt ein Element aus einer Menge gewählt. Die Menge ist normalerweise eine Zahlenmenge oder der Adressierbereich.

7.3.4. Optionale Elemente {}

Diese Notation erlaubt es, fakultative Elemente in einem Ausdruck zu wählen. Sie werden mit geschweiften Klammern dargestellt. So ist LDR{B} eine Kurzfassung für LDR | LDRB.

7.4. Datentransfer

Durch die RISC-Architektur kennt die Cortex-Mx CPU Register-Register-Transportbefehle (MOV), Speicher-Register-Transportbefehle (LDR) und Register-Speicher-Transportbefehle (STR). Speicher-Speicher-Transfers sind nicht in einer Instruktion möglich. Die Transferoperationen können als Word, Halfword oder Byte erfolgen.

Instruktion	Operanden	Operation	Beschreibung
MOV	Reg, Op2	Reg := Op2	Registerinhalt oder Konstante in Register kopieren
MVN	Reg, Op2	Reg := ~Op2	Registerkomplement oder Konstantenkomplement in Register kopieren
LDR	Reg, Address	Reg := [Address]	Speicherinhalt in Register kopieren
STR	Reg, Address	[Address] := Reg	Registerinhalt in Speicher kopieren
LDM	Rbase, {Rlist}	[Rbase] → {Rlist}	Mehrere Register vom Speicher laden
STM	Rbase, {Rlist}	{Rlist} → [Rbase]	Mehrere Register in Speicher kopieren
POP	{Rlist}	[SP] → {Rlist}	Mehrere Register vom Stack lesen
PUSH	{Rlist}	{Rlist} → [SP]	Mehrere Register auf den Stack kopieren

Tabelle 7.6.: Datentransferbefehle

7.4.1. Registertransportbefehle MOV, MVN

Sie dienen zum Datentransport von Registern untereinander wie auch zum Laden von Registern mit konstanten Werten.

Die allgemeine Syntax lautet:

```
<MOV | MVN>{<cond>} {S} Rd , N
```

Der Operationscode MOV lädt einen Datenwert, MVN lädt den Komplementwert. Für die bedingte Ausführung sind in <cond> alle Bedingungen nach Tabelle 7.2 erlaubt. Sollen die Condition Codes neu gesetzt werden wird zusätzlich das Suffix S spezifiziert. Rd verkörpert immer das Zielregister. Der Operand N kann ein Register, eine Konstante oder ein geschobener Registerwert sein.

Beispiele:

```
MOV r0,r1          @ r0 := r1
MOVS r0,#10        @ r0 := 10, Flags neu setzen
MVN r1,r0          @ r1 := ~r0
MOV pc, lr          @ Ruecksprung aus Subroutine
```

Listing 7.3: Beispiel Instruktion MOV und MOVN

7.4.2. Lade- und Speicherbefehle LDR, STR

Lade- und Speicherbefehle dienen zum Transferieren von Datenwerten aus dem Speicher in ein Register und umgekehrt. Das Laden eines Konstantenwertes in ein Register, das nicht direkt mit einer MOV/MVN-Instruktion möglich ist, wird vom Assembler ebenfalls in eine LDR-Instruktion umgesetzt.

Alle Lade- und Speicherbefehle, ausser LDRD und STRD, können für Words, Halfwords und Bytes angewandt werden.

Instruktion	Beschreibung	Wirkung
LDR	Word in Register laden	Rd := mem32[Adresse]
STR	Word aus Register speichern	mem32[Adresse] := Rd
LDRB	Byte in Register laden	Rd := mem8[Adresse]
STRB	Byte aus Register speichern	mem8[Adresse] := Rd
LDRH	Halfword in Register laden	Rd := mem16[Adresse]
STRH	Halfword aus Register speichern	mem16[Adresse] := Rd
LDRSB	Vorzeichenbehaftetes Byte in Register laden	Rd := (signExtend) mem8[Adresse]
LDRSH	Vorzeichenbehaftetes Halfword in Register laden	Rd := (signExtend) mem16[Adresse]
STRD	Zwei Register speichern (DSP)	Mem32[Adresse] := Rd Mem32[Adresse+4] := R(d+1)
LDRD	Zwei Register laden (DSP)	Rd = Mem32[Adresse] R(d+1) = Mem32[Adresse+4]

Tabelle 7.7.: Lade- und Speicherbefehle

Die Syntax der LDR/STR Instruktionen lautet:

```
<LDR|STR>{<cond>} {B} Rd, <Adresse>
LDR{<cond>} {SB|H|SH} Rd, <Adresse>
STR{<cond>} H Rd, <Adresse>
STR{<cond>} D Rd, <Adresse>
```

Die Angabe der Speicherplatzadresse kann auf vielfältige Art erfolgen. Im einfachsten Fall wird eine konstante Speicherplatzadresse angegeben oder ein Register, welches eine Speicherplatzadresse enthält. Die Adressiermodi werden im Kapitel 7.4.3 detailliert gezeigt.

Die LDRD-/ STRD-Instruktionen sind eine Erweiterung des DSP-Befehlssatzes. Sie erlauben das Laden/Speichern zweier Register im Speicher als aufeinanderfolgende Datenworte. Bedingung für Rd ist eine gerade Registernummer. Bei LDRD mit Rd=r2 werden die Registerinhalte aus r2 und r3 im Speicher abgelegt. Ist die Registernummer ungerade, ist das Verhalten der Operation undefined.

Beispiele:

Die LDR- und STR-Instruktionen können Daten von Speicherplatzadressen laden oder auf Speicherplatzadressen speichern. Bedingung für die Speicherplatzadresse ist, dass sie dieselbe Ausrichtung (Alignment) im Speicher hat wie der Datentyp selbst. Das heisst, Words und Halfwords müssen auf einer geraden Adresse liegen. Bytezugriffe können beliebig erfolgen.

```
/* Register r5 mit Adresse laden */
LDR r5,=0xA0001000

/* Register r0 mit Wert aus Speicherstelle laden, die durch r5 adressiert wird. */
LDR r0,[r5]

/* Wert aus Register r0 in Speicherstelle schreiben, die durch r5 adressiert wird. */
STR r0,[r5]

/* Wert aus Register r0 mit Wert in Speicherstelle schreiben, das durch das Symbol myvar
referenziert wird (myvar muss im gleichen Segment definiert sein). */
STR r0,myvar
```

Listing 7.4: Beispiel Instruktion LDR und STR

7.4.3. Adressiermodi für Lade- und Speicherbefehle

Die Cortex-Mx Architektur sieht vielfältige Adressierungsmöglichkeiten für den Speicherzugriff vor. Die Speicherplatzadresse, auf welche zugegriffen wird, setzt sich immer aus einer Basisadresse und einem Offset zusammen. Die Basis wird in einem Register gehalten. Ausser pc sind hierzu alle Register zulässig.

Die Adressiermodi werden nach ARM-Leseart „Indexing-Methoden“ genannt. Gemäss Tabelle 7.8 existieren drei grundsätzliche Indexing-Methoden mit zahlreichen Varianten der Offsetberechnung.

Indexmethode	Daten	Basisadressregister nach der Operation	Beispiel
Preindex mit Writeback	mem[Basis+Offset]	Basis+Offset	LDR r0,[r1,#4]!
Preindex	mem[Basis+Offset]	Basis	LDR r0,[r1,#4]
Postindex	mem[Basis]	Basis+Offset	LDR r0,[r1],#4

Tabelle 7.8.: Adressierungsmethoden für den Speicherzugriff

Der Offset ist immer eine vorzeichenbehaftete Grösse. So kann der Offset als ± 12 -Bit Wert um die Basisadresse notiert werden.

Beim Preindex wird die Speicherplatzadresse aus dem Basisregister und Offset berechnet. Nach dem Zugriff bleibt der Wert des Basisregisters unverändert. Wird mit „!“ ein Writeback angegeben, wird die berechnete Speicherplatzadresse als neuer Wert in das Basisregister zurückgeschrieben und zwar bevor der Speicherzugriff erfolgt.

Bei Postindex wird ebenfalls die Adresse in das Basisregister übernommen, aber erst nachdem der Speicherzugriff erfolgt ist.

Postindex und Preindex mit Writeback werden vor allem bei der Arbeit mit Arrays verwendet. Preindex wird häufig für den Zugriff in Datenstrukturen benutzt.

Beispiele:

Das folgende Beispiel zeigt den Zugriff mit allen drei Adressiermodi nach Tabelle 7.8 mit den Registerwerten vor und nach der Instruktionsausführung. Alle Instruktionen haben dieselbe Ausgangslage.

```

Vorher:
r0 = 0x00000000
r1 = 0x00010000
mem32[0x00010000]=0x10101010
mem32[0x00010004]=0x20202020

Preindex mit Writeback:
LDR r0,[r1,#4]!

Nachher:
r0 = 0x20202020
r1 = 0x00010004

Preindex:
LDR r0,[r1,#4]

Nachher:
r0 = 0x20202020
r1 = 0x00010000

Postindex:
LDR r0,[r1],#4

Nachher:
r0 = 0x10101010
r1 = 0x00010004

```

Listing 7.5: Beispiel Preindex mit Writeback

Für die Offsetberechnung sind zahlreiche Varianten möglich. Neben Direktwerten sind auch Register und geschobene Registerwerte möglich. Tabelle 7.9 zeigt eine Übersicht der Adressmodi.

Adressiermodus	Syntax
Preindex mit konstantem Offsetwert	[Rn,#±<offset12>]
Preindex mit Registeroffset	[Rn,±Rm]
Preindex mit skaliertem Registeroffset	[Rn,±Rm, shift #<shift_wert>]
Preindex Writeback mit konstantem Offsetwert	[Rn,#±<offset8>]!
Postindex mit konstantem Offsetwert	[Rn],#±<offset8>

Tabelle 7.9.: Syntax der Adressiermethoden für einfachen Speicherzugriff mit Word

Für andere Datentypen gelten Beschränkungen. So können STRSB, STRSH, STRH, STRB den Barrel-Shifter nicht benutzen. Ebenso ist der konstante Offset um ±8-Bit um den Basisregisterwert beschränkt.

Adressiermodus	Syntax
Preindex mit konstantem Offsetwert	[Rn,#±<offset8>]
Preindex mit Registeroffset	[Rn,±Rm]
Preindex Writeback mit konstantem Offsetwert	[Rn,#±<offset8>]!
Postindex mit konstantem Offsetwert	[Rn],#±<offset8>

Tabelle 7.10.: Zulässige Adressiermethoden für signed /unsigned Halfword und signed/unsigned Byte Daten

Zusammenfassende Beispiele:

```

LDR r2,[r0,#1024]      @ Preindex mit konstantem Offset,
@ r2 := [r0 + 1024]

LDR r2,[r0,r1]          @ Preindex mit Registeroffset,
@ r2 := [r0 + r1]

```

```

LDR    r2,[r0,r1,LSL#2]  @ Preindex mit skaliertem Registeroffset,
@ r2 := [r0 + 4*r1]

LDR    r2,[r0,#252]!     @ Preindex Writeback mit konstantem Offsetwert,
@ 1) r2:= [r0 + 252] 2) r0 = r0 + 252

LDR    r2,[r0],#252      @ Postindex mit konstantem Offsetwert,
@ 1) r2 := [r0] 2) r0 = r0 + 252

```

Listing 7.6: Beispiel verschiedene Adressiermodi

7.4.4. Mehrfach Lade- und Speicherbefehle LDM, STM

LDM und STM dienen zum Transferieren von ganzen Register- oder Speicherblöcken mit einer Instruktion. Der Transfer mit einer Instruktion ist effizienter als der Transfer mit Einzelanweisungen.

Die Instruktionsausführung braucht $2 + N * t$ Taktzyklen, wobei N die Anzahl Register und t die benötigte Anzahl Taktzyklen für den externen Speicherzugriff verkörpern. Während der Instruktionsausführung erfolgt keine Reaktion auf einen möglichen Interrupt. Er wird erst bedient, wenn die LDM-/STM-Instruktion vollständig abgearbeitet ist. Daher verschlechtert sich durch LDM-/STM-Nutzung die Interrupt-Latenzzeit.

Die Syntax der LDM-/STM-Instruktionen lautet:

```
<LDM | STM>{<cond>}<Adressmodus> Rn{!}, <Registerblock>
```

Listing 7.7: Beispiel Instruktion LDM und STM

Der Adressmodus beschreibt gemäss Tabelle 7.11 wie die fortlaufende Speicheradresse berechnet wird. Dabei kann festgelegt werden, ob die im Speicher nachfolgende Speicheradresse (Inkrement) oder vorhergehende (Dekrement) Speicheradresse benutzt wird. Weiter wird unterschieden, ob die Adressberechnung für die neue Adresse vor dem Zugriff (before) oder nach dem Zugriff (after) erfolgt.

Der Adressendwert kann bei Bedarf analog dem Preindex mit Writeback mit dem Ausrufezeichen Rn! zurückgeschrieben werden.

Adressmodus	Beschreibung	Startadresse	Endadresse	Rn!
IA	Increment after	Rn	$Rn + 4 * N - 4$	$Rn + 4 * N$
DB	Decrement before	$Rn - 4 * N$	$Rn - 4$	$Rn - 4 * N$

Tabelle 7.11.: Adressmodi für LDM-/STM-Instruktionen

Das Register Rn beinhaltet die Speicheradresse, wo die Daten im Speicher gelesen oder abgelegt werden. Folgt dem Register ein Writeback-Operator (!), wird der Zeigerendwert angepasst, analog der LDR-/STR-Instruktion.

Der Registerblock wird in geschweiften Klammern aufgeführt. Er nennt alle Register, die transportiert werden sollen, entweder durch Aufzählung mit Kommata (r0,r3,r6) oder als Bereich mit Bindestrich (r0-r3).

Die Register werden immer in aufsteigender Reihenfolge, beginnend bei der Startadresse, transportiert. Beim Dekrement erfolgt zuerst eine Subtraktion für die Startadresse, so dass beim Dekrement die Register in aufsteigender Reihenfolge korrekt abgelegt werden.

Beispiele:

Die folgenden Beispiele zeigen die Wirkung der Adressmodi nach Tabelle 7.11 beim Speichern der Register r0,r1,r4 im Speicher ab Adresse 0x00010000.

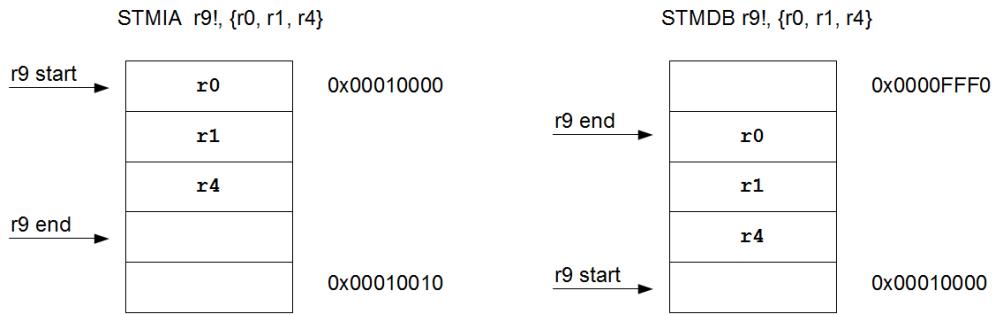


Abbildung 7.2.: Mehrfach Lade- und Speicherbefehle LDM und STM

Aufgabe:

Der Registerblock r0-r4 soll an die Speicherstelle 0x0001A000 aufsteigend abgelegt werden. Das Register r6 wird als Zeigerregister benutzt und soll am Schluss der Instruktion die letzte Speicheradresse beinhalten, d.h. auf den Inhalt von r4 zeigen.

Lösung:

Da r6 am Schluss auf eine neue Adresse zeigen muss, wird der Writeback-Operator (!) benutzt. Der Inkrementmodus wird mit increment before (IB) gewählt, weil r6 auf das letzte Element zeigen soll. Damit in diesem Modus das erste Register an die Adresse 0x0001A000 geschrieben wird, ist als Startadressenoffset -4 zu notieren.

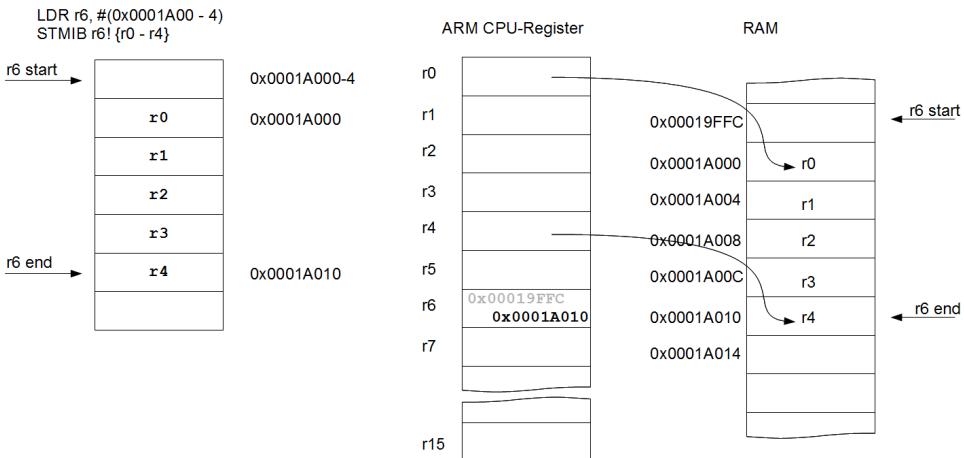


Abbildung 7.3.: Beispiel Speicherbefehle STMIB

7.4.5. Stackoperationen

Stackzugriffe werden bei Cortex-Mx Prozessoren üblicherweise nicht mit den LDM-/STM-Instruktionen ausgeführt, sondern es werden die Instruktionen PUSH und POP verwendet. Die PUSH und POP Instruktionen arbeiten gemäß AAPCS mit einem Full Descending Stack, d.h. die Daten werden hin zu tieferen Adressen auf dem Stack abgelegt, und der Stack-Pointer zeigt am Schluss einer Instruktion auf den zuletzt geschriebenen Wert.

Beispiel:

Die Register r4 bis r6 sind auf dem Stack zu retten und anschliessend zurückzulesen. Annahme: Der aktuelle Wert des Stack-Pointers ist 0x00010000.

```
PUSH {r4 - r6} @ push r4 to r6 to stack
...
POP {r4 - r6} @ pop r4 to r6 from stack
```

Listing 7.8: Die Stack-Operationen PUSH und POP

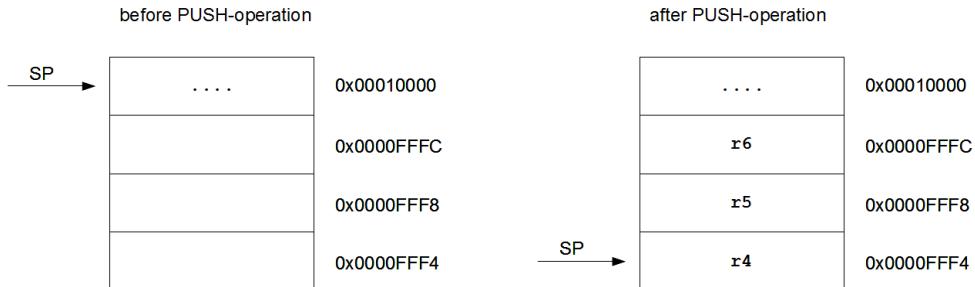


Abbildung 7.4.: Beispiel PUSH Instruktion

Beachten Sie, dass die PUSH und POP-Instruktionen automatisch mit dem Stack-Pointer arbeiten. Den beiden Instruktionen kann entweder ein einzelnes Register oder eine Liste von Registern übergeben werden.

Für das Arbeiten in Subroutinen wird in der Regel auch die Rücksprungadresse (im Link-Register LR) auf dem Stack abgelegt. Der Rücksprung kann durch die POP-Instruktion ausgeführt werden (siehe auch Kapitel 9.2), indem die Rücksprungadresse in den Program-Counter (PC) kopiert wird.

Beispiel:

Die Register r4 bis r6 sind in einer Subroutine inklusive die Rücksprungadresse auf dem Stack zu retten. Der Rücksprung aus der Subroutine erfolgt mit Hilfe der POP-Instruktion:

```
subr:
    PUSH {r4 - r6, lr} @ push r4 to r6 and lr on stack
    ...
    POP {r4 - r6, pc} @ pop r4 to r6 and lr, return
```

Listing 7.9: Stack-Operationen PUSH und POP in einer Subroutine, die Rücksprungadresse wird ebenfalls auf dem Stack abgelegt.

7.5. Arithmetische und Logische Instruktionen

Die arithmetisch-logischen Instruktionen stellen den wesentlichen Teil der datenverarbeitenden Instruktionen dar. Wie bei RISC-Prozessoren üblich, können diese Operationen nur auf Register angewandt werden. Speicherzugriffende ALU-Operationen sind nicht möglich. Alle Operanden haben mit 32-Bit immer ein einheitliches Format. Dies gilt auch für die Resultate. Einzelne arithmetische Operationen liefern ein 64-Bit Resultat, welches in einem Registerpaar abgelegt wird.

Als Spezialität der ARM-CPU ist zu nennen, dass eine logische oder arithmetische Instruktion standardmäßig nicht die Statusbits anhand des Resultates setzt. Dies muss mit einem „S“-Suffix bei der Instruktion erzwungen werden. Erweiterte Operationen, die nicht zum ARM-Kernbefehlssatz gehören, z.B. DSP-Instruktionen (QADD, SMLA,...) erlauben kein S-Suffix und setzen außer dem Q-Flag keine Statusbits.

Die in Tabelle 7.12 aufgeführten Instruktionen werden in den nachfolgenden Kapiteln näher beschrieben.

Instruktion	Beschreibung
ADD	32-Bit Addition
ADC	32-Bit Addition mit Übertrag
AND	Logisches bitweises UND
BIC	Logisches bitweises Bit nullsetzen (NICHT UND)
CMP	Vergleich
CMN	Negierter Vergleich
EOR	Logisches bitweises EXODER
MUL	32-Bit Multiplikation
MLA	32-Bit Multiplikation/Addition
ORR	Logisches bitweises ODER
QADD	32-Bit Addition mit Überlaufbegrenzung (DSP)
QDADD	32-Bit Addition mit Überlaufbegrenzung (DSP)
QDSUB	32-Bit Subtraktion mit Überlaufbegrenzung (DSP)
QSUB	32-Bit Subtraktion mit Überlaufbegrenzung (DSP)
RSB	Umgekehrte 32-Bit Subtraktion
SMLA	Vorzeichenbehaftete 16-Bit Multiplikation / 32-Bit Addition (DSP)
SMLAL	Vorzeichenbehaftete 16-Bit Multiplikation / 64-Bit Addition (DSP)
SMLAW	Vorzeichenbehaftete 32x16-Bit Multiplikation / Addition (DSP)
SMUL	Vorzeichenbehaftete 16-Bit Multiplikation (DSP)
SUB	32-Bit Subtraktion
UMULL	Vorzeichenlose long-Multiplikation

Tabelle 7.12.: Zusammenfassung der arithmetisch-logischen Instruktionen

7.5.1. Integer-Arithmetik

Diese Instruktionen beinhalten die Grundrechenoperationen sowie arithmetische Vergleichsoperationen.

Instruktion	Operanden	Beschreibung	Operation
ADD	Rd, Rn, N	32-Bit Addition	$Rd = Rn + N$
ADC	Rd, Rn, N	32-Bit Addition mit Übertrag	$Rd = Rn + N + \text{Carryflag}$
MUL	Rd, Rn, Rs	32-Bit Multiplikation	$Rd = Rn * Rs$
MLA	Rd, Rn, Rs, Rm	32-Bit Multiplikation / Addition	$Rd = Rn * Rs + Rm$
SMLA<x><y>	Rd, Rn, Rs, Rm	Vorzeichenbehaftete 16-Bit Multiplikation / 32-Bit Addition	$Rd = (Rn < x > *Rs < y >) + Rm$

Tabelle 7.13.: Integer-Arithmetik Instruktionen, Teil I

Instruktion	Operanden	Beschreibung	Operation
SMLAL	RdLo, RdHi, Rm, Rs	Vorzeichenbehaftete 32-Bit Multiplikation / 64-Bit Addition	$[RdHi, RdLo] = [RdHi, RdLo] + Rm * Rs$
SMLAL<x><y>	RdLo, RdHi, Rm, Rs	Vorzeichenbehaftete 16-Bit Multiplikation / 64-Bit Addition	$[RdHi, RdLo] = [RdHi, RdLo] + Rm < x > *Rs < y >$
SMLAW<y>	RdLo, RdHi, Rm, Rs	Vorzeichenbehaftete 32x16-Bit Multiplikation / Addition	$Rd = Rn + (((signed)Rm * Rs < y >) >> 16)$
SMUL<x><y>	RdLo, RdHi, Rm, Rs	Vorzeichenbehaftete 16-Bit Multiplikation	$Rd = Rm < x > *Rs < y >$
SMULL	RdLo, RdHi, Rm, Rs	Vorzeichenbehaftete long-Multiplikation	$[RdHi, RdLo] = Rm * Rs$
UMLAL	RdLo, RdHi, Rm, Rs	Vorzeichenlose long-Multiplikation / Addition	$[RdHi, RdLo] = [RdHi, RdLo] + Rm * Rs$
UMULL	RdLo, RdHi, Rm, Rs	Vorzeichenlose long-Multiplikation	$[RdHi, RdLo] = Rm * Rs$
QADD	Rd, Rm, Rn	32-Bit Addition mit Überlaufbegrenzung	$Rd = sat32(Rm + Rn)$
QDADD	Rd, Rm, Rn	32-Bit Addition mit Überlaufbegrenzung	$Rd = sat32(Rm + sat32(2 * Rn))$
QDSUB	Rd, Rm, Rn	32-Bit Subtraktion mit Überlaufbegrenzung	$Rd = sat32(Rm - sat32(2 * Rn))$
QSUB	Rd, Rm, Rn	32-Bit Subtraktion mit Überlaufbegrenzung	$Rd = sat32(Rm - Rn)$
SBC	Rd, Rn, N	32-Bit Subtraktion mit Übertrag	$Rd = Rn - N - !(Carryflag)$
SUB	Rd, Rn, N	32-Bit Subtraktion	$Rd = Rn - N$
RSB	Rd, Rn, N	Umgekehrte Subtraktion	32-Bit $Rd = N - Rn$
RSC	Rd, Rn, N	Umgekehrte Subtraktion mit Übertrag	32-Bit $Rd = N1 - Rn - !(Carryflag)$
CMP	Rn, N	Vergleich	Flags gemäss Resultat $Rn - N$
CMN	Rn, N	Negierter Vergleich	Flags gemäss Resultat $Rn + N$
TEQ	Rn, N	Test auf Gleichheit	Flags gemäss Resultat Rn^N
TST	Rn, N	Test auf 1-Bits	Flags gemäss Resultat $Rn \& N$

Tabelle 7.14.: Integer-Arithmetik Instruktionen, Teil II

Der Operand N kann ein Direktwert (Konstante mit #), ein Register oder ein geshiftetes Register sein. 64-Bit Resultate werden in einem Registerpaar abgelegt.

Die Suffix <x>,<y> können B oder T sein. Bei B werden nur die unteren 16 Bit [0-15] des Registers verwertet. Bei T werden die oberen 16 Bit ([16-31] des Registers für die Rechnung benutzt. <x> steht für das Register Rm, <y> für das Register Rs.

DSP-Befehle haben ein Q-Prefix. Sie arbeitet mit „gesättigter Arithmetik“, d.h. bei einer Rechnung kann kein

Bereichsüberlauf stattfinden. Wird der Wertebereich überschritten, erfolgt eine Begrenzung auf die Minimal- oder Maximalwert des Zahlenbereiches (saturated 32-bit arithmetic).

Weil der Operand N eine Vielzahl von Formen annehmen kann, ist es sinnvoll, die Subtraktion auch umgekehrt ausführen zu können. Hierzu sind die Reverse-Subtract-Instruktionen RSB und RSC implementiert. Die Befehle CMP, CMN, TEQ, TST führen intern die Befehle SUB, ADD, EOR und AND aus, modifizieren als Ergebnis aber nur die Statusbits. MUL und MLA haben die Einschränkung, dass Rd nicht pc oder Rn sein darf.

Merke: Die Statusbits werden nur bei den Test- und Vergleichsoperationen standardmäßig gesetzt. Sollen die Flags auch bei arithmetischen Operationen gesetzt werden, ist der Opcode mit dem Suffix S zu notieren.

Beispiel:

Die einfache Subtraktion erfolgt mit SUB. Der folgende Code subtrahiert den Wert in Register r2 vom Wert in Register r1 und speichert das Resultat in Register r0.

```
Vorher:
r0 = 0x00000000
r1 = 0x00000005
r2 = 0x00000002

SUB r0,r1,r2 @ r0 = r1 - r2

Nachher:
r0 = 0x00000003
```

Beispiel:

Der Instruktionssatz kennt keine Negierungsinstruktion. Diese kann aber durch eine umgekehrte Subtraktion erreicht werden, indem vom Wert 0 subtrahiert wird, auf der Grundlage $-z = 0 - z$.

```
Vorher:
r0 = 0x00000002

RSB r0,r0,#0 @ r0 = -r0

Nachher:
r0 = 0xFFFFFFFF
```

Listing 7.10: Beispiel Instruktion RSB

7.5.2. Logische Instruktionen

Die logischen Instruktionen beinhalten die logischen Verknüpfungsoperationen UND, ODER, EXOR und NOT. Mit ihnen lassen sich Operationen auf Bitmuster durchführen (z. B. löschen, setzen oder komplementieren von Bits) oder Bedingungen für Programmverzweigungen aufstellen.

Syntax der logischen Instruktionen:

```
<Instruktion>{<cond>} {S} Rd ,Rn ,N
```

Instruktion	Operanden	Beschreibung	Operation
AND	Rd, Rn, N	Logische UND-Verknüpfung zweier 32-Bit Werte	$Rd = Rn \& N$
ORR	Rd, Rn, N	Logische ODER-Verknüpfung zweier 32-Bit Werte	$Rd = Rn N$
EOR	Rd, Rn, N	Logische Exklusiv-ODER-Verknüpfung zweier 32-Bit Werte	$Rd = Rn \hat{N}$
BIC	Rd, Rn, N	Logisches Bit Nullsetzen (UND NICHT)	$Rd = Rn \& \sim N$

Tabelle 7.15.: Logische Instruktionen

Der Operand N kann ein Direktwert (Konstante mit #), ein Register oder ein geschobenes Register sein. Anders als bei den meisten anderen Prozessortypen, werden bei ARM bei logischen Operationen die Statusbits standardmäßig nicht dem Resultat entsprechend gesetzt. Soll dies erfolgen, ist die Instruktion mit dem Suffix „S“ zu notieren.

Beispiel:

Die BIC-Instruktion wird zum Nullsetzen einzelner Bits in einem Datenwort benutzt, ohne die anderen Bits zu beeinflussen. Im folgenden Code werden alle Bits in r2, die eine „1“ verkörpern, in r1 auf Null gesetzt. Das Resultat wird in r0 gespeichert.

```
Vorher:
r1=0b1111
r2=0b0101

BIC r0,r1,r2

Nachher:
r0 =0b1010
```

Listing 7.11: Beispiel Instruktion BIC

Beispiel:

Die ORR-Instruktion führt in jeder Bitstelle eine Oder-Verknüpfung aus. Hier werden r1 und r2 miteinander ODER-verknüpft und das Resultat in r0 abgelegt.

```
Vorher:
r0=0x00000000
r1=0x10305070
r2=0x02040608

ORR r0,r1,r2

Nachher:
r0 =0x12345678
```

Listing 7.12: Beispiel Instruktion ORR

7.5.3. Shift und Rotate

Um Werte zu schieben oder zu rotieren gibt es zwei Varianten:

1. Verwendung der MOV-Instruktion mit Barrel-Shifter
2. Verwendung der Instruktionen ASR, LSL, LSR und ROR

Beispiel:

Der Wert im Register r0 soll um eine Bitstelle nach links geschoben werden. Dies entspricht einer Multiplikation mit 2. Am einfachsten wird dies mit einer MOV-Instruktion erreicht, die auf sich selbst wirkt.

```
Vorher:
r0=0x00004321

MOVS r0, r0, LSL #1    @ r0 = r0 << 1, using Barrel Shifter
LSL r0, r0, #1          @ r0 = r0 << 1, using LSL Instruction

Nachher:
r0 =0x00008642
```

7.6. Programmverzweigungen

7.6.1. Generelles

Verzweigungsbefehle werden nach folgenden Kriterien unterteilt:

1. unbedingte oder bedingte Verzweigung
2. absolute oder relative Verzweigung

Eine bedingte Verzweigung wird nur ausgeführt, wenn gewisse Bedingungen (Statusbits) erfüllt sind. Eine unbedingte Verzweigung wird hingegen immer ausgeführt.

Bei einer absoluten Verzweigung springt das Programm auf eine fixe, vorgegebene Adresse. Diese Form der Verzweigung kennen ARM-Prozessoren nicht. Sie ist aber bei vielen anderen Prozessorfamilien üblich. Bei einer relativen Verzweigung springt das Programm von der aktuellen Adresse um einen Offset weiter. Der Offset wird vom Assembler berechnet. Alle Verzweigungen erfolgen beim ARM offsetbasiert, d.h. relativ.

absolute Verzweigung

(Z.B. Intel x86, Motorola 68x, jedoch nicht ARM!)

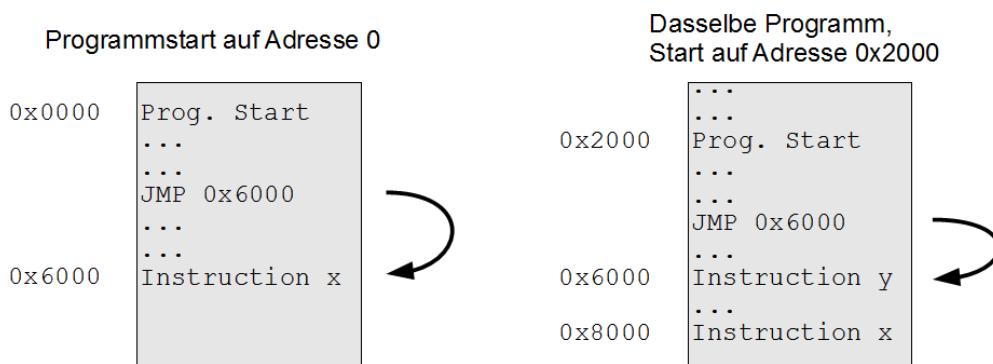


Abbildung 7.5.: Absolute Programmverzweigung

Das Programm links mit Startadresse 0x0 verzweigt nach der Jump-Instruktion (JMP) auf Adresse 0x6000, was korrekt ist. Wird nun dasselbe Programm im Speicher auf Startadresse 0x2000 abgelegt (rechtes Bild), so verzweigt die Instruktion JMP 0x6000 nach wie vor auf die Adresse 0x6000, was nicht die ursprüngliche Idee der Programmierer*innen war.

→ Programme mit absoluten Verzweigungen können nicht beliebig im Speicher verschoben werden.

relative Verzweigung

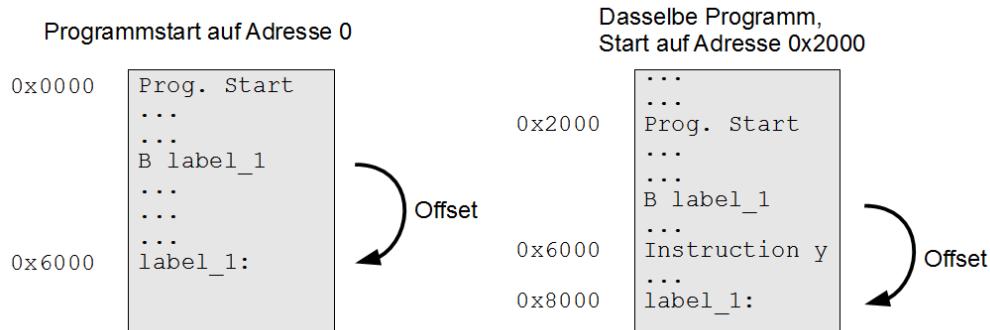


Abbildung 7.6.: relative Programmverzweigung

Das Programm mit Startadresse 0x0 (linkes Bild) verzweigt nach der B-Instruktion (Branch) um einen Offset zum label_1. Wird dieses Programm im Speicher verschoben (rechtes Bild), so läuft es immer noch korrekt, weil die Sprungdistanz zwischen B-Instruktion und label_1 dieselbe bleibt.

→ Programme mit relativen Verzweigungen können beliebig im Speicher verschoben werden.

7.6.2. Unbedingte Programmverzweigungen

Unbedingte Programmverzweigungen sind Sprungbefehle. Diese Instruktionen werden verwendet um den sequenziellen Programmlauf zu ändern. Einige dieser Sprungbefehle werden für den Aufruf eines Unterprogrammes verwendet.

Durch das Konzept der bedingten Instruktionsausführung stellen diese Verzweigungsinstruktionen nur einen Spezialfall der bedingten Instruktionsausführung dar.

Die Syntax der unbedingten Verzweigungsinstruktionen lautet:

```
<B | BL> <label>
BX Rm
BLX <label> | Rm
```

Listing 7.13: Verzweigungsinstruktionen

Das Label verkörpert als symbolische Konstante die Sprungadresse zu der verzweigt werden soll. Sie wird vom Assembler als pc-relativer Versatz (Offset) von der aktuellen Speicherstelle berechnet und muss innerhalb ± 16 MB liegen. Während bei der B-/BL-Instruktion das Sprungziel als (symbolische) Konstante vorliegen muss, kann mit den BX-/BLX- Verzweigungen auch registerbasiert eine Verzweigung erfolgen. Dies ist besonders dann nützlich, wenn ein Sprungziel während des Programmlaufes berechnet wird oder aus einer Tabelle gelesen wird.

Die vier Verzweigungsinstruktionen sind in Tabelle 7.16 aufgeführt.

Instruktion	Sprungdistanz	Beschreibung	Wirkung
B	$\pm 16 \text{ MB}$	Generelle Programmverzweigung	$\text{pc} = \text{Offset(label)}$
BL	$\pm 16 \text{ MB}$	Unterprogrammaufruf. Programmverzweigung mit Speichern der Rücksprungadresse im Link-Register	$\text{pc} = \text{Offset(label)}$; $\text{lr} = \text{Adresse der nächsten Instruktion nach BL}$
BX	Full	Analog B, aber mit UsageFault Exception wenn Bit 0 von Rm 0 ist.	$\text{pc} = \text{Rm} \& 0xffffffff; \text{T} = \text{Rm} \& 1$
BLX	Full	Analog BL, aber mit UsageFault Exception wenn Bit 0 von Rm 0 ist.	$\text{pc} = \text{Rm} \& 0xffffffff; \text{T} = \text{Rm} \& 1; \text{lr} = \text{Adresse der nächsten Instruktion nach BL}$

Tabelle 7.16.: Programmverzweigungen

Der Aufruf einer Subroutine erfolgt üblicherweise mit einer BL-Instruktion. Bei dieser Sprungvariante wird im Link-Register (lr, r14) die Adresse der dem Aufruf nachfolgenden Instruktion gespeichert. Sie dient als Rücksprungadresse aus der Subroutine (siehe auch Kapitel 9.2).

Beispiel:

Sprünge können grundsätzlich vorwärts und rückwärts im Programmcode erfolgen. Die hierzu notwendigen Offsetberechnungen werden automatisch vom Assembler durchgeführt. Der folgende Code zeigt wie unbedingte Sprünge codiert werden können. Der erste Sprung überspringt die ersten drei Additionsbefehle und führt direkt zur ersten Subtraktion. Der zweite Sprung im Code springt rückwärts und produziert eine nicht abbrechende Schleife.

```

B    vorwaerts
ADD r1,r2,#4
ADD r0,r6,#2
ADD r3,r7,#4
vorwaerts:
SUB r1,r2,#4
rueckwaerts:
ADD r1,r2,#4
SUB r1,r2,#4
ADD r4,r6,r7
B    rueckwaerts

```

Listing 7.14: Programmverzweigungen

Beispiel:

Der Aufruf einer Subroutine erfolgt mit Hilfe der BL-Instruktion. Der Rücksprung erfolgt mit der durch BL im Link-Register gespeicherten Rücksprungadresse. Der folgende Code zeigt das Prinzip des Subroutinenaufrufes mit Rücksprung.

```

. . .
BL subroutine
ADD r0,r1,r2
. . .
subroutine:
<Subroutinen Code >
MOV pc,lr          @ Ruecksprung

```

Listing 7.15: Beispiel Instruktion BL

Cortex-Mx Prozessoren kennen keinen speziellen Subroutinenrücksprungbefehl. Dies erfolgt immer mit MOV pc,lr oder mit dem Lesen der Rücksprungadresse vom Stack (POP), siehe Kapitel 9.2 und Kapitel 7.4.5.

7.6.3. Bedingte Programmverzweigungen

Die Syntax für die bedingten Sprungbefehle lautet:

```
<B | BL>{<cond>} label
BX{<cond>} Rm
BLX{<cond>} Rm
```

Listing 7.16: Sprungbefehle

Hierbei ist <cond> eine Bedingung als Suffix nach Tabelle 7.17. Der Rest ist analog Kapitel 7.6.2.

Bedingung	Beschreibung	Flags
EQ	Equal	Z
NE	Not Equal	!Z
CS, HS	Unsigned higher or same	C
CC, LO	Unsigned lower	!C
MI	Minus, Negative	N
PL	Positive or zero	!N
VS	Overflow	V
VC	No overflow	!V
HI	Unsigned higher	C && !Z
LS	Unsigned lower or same	!C && Z
GE	Signed greater or equal	(N == V)
LT	Signed less than	(N != V)
GT	Signed greater than	!Z und (N == V)
LE	Less or equal	Z oder (N != V)

Tabelle 7.17.: Bedingungen (Condition Codes) für bedingte Programmverzweigungen

Beispiel:

Berechnen der Fakultät einer positiven ganzen Zahl durch fortgesetzte Multiplikation mit Hilfe einer Subroutine. Die Subroutine erhält die Zahl als Argument in r0 und führt die Kettenmultiplikation aus. Das Resultat wird in r0 an die aufrufende Einheit zurückgegeben.

```
main:
    MOV r0,#5          @ Beispiel 5!
    BL Fakultaet       @ (r0!) berechnen
    B main             @ Endlosschleife

# Subroutine zum Berechnen der Fakultät durch fortgesetzte Multiplikation
# n!=1*(n-1)*(n-2)*...*3*2
# Input r0: Argument n
# Output r0: Resultat n!
# Benutzte Register: r1 als Zaehler
Fakultaet:
    MOVS r1,r0          @ r1 ist Zaehlerregister, initialisiert mit Startwert
    BGE l1              @ Startwert n >= 0? dann Sprung zu l1
    MOV r0,#0            @ nein, n<0, Resultat ist 0 (Konvention)
    B l2
l1: MOV r0,#1          @ Resultat von 0! oder Startwert fuer loop
l2: BLE endFakultaet   @ Abbruch falls n<=0

loop:
    MUL r0,r1,r0        @ Fortgesetzte Multiplikation n*(n-1)*(n-2)*...*3*2
    SUB r1,r1,#1         @ Zaehlerregister dekrementieren
```

```
CMP    r1,#1          @ Zaehlerregister > 1: naechster loop
BGT    loop

endFakultaet:
MOV    pc,lr          @ Ruecksprung
```

Listing 7.17: Beispiel Berechnung der Fakultät

8. Assembler Direktiven

8.1. Einleitung

Assembler-Direktiven sind Pseudoinstruktionen zur Steuerung des Assemblierorganges. Im Gegensatz zu „echten“ Instruktionen wird aus den Assembler-Direktiven kein Maschinencode erzeugt. Die Assembler-Direktiven sind vom verwendeten Assembler abhängig. In der Regel werden folgende Funktionen zur Verfügung gestellt:

- Steuerung des Assemblierorgangs
- Steuerung des Linkvorganges
- Definition von Variablen und Konstanten

8.2. Kurzübersicht

Tabelle 8.1 gibt eine kurze Übersicht über die wichtigsten Assembler-Direktiven des GNU-Assemblers.

Namensfeld	Anweisung	Operandenfeld	Beschreibung	
	.arm	Verwendung des ARM Befehlssatzes (32-Bit)		
	.thumb	Verwendung des Thumb Befehlssatzes (16-Bit)		
	.syntax	Spezifikation der verwendeten Syntax (16-Bit)		
	.global	Symbolname	Export eines Symbols	
	.extern	Symbolname	Import eines Symbols (wird von GNU ignoriert)	
	.align .balign .p2align	Ausrichtung, Füllwert	Ausrichten der Speicherplatzadresse	
Symbolname	.ascii, .asciz	String	Definiert Stringkonstanten	
Symbolname	.byte .2byte, .hword .4byte, .word	Wert	Definiert einen Datenwert	
Symbolname	.space	Anzahl, Wert	Reserviert Speicherplatz mit Initialisierung für ganzen Block	
	.include	Pfadfilename	Fügt Dateiinhalte ein	
	.equ, .set, =	Symbolname, Wert	Wertzuweisung für ein Symbol	
	.org	Adresse	Zuweisung einer absoluten Adresse	
	.section	Symbolname, Flags	Speicherbereich definieren oder wechseln.	
	.text, .data, .bss		Standard-Sections	
	.end		Quellcode-Ende	
	.if .ifdef .ifndef .endif	Bedingung	Bedingte Assemblierung	
	.macro	Symbolname, Parameter	Makrodefinition	
	.rep, .irp	Anzahl Parameter	Wiederholung von Codeblöcken	
	.endm .endr .endif		Ende des Makros oder der Kontrollstruktur	

Tabelle 8.1.: Kurzübersicht Assembler-Direktiven

8.3. .arm, .thumb, .syntax

Die Assembleranweisung .arm legt fest, dass der nachfolgende Code aus dem 32-Bit ARM-Befehlssatz stammt und verarbeitet wird. Der 16-Bit Thumb-Code wird mit der Anweisung .thumb eingeleitet. Im Thumb-Modus gilt eine leicht andere Syntax und es stehen nur die Instruktionen und Register aus dem Thumb-Befehlssatz zur Verfügung. Die beiden Anweisungen sind synonym zu .code 32 und .code 16.

Die Direktive „.syntax unified“ definiert die Verwendung der UAL (Unified Assembler Language). Die UAL ist ein Superset für den ARM- und den Thumb-Instruktionssatz und löst die ursprünglichen Versionen ab.

Beispiel:

```
.arm
. . .      @ 32-Bit ARM-Instruktionen

.thumb
. . .      @ 16-Bit Thumb-Instruktionen

.syntax unified
. . .      @ Use Unified Assembler Language
```

Listing 8.1: .arm und .thumb Assembler Direktiven

8.4. .global, .extern

Mit .global wird ein lokal definiertes Symbol wie beispielsweise eine Subroutine oder eine Variable öffentlich (exportiert) und dadurch in anderen Modulen (Dateien) verwendbar. Die .extern-Anweisung wird aus Kompatibilitätsgründen zu anderen Assemblern vom GNU-Assembler akzeptiert. Sie hat aber keine Wirkung, weil der GNU-Assembler alle in einem Modul undefinierten Symbole automatisch als extern definiert betrachtet. Für die Dokumentation der Modulschnittstelle kann mit .extern aber gezeigt werden, welche Symbole aus anderen Modulen referenziert werden.

main.s <pre>.extern InitDisplay BL InitDisplay</pre>	lcd.s <pre>.global InitDisplay InitDisplay: ... MOV pc,lr</pre>
--	---

Abbildung 8.1.: Beispiel .global und .extern Assembler Direktiven

In Abbildung 8.1 wird im File main.s die Subroutine InitDisplay aufgerufen (mehr zu Subroutinen erfahren Sie in Kapitel 9). Im File lcd.s wird die Subroutine InitDisplay definiert und mit .global exportiert. Wird .global vergessen, so wird ein Linker-Fehler ausgegeben. Aus Dokumentationsgründen kann in main.s mit .extern gezeigt werden, dass das Symbol InitDisplay in einem anderen Modul definiert ist. Für die Symbolauflösung durch den Linker ist .extern nicht notwendig.

Anders als bei einer .extern-Definition in der Hochsprache C sind auch keine Informationen über eventuelle Rückgabewerte oder Parameter möglich, wie dies für eine Funktion beispielsweise mit einer Deklaration extern void func(int) der Fall ist. Der korrekte Umgang mit Argumenten und Funktionswert liegt in der alleinigen Verantwortung der Assemblerprogrammierer*innen.

Beispiel:

Ein kleines Assemblerprogramm, welches nach dem Start bei Label main eine nicht abbrechende Schleife ausführt, liegt im Text-Segment (siehe Kapitel 8.14). Im Startup-Code muss zuvor sichergestellt sein, dass nach dem Initialisieren der Hardware auch zum main-Label verzweigt wird. Im folgenden Listing ist zu erkennen, dass das Label main durch die .global Assembler-Direktive exportiert wird, sodass sie im Modul des Startup-Codes bekannt ist.

```
.arm
.global main          @ export symbol main

.text                 @ text segment, including code to execute
main:                @ label main
    B main            @ endless loop, branch to main label
```

Listing 8.2: .global Assembler Direktive am Beispiel des Labels main

8.5. .align

Mit Hilfe der Anweisung .align wird auf die nächste 32-Bit Adresse ausgerichtet. Hierzu werden 0 bis drei Null-Bytes eingefügt. Der ARM-Prozessor verlangt, dass der Programmcode und die Word-Daten immer bezüglich einer 32-Bit Wortgrenze im Speicher ausgerichtet abgelegt werden.

Beispiel:

0000 AA55	b1: .byte 0xAA,0x55	
0002 0000	.align	@ inserts two bytes with value 0
0004 48616C6C	str1: .asciz "Hello"	
6F00		

Listing 8.3: .align Assembler Direktive

Im obigen Listing ist zu erkennen:

- Ab der virtuellen Adresse 0000 (erste Spalte) werden zwei Bytes mit den Werten 0xAA und 0x55 gespeichert (.byte siehe Kapitel 8.7).
- Die Assembler-Direktive .align füllt die nachfolgenden Bytes mit dem Wert 00, bis die nächste Word-Adresse beginnt. In diesem Beispiel sind dies zwei Bytes.
- Ab der virtuellen Adresse 0004 wird der String „Hello“ gespeichert (.asciz siehe Kapitel 8.6). Der String wird mit einem '\0' abgeschlossen, dieses Terminierungs-Byte liegt auf Adresse 0009.

Für die Anweisung gilt die Syntax:

```
.align AL {, FB}
```

Die geforderte Ausrichtung kann mit „AL“ angegeben werden. Sie ist aber plattformabhängig. So bewirkt .align 3 bei ARM oder Intel i386 eine Adressenausrichtung auf 8 oder 0 (2^3 Modulo 0). Zusätzlich kann ein beliebiges Füllbyte „FB“ anstatt 0x00 verwendet werden.

Beispiel:

Der folgende Code zeigt die Wirkung der .align-Anweisung zum Ausrichten auf die nächste Adresse. Hier wird mit .align 3 auf die nächste $2^3 = 8$ Byte-Adresse ausgerichtet, zudem wird ein Füllbyte mit dem Wert 0xFF definiert:

0000 AA	b1: .byte 0xaa	
0001 FFFFFFFF	.align 3,0xFF	@ Modulo 2^3 == 0 aligned, filled with 0xFF
FF		
0008 48616C6C	str1: .asciz "Hello"	
6F00		

Listing 8.4: .align 3 Assembler Direktive

Da die Wirkung von .align prozessorspezifisch wirkt, wurden beim GNU-GCC-Assembler zwei prozessorunabhängige Ausrichtungen .balign und .p2align definiert.

Die Anweisung .balign (Byte Align) arbeitet wie .align, nur dass der Ausrichtungsparameter AL immer als Byte zu verstehen ist. Er beschreibt den Modulowert, auf den die Adresse ausgerichtet wird. Ein .balign 8 wird also plattformunabhängig immer auf eine Adresse ausrichten, die ein Mehrfaches von 8 ist.

Die Anweisung .p2align (Power of 2 Align) arbeitet wie .align, nur dass der Ausrichtungsparameter AL immer als Exponent zur Basis 2 zu verstehen ist. Die Potenz beschreibt den Modulowert, auf den die Adresse ausgerichtet wird. Ein .p2align 3 wird also plattformunabhängig immer eine Ausrichtung auf eine Adresse, die ganzzahlig durch 8 teilbar ist, bewirken.

8.6. .ascii, .asciz

Stringliterale können mit .ascii oder .asciz in den Code eingebracht werden. Mehrere Strings werden mit Kommata getrennt in der gleichen Anweisung aufgeführt. Die ASCII-Werte der einzelnen Zeichen werden direkt aufeinander folgend im Speicher abgelegt. Mit .asciz wird jeder String automatisch mit einem Nullbyte terminiert, wie bei C-Strings üblich.

Syntax:

```
{label:}<.ascii|.asciz> <string> {,<string>} . . .
```

Beispiel:

```
0000 48616C6C      str1: .ascii "Hello"
                   6F
0005 57656C74      str2: .asciz "World!"  @ Terminierung mit Nullbyte
                   2100
```

Listing 8.5: .ascii und .asciz Assembler Direktive

8.7. .byte

Die Anweisung .byte ermöglicht das Einbringen von initialisierten Byte-Konstanten und Variablen. Mehrere aufeinanderfolgende Werte können mit Kommata separiert aufgeführt werden. Sie werden in der Aufzählungsreihenfolge im Speicher aufsteigend abgelegt.

Syntax:

```
{label:}.byte <byte1> {,<byte2>} . . .
```

Das Label verkörpert einen Namen, der als symbolische Adresse für den Zugriff benutzt werden kann. Im Gegensatz zu vielen anderen Assemblern werden beim GNU-Assembler Zahlen mit einer führenden Null als Oktalzahlen interpretiert, wie in C. Negative Zahlen werden in Zweierkomplementform abgelegt. Hexadezimalzahlen beginnen mit 0x oder 0X, Binärzahlen mit 0b oder 0B. Zeichenkonstanten werden in einfachen Hochkomma formuliert. Zahlen ohne führende Null (0b, 0B, 0x, 0X) werden als Dezimalzahl aufgefasst.

Beispiel:

```
.data
0000 416225  b1: .byte 'A','b',37      @ Zeichenkonstanten
0003 20D0      b2: .byte 0x20, -0x30    @ Hexadezimal
0005 F6        b3: .byte -10            @ Dezimal (Zweierkomplement)
0006 08        b3: .byte 010             @ Oktal
0007 0A        b5: .byte 0B1010          @ Binaer
```

Listing 8.6: .byte Assembler Direktive

8.8. .hword, .2byte

Die Anweisungen .hword und .2byte ermöglichen das Einbringen von initialisierten 16-Bit Konstanten und Variablen. Die Anweisungen sind auf der ARM-Plattform synonym. Mehrere aufeinanderfolgende Werte können mit Kommata separiert aufgeführt werden. Sie werden in der Aufzählungsreihenfolge im Speicher aufsteigend abgelegt. Bezüglich Label und Zahlenwerten gelten dieselben Bedingungen wie schon bei .byte.

Syntax:

```
{label:}.2byte | .hword <short1> {,<short2>} . . .
```

Beispiel:

```
0000 1100      .data
0002 33224400  h1:  .2byte 0x11
0002 33224400  h2:  .2byte 0x2233,0x44
0006 1100      h3:  .hword 0x11
0008 33224400  h4:  .hword 0x2233,0x44...
```

Listing 8.7: .hword und .2byte Assembler Direktive

Bemerkung: ARM-CPUs verwenden das „little-endian Format“ (siehe Kapitel 5.6). D.h. das tieferwertigste Byte wird auf der tiefsten Adresse gespeichert. Im Beispiel oben wird deshalb bei Symbol h2 vom Wert 0x2233 das tieferwertige Byte 0x33 auf die tiefere Adresse (0002) gespeichert.

8.9. .word, .4byte

Die Anweisungen .word und .4byte ermöglichen das Einbringen von initialisierten 32-Bit Konstanten und Variablen. Die Anweisungen sind auf der ARM-Plattform synonym. Mehrere aufeinanderfolgende Werte können mit Kommata separiert aufgeführt werden. Sie werden in der Aufzählungsreihenfolge im Speicher aufsteigend abgelegt. Bezüglich Label und Zahlenwerten gelten dieselben Bedingungen wie schon bei .byte.

Syntax:

```
{label:}.4byte | .word <word1> {,<word2>} . . .
```

Beispiel:

```
0000 11000000  .data
0004 55443322  w1:  .4byte 0x11
0004 55443322  w2:  .4byte 0x22334455,0x66
000c 11000000  w3:  .word 0x11
0010 55443322  w4:  .word 0x22334455,0x66
0010 55443322  66000000
```

Listing 8.8: .word und .4byte Assembler Direktive

Bemerkung: Verwenden Sie .byte, .hword und .word nur für initialisierte Konstanten und Variablen. Für uninitialisierte Konstanten und Variablen ist .space zu verwenden.

Achtung: Der Code im folgenden Listing legt keinen Speicher an!

```
.data
myvar: .word
```

Listing 8.9: fehlerhafte Definition mit .word

8.10. .space

Die .space-Anweisung reserviert Speicherplatz in Form eines Byte-Blocks. Dieser wird normalerweise zur Speicherung von Variablen (z.B. Arrays) benutzt.

Syntax:

```
{label:} .space <AB> {,<FB>}
```

Der Parameter AB spezifiziert die Anzahl Bytes, die für den Speicherblock reserviert werden. Ein leerer Parameter ist zulässig, führt aber zu keinem Speicherplatz. Der optionale Parameter FB definiert ein Füllbyte, das für den gesamten Block benutzt wird.

Beispiel:

Datenblöcke sind linear aufeinanderfolgende Bytes im Speicher. Sie können auch mit Initialisierung definiert werden. Es ist aber Aufgabe des Startup-Codes, dafür zu sorgen, dass in einer Embedded-Anwendung dieser Bereich tatsächlich beim Start mit den Initialwerten beschrieben wird. Diesem Umstand ist besonders bei selbst definierten Segmenten Rechnung zu tragen.

```
0000 00000000      .bss
00000000
00000000
00000000
00000000
00000000      .data
0010 AAAAAAAA      label_1: .space 0x10
AAAAAAA
AAAAAAA
AAAAAAA
AAAAAAA
AAAAAAA
.....
10010 FFFFFFFF      label_2: .space 0x10000,0xAA
FFFFFFF
FFFFFFF
FFFFFFF
FFFFFFF
FFFFFFF
FFFFFFF
```

Listing 8.10: .space Assembler Direktive

Beachten Sie, dass ohne Angaben eines Füllbytes der Speicherbereich nicht initialisiert ist (.bss), dass mit Angabe eines Füllbytes der Speicherbereich aber initialisiert ist (.data). Zu .bss und .data siehe auch Kapitel 8.14.

8.11. .include

Die .include-Direktive kopiert zur Übersetzungszeit den Inhalt der angegebenen Datei in den aktuellen Quellcode. Für alle Include-Dateien gilt für den Bezugspfad das aktuelle Debug-Directory. Im Regelfall ist dies nicht das Verzeichnis, wo sich der Quellcode befindet. Mittels Kommandozeilen-Option -I kann bei GCC ein Pfad für Include-Dateien spezifiziert werden.

Syntax:

```
.include "PfadFilename"
```

Beispiel:

```
.include      ".../src/myincludes.s"
```

Listing 8.11: .include Assembler Direktive

8.12. .equ, .set, =

Die .equ-Direktive weist wie .set einem Symbol einen Wert zu. Die Anweisungen sind synonym und die Zuweisung mit „=“ ist ebenfalls funktionell identisch. Alle so definierten Symbole können auch innerhalb des Moduls umdefiniert werden.

Beispiel:

```

1          .text
2          .equ newline, 0xa           @ defined symbols
3          .set bitmask, 0b10010111
4          TAB = 011
5
6 0000 0A00AOE3      MOV r0,#newline        @ using the symbols
7 0004 9710AOE3      MOV r1,#bitmask
8 0008 011000E0      AND r1,r0,r1
9 000c 0930AOE3      MOV r3,#TAB
10
11         .equ newline,100        @ symbol newline redefined
12
13 0010 6400AOE3      MOV r0,#newline

```

Listing 8.12: .equ, .set und „=“ Assembler Direktiven

Die Symbole „newline“, „bitmask“ und „TAB“ werden mit numerischen Äquivalenten versehen. Im nachfolgenden Code kann mit den Symbolen anstelle der Zahlenwerte gearbeitet werden. Die zentrale Definition ist übersichtlicher und wartungsfreundlicher. Das Einbringen von direkten numerischen Werten in die Assemblerinstruktionen sollte den Ausnahmefall darstellen.

8.13. .org

Die Anweisung .org definiert eine absolute Adresse, auf welcher die nachfolgenden Anweisungen (Programm oder Daten) abgelegt werden. Da bei ARM ausschliesslich relative Sprünge verwendet werden (siehe Kapitel 7.6), sind Fixierungen mit .org äusserst selten.

Beispiel:

```

mysub:   .org    0xa0000800
...       MOV pc,lr

```

Listing 8.13: .org Assembler Direktive

Im obigen Beispiel wird dem Assembler mitgeteilt, dass die Startadresse der Subroutine „mysub“ auf der Adresse 0xa0000800 liegen soll. In der Regel ist es eleganter, Sektionen zu verwenden (siehe Kapitel 8.14).

8.14. .section

Der Adressbereich für Programm und Daten kann in mehrere Blöcke (Segmente, Sektionen) aufgeteilt werden. Man unterscheidet primär Codesegmente und Datensegmente. Codesegmente (.text) beinhalten ausführbaren Code und liegen normalerweise in einem ROM-Bereich. Datensegmente beinhalten Variablen, Stack und andere beschreibbare Größen im RAM-Bereich. Man unterscheidet weiter initialisierte Bereiche (.data) und nicht initialisierte Bereiche

(.bss). Initialisierte Bereiche (.data) beinhalten alle initialisierten Variablen. Beim Programmstart müssen die Initialwerte mit einer Laderoutine in den RAM-Bereich übertragen werden. Bei einem C-Programm wird dies durch den sogenannten Startup-Code erledigt (siehe Kapitel 10.1).

In einer Klasse werden mehrere Segmente zusammengefasst. Die wichtigsten Basisklassen sind: .text, .data und .bss. In diesen Klassen können Unterklassen, d. h. Sektionen gebildet werden. Weil die vordefinierten Klassen .text, .data, .bss zur Verfügung stehen, werden in Kleinprogrammen normalerweise keine selbstdefinierten Sektionen erstellt. Bei komplexen Programmen oder wenn mit fremden Objektmodulen gearbeitet wird, können aber Segmentdefinitionen notwendig werden. Weitere vordefinierte Segmente sind in Tabelle 8.2 aufgeführt.

Segmentname (Section)	BasisKlasse	Inhalt
.text	.text	Programm-Code
.data	.data	Initialisierte Variablen
.bss	.bss	Nicht initialisierte Variablen
.startup	.text	Startup-Code
.rodata, .rodata*	.text	Read-Only Daten (Strings, Konstanten)
.glue_7, .glue_7t	.text (ev. .data)	Arm-Thumb Interface Code

Tabelle 8.2.: Vordefinierte Segmentnamen

Mit der .section-Direktive kann ein neues Segment definiert oder zu einem bereits definierten Segment gewechselt werden. Alle Anweisungen (Programm oder Daten), die nach der .section-Direktive folgen, werden diesem Segment zugeordnet. Im Gegensatz zur .org-Direktive ist diese Zuweisung aber nicht absolut. Die Adressfestlegung erfolgt während dem Linkvorgang. In STM32CubelDE wird dies für den STM32H743 (siehe Kapitel 5) im Linkerskript „stm32h743zitx_flash.ld“ definiert. Dort werden die Startadressen festgelegt. Ebenso wird festgelegt, in welcher Reihenfolge die Sections im Speicher angeordnet werden und wie die Initialisierung stattfindet.

Syntax:

```
.section <SectionName> { , 'flags' }
```

Eine neue Sektion kann mit <SectionName> symbolisch frei definiert werden. Die Flags legen die Zugriffseinstellungen für den Speicher fest. Die möglichen Flags für ein ELF-Target zeigt Tabelle 8.3. Werden keine Flags definiert, gelten die Einstellungen der Basisklasse.

Flag	Beschreibung
a	Allozierbarer Bereich für dynamische Speicher-verwaltung
w	Beschreibbarer Bereich
x	Ausführbarer Code

Tabelle 8.3.: Zugriffs-Flags für die Segmentdefinition

Segmentwechsel auf Nicht-Basissegmente müssen immer mit .section eingeleitet werden. Bei Basisklassen kann .section weggelassen werden:

```
.text
.data
.bss
.section .subroutines , "x"
.section .text          @ Synonym for .text
.section .startup
.section MySection, "w"
```

Listing 8.14: .text, .data, .bss und .section Assembler Direktiven

Beispiel:

Sind eigene Sections notwendig, werden diese im Assemblermodul mit Namen definiert. Im folgenden Listing werden zwei Subroutinen (mysub1 und mysub2) in einer eigenen Section „mysubroutines“ definiert:

```
.text
loop:                                @ located in .text segment
    BL mysub1
    BL mysub2
    B loop

    .section .mysubroutines
mysub1:                                @ located in .mysubroutines segment
    MOV pc,lr

    .data
b1: .byte 0x22,0x33                  @ located in .data segment
msg: .asciz "hallo"

    .section .mysubroutines
mysub2:                                @ located in .mysubroutines segment
    MOV pc,lr
```

Listing 8.15: Beispiel mit .section Assembler Direktiven

Im Linkerskript „stm32h743zitx_flash.ld“ wird festgelegt, zu welcher Basisklasse (.text, .data, .bss) die Section „mysubroutines“ gehört. In diesem Beispiel ist es sinnvoll, es zur Basisklasse .text hinzuzufügen (die beiden Subroutinen entsprechen ausführbarem Code). Im Linkerskript erfolgt eine Ergänzung durch Anfügen der Section an die bereits definierten Teile. So kann auch festgelegt werden, in welcher Reihenfolge die Sections im Speicher abgelegt werden. Es ist zu beachten, dass Einträge im Linkerskript mit der notwendigen Sorgfalt und Sachkenntnis erfolgen müssen. Fehler führen häufig bereits beim Start zum Systemabsturz.

```
SECTIONS
{
    /* The startup code goes first into FLASH */
    .isr_vector :
    {
        . = ALIGN(4);
        KEEP(*(.isr_vector)) /* Startup code */
        . = ALIGN(4);
    } >FLASH

    /* The program code and other data goes into FLASH */
    .text :
    {
        . = ALIGN(4);
        *(.text)           /* .text sections (code) */
        *(.text*)          /* .text* sections (code) */
        *(.mysubroutines) /* --> user defined section .mysubroutines */
        *(.glue_7)          /* glue arm to thumb code */
        *(.glue_7t)         /* glue thumb to arm code */
        *(.eh_frame)

        KEEP (*(.*(.init)))
        KEEP (*(.*(.fini)))

        . = ALIGN(4);
        _etext = .;          /* define a global symbols at end of code */
    } >FLASH

    ...
}
```

Listing 8.16: Auszug aus dem Linkerskript mit einer zusätzlichen Sections „mysubroutines“.

Die absoluten Speicherplatzadressen können anschliessend im Map-File eingesehen werden. Das Map-File wird beim Linkvorgang erzeugt und wird normalerweise im Debug-Verzeichnis als Projektnname.map abgelegt.

```
*fill*          0x0000000008000352      0x2
.text._libc_init_array
```

```

          0x0000000008000354      0x48 c:/apps/st/....
          0x0000000008000354      __libc_init_array
*(.mysubroutines)
.mysubroutines
          0x000000000800039c      0x4 ./Src/Leguan_ASM.o
*(.glue_7)
.glue_7      0x00000000080003a0      0x0 linker stubs
...

```

Listing 8.17: Auszug aus dem Map-File

8.15. .end

Die .end-Direktive zeigt dem Assembler, dass das Quellprogramm abgeschlossen ist. Alle Anweisungen nach der .end-Direktive werden vom Assembler ignoriert. Im ARM-GCC Assembler ist die .end-Direktive fakultativ und wird normalerweise weggelassen.

Beispiel:

```

mysub: . .
      MOV pc,lr           @ last Instruction, return
      .end                @ instructions following this directive will not be executed

```

Listing 8.18: .end Assembler Direktive

8.16. Assembler Operatoren

Innerhalb von Assembleranweisungen stehen für Berechnungen zahlreiche Operatoren zur Verfügung. Sie dienen primär zur Berechnung von Offsetwerten und anderen speicherbasierten Größen, z. B. Adressen, Array-Größen, etc. Bedingung für die Berechnung ist, dass die Operanden zur Assemblierzeit einen definierten Wert aufweisen. Dies setzt im Regelfall voraus, dass die Operatoren aus Konstanten gebildet worden sind.

Der GCC-Assembler kennt zwei Vorzeichenoperatoren. Sie wirken analog den Operatoren in C.

Operator	Beschreibung
-	Negatives Vorzeichen (2er Komplement)
~	Bitweise Negation

Tabelle 8.4.: Assembler Vorzeichenoperatoren

Die gültigen arithmetisch-logischen und Vergleichsoperatoren sind in Tabelle 8.5 aufgeführt. Anhand der Beschreibungen zum GCC-Assembler kann man nicht davon ausgehen, dass die Operatorenpräzedenz so klar definiert ist wie in ANSI-C.

Operator	Priorität	Beschreibung
/	1	Division (ganzzahlig)
*	1	Multiplikation
%	1	Divisionsrest (Modulus)
<<	1	Schieben nach links
>>	1	Schieben nach rechts
	2	Bitweise ODER-Verknüpfung
^	2	Bitweise EXOR-Verknüpfung
&	2	Bitweise UND-Verknüpfung
!	2	Bitweise ODER-NICHT-Verknüpfung
+	3	Addition
-	3	Subtraktion
==	3	Vergleich auf Gleichheit
<>	3	Vergleich auf Ungleichheit
<	3	Relationalvergleich kleiner als
>	3	Relationalvergleich grösser als
<=	3	Relationsvergleich kleiner/gleich als
>=	3	Relationsvergleich grösser/gleich als
&&	4	Logisches UND
	5	Logisches ODER

Tabelle 8.5.: Assembler Operatoren

Das Resultat einer Vergleichsoperation ist immer ein numerischer Wert. Dabei verkörpert -1 den Wert wahr, 0 den Wert falsch. Alle Relationalvergleiche erlauben den Vergleich vorzeichenbehafteter Zahlen.

Logische Operationen liefern als Resultat 1 (wahr) und 0 (falsch).

Zu beachten ist, dass keine arithmetischen Berechnungen erfolgen können, wenn die Operanden in verschiedenen Sections liegen.

8.17. Assembler Kontrollstrukturen

Assembler Kontrollstrukturen dienen zur Steuerung des Assemblierorganges. So lassen sich aufgrund einer Bedingung verschiedene Codeversionen erzeugen. Für Testzwecke kann beispielsweise ein zusätzlicher Code eingebracht werden, der Testausgaben und Prüfwerte enthält. In der Schlussversion ist dieser Code nicht mehr notwendig und kann weggelassen werden.

8.17.1. .if

Mit einer .if-Anweisung kann das Assemblieren des nachfolgenden Codeblocks bis zur .endif- oder wenn vorhanden, .else-Anweisung mit einer Bedingung verknüpft werden. Nur wenn sie erfüllt ist, wird der Code assembliert.

Syntax:

```
.if <expression>
...
{.else}
...
.endif
```

Wenn der logische Ausdruck wahr ist, d. h. ein Wert $<> 0$ ist, wird der Block assembliert. Andernfalls wird wenn ein .else-Zweig vorhanden ist, der .else-Block bis zum zugehörigen .endif assembliert.

Beispiel:

```
.macro SHIFTLEFT a,b
    .if \b < 0
        MOV \a,\a,ASR #-\b
    .else
        MOV \a,\a,LSL #\b
    .endif
.endm
```

Listing 8.19: .if Assembler Direktive

Bemerkung: Die .macro Assembler-Direktive wird in Kapitel 8.18 beschrieben.

8.17.2. .ifdef

Mit einer .ifdef-Anweisung wird der nachfolgende Code nur assembliert, wenn das Symbol definiert ist. Das Symbol muss zuvor als Label mit .set, .equ oder = definiert worden sein.

Syntax:

```
.ifdef <symbol>
    ...
{.else}
    ...
.endif
```

Listing 8.20: .ifdef Assembler Direktive

Beispiel:

Für Testzwecke kann selektiv zusätzlicher Code eingebunden werden. Nachfolgend wird beim Programmablauf ein Breakpoint eingefügt, wenn das Symbol DEBUG zuvor definiert worden ist.

```
.set DEBUG ,1

start:
    LDR r0 ,=0x80
ifdef DEBUG
    BKPT
endif
    LDR r1 ,=0x80
```

Listing 8.21: .ifdef Assembler Direktive

8.17.3. .ifndef

Mit einer .ifndef-Anweisung wird der nachfolgende Code nur assembliert, wenn das Symbol undefiniert ist. Die Wirkung ist analog .ifdef.

Syntax:

```
.ifndef <symbol>
...
{.else}
...
.endif
```

Listing 8.22: .ifndef Assembler Direktive

8.18. .macro

Die Makroanweisung .macro erzeugt für ein Symbol eine bestimmte Menge Text. Die Texterzeugung kann auch parametrisiert erfolgen. Die Wirkung ist immer auf dem Niveau der rein textuellen Substitution zu sehen. Jede Makrodefinition muss mit .endm abgeschlossen werden.

Syntax:

```
.macro <name> {<arg1>} {<arg2>} {<arg3>} ... {<argN>}
  {codeblock}
.endm
```

Der Name orientiert sich an den Regeln der Namen für Assemblersymbole. Konventionsgemäß werden Makronamen in Grossschrift formuliert. Die Argumente verkörpern formale Parameter, ähnlich einer Funktion in C. Für den Zugriff innerhalb des Makros ist ein Backslash '\' notwendig. Eine Makroanweisung kann vorzeitig mit .exitm beendet werden. Dies widerspricht zwar den Regeln der strukturierten Programmierung, kann aber fallweise den Makrocode vereinfachen.

Beispiel:

Analog zum Beispiel in Kapitel 8.17.1 kann unter Verwendung von .exitm ein Makro ohne .else-Zweig definiert werden.

```
Vorher:
r0 = 0x80
r1 = 0x80

.macro SHIFTLEFT a,b
.if \b < 0
    MOV \a,\a,ASR #-\b
    .exitm
.endif
    MOV \a,\a,LSL #\b
.endm

. .
SHIFTLEFT r0,2
SHIFTLEFT r1,-2

Nachher:
r0 = 0x200
r1 = 0x20
```

Listing 8.23: .macro Assembler Direktive

8.19. Assembler Wiederholungsstrukturen

Wiederholungsanweisungen zählen im weiteren Sinne auch zu den Makros, da sie ebenfalls eine bestimmte Menge Text erzeugen. Mit Wiederholungsanweisungen kann ein Codeblock eine bestimmte Anzahl Male wiederholt werden. Ein Block kann aus Instruktionen oder Zeichenfolgen bestehen. Der GNU-Assembler stellt hierzu zwei Anweisungen zur Verfügung:

- .rept (Repeat)
- .irp (Indefinite Repeat)

8.19.1. .rept

Die .rept-Anweisung wird benutzt, um einen Codeblock eine bestimmte Anzahl Male zu wiederholen.

Syntax:

```
.rept <number>
    {codeblock}
.endr
```

Beispiel:

Das Datenwortmuster 0x55AA55AA, 0x00000000, 0xAA55AA55, 0x11111111 soll als Folge von Datenworten vier mal im Speicher abgelegt werden. Mit einer .rept-Anweisung kann dies sehr kompakt formuliert werden.

```
tab1:
.rept 4
    .word 0x55AA55AA
    .word 0x00000000
    .word 0xAA55AA55
    .word 0x11111111
.endr
```

Listing 8.24: .rept Assembler Direktive

Beispiel:

Mit .rept kann unter Verwendung eines Symbols auch eine Tabelle mit berechneten Werten aufgebaut werden. Nachfolgende Sequenz zeigt den Aufbau einer Byte-Tabelle mit den Werten von 0..255:

```
.set L1,0          @ L1 ist lokales Symbol mit Startwert
tab2: .rept 256
      .byte L1
      .set L1,L1+1
.endr
```

Listing 8.25: .rept Assembler Direktive mit Symbolen

Mit dem Debugger können die beiden Tabellen tab1 und tab2 aus den obigen beiden Beispielen im Memory-Window betrachtet werden:

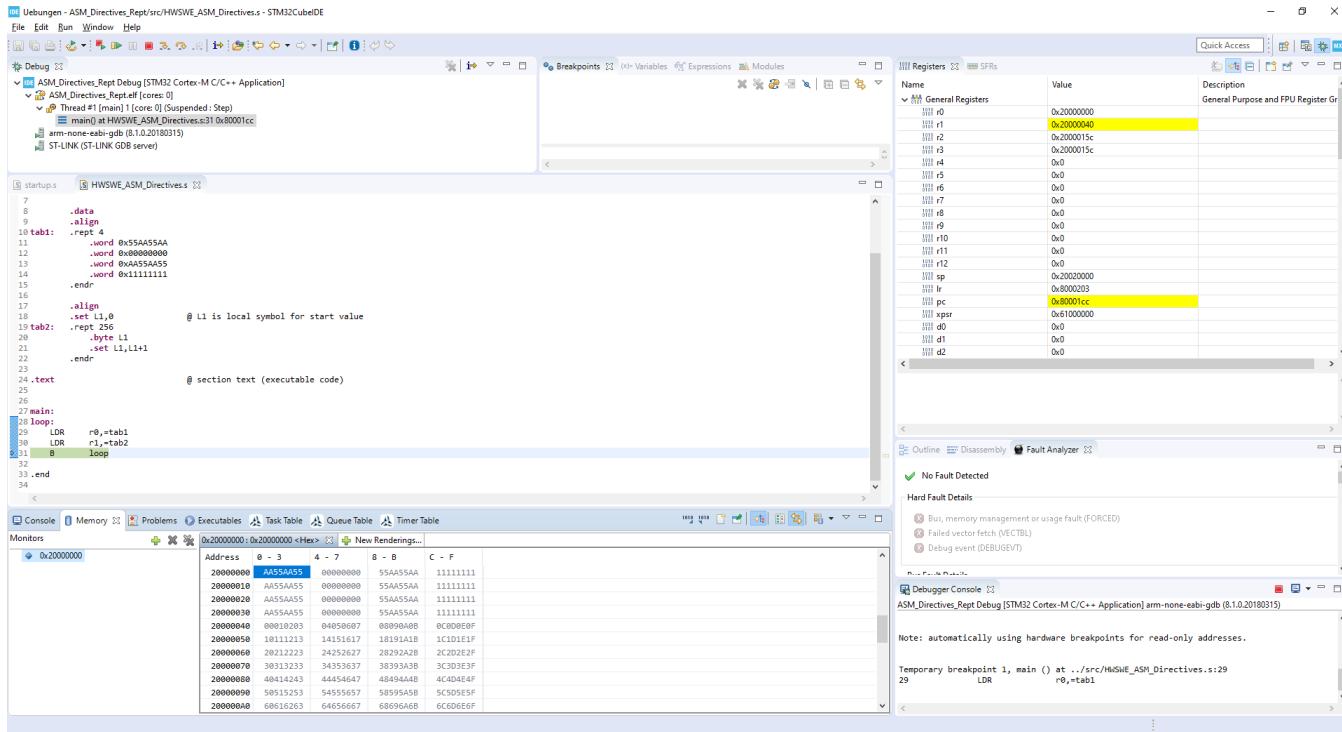


Abbildung 8.2.: Mit .rept wiederholte Struktur im Speicher

Bemerkungen: Die Tabelle tab1 ($4 * 4$ Word) startet auf Adresse 0x20000000 (in Abbildung 8.2 im Memory-Fenster unten links, Adresse ist blau hinterlegt). Tabelle tab2 startet auf Adresse 0x20000040, wobei nur die ersten Bytes in der Abbildung zu sehen sind. Beachten Sie, dass im Memory-Fenster jeweils Words im „little-endian Format“ dargestellt werden, d.h. dass von der Byte-Tabelle jeweils 4 Werte zusammengefasst sind.

8.19.2. .irp

Mit .irp kann eine Befehlsfolge auf mehrere Operanden angewandt werden (parametrisierte Wiederholung). So gesehen ist dies eine vereinfachte Form einer Makrodefinition für eine variable Anzahl Parameter. Der erste Parameter ist ein Symbol für den Operanden. Danach folgt eine beliebige Menge von Parametern, die nacheinander in den Codeblock eingesetzt werden.

Syntax:

```
.irp <symbol> { ,<value>} { ,<value>} ...
  {codeblock}
.endr
```

Die Grösse <value> muss ein ganzzahliger Wert sein.

Beispiel:

Mit .irp kann ein Codeblock für eine Liste von Parametern wiederholt werden. Der folgende Code addiert alle Parameter zum Register r0 mit Hilfe von r1:

```
.irp param,1,2,3,4
  MOV r1,#\param
  ADD r0,r0,r1
.endr
```

Listing 8.26: .irp Assembler Direktive mit Symbolen

Beim Assemblieren wird das Makro expandiert:

```
MOV r1, #1 ; 0x1
ADD r0, r0, r1
MOV r1, #2 ; 0x2
ADD r0, r0, r1
MOV r1, #3 ; 0x3
ADD r0, r0, r1
MOV r1, #4 ; 0x4
ADD r0, r0, r1
```

Listing 8.27: Expandiertes .irp-Makro aus obigem Beispiel

9. Subroutinen

9.1. Einführung

Unterprogramme werden in der Assembler-Programmierung „Subroutinen“ genannt. Sie entsprechen den Funktionen, Prozeduren oder Methoden in Hochsprachen.

Eine Subroutine hat folgende Eigenschaften:

- Eine Subroutine ist eine Folge von Instruktionen mit definierter Funktionalität und Schnittstelle (Parameter).
- Für das aufrufende Programm ist die Subroutine eine Blackbox. Das aufrufende Programm muss sich lediglich an die Schnittstellendefinition halten (Prinzip des Information Hiding).
- Subroutinen können mehrmals aufgerufen werden.

Subroutinen besitzen folgende Vorteile:

- Modularisierung: Eine Applikation wird in viele kleine Subroutinen aufgeteilt. Dadurch wird das Programm übersichtlicher, besser lesbar und wartbar.
- Teamarbeit: Die Software kann besser auf mehrere Personen aufgeteilt werden.
- Reduzierte Programmgrösse: Mehrmals durchlaufener Code ist nur einmal im Programmspeicher abgelegt.
- Steigerung der Qualität: Einzelne Subroutinen können gut getestet und dokumentiert werden.
- Wiederverwendbarkeit: Subroutinen können auch in anderen Projekten wieder verwendet werden. Dadurch sinkt der Programmieraufwand.

Bei Funktionsaufrufen in einer Hochsprache übernimmt der Compiler die Steuerung zwischen aufrufendem Programm („caller“) und dem Unterprogramm („callee“). Bei der Assemblerprogrammierung ist dies anders: Hier sind die Programmierer*innen für den korrekten Ablauf verantwortlich. Insbesondere können bei folgenden Aktionen Probleme auftreten:

- Aufrufen der Subroutine und Rücksprung am Ende der Subroutine
- Datenaustausch zwischen aufrufendem Programm und Subroutine (Parameterübergabe, Rückgabewerte)
- lokale Variablen

9.2. Aufruf und Rücksprung

9.2.1. Unverschachtelte Aufrufe von Subroutinen

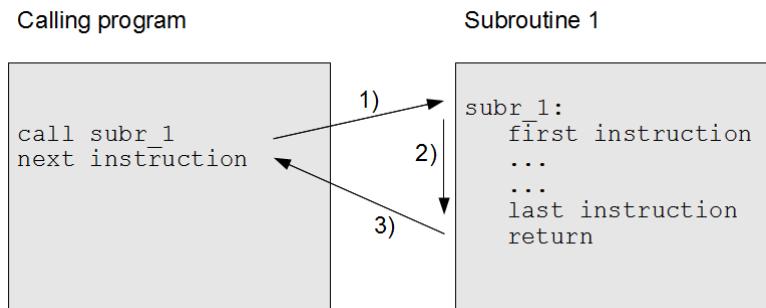


Abbildung 9.1.: Aufruf einer Subroutine

Aus dem aufrufenden Programm wird eine Subroutine aufgerufen. Nach deren Abschluss wird die Codesequenz im aufrufenden Programm fortgesetzt.

Für den Aufruf der Subroutine wird bei ARM der Branch-and-Link Befehl (BL) verwendet. Dieser Befehl setzt den Program-Counter auf die Startadresse der Subroutine und speichert gleichzeitig die Rücksprungadresse (die Adresse der nächsten auszuführenden Instruktion des aufrufenden Programms) im Link-Register r14 (LR).

Beispiel:

```
Hauptprogramm:
  BL  subr_1    @ branch to subr_1 and save return address in lr (r14)
  ...
            @ next instruction

subr_1:
  ...
  MOV pc, lr  @ return
```

Listing 9.1: Aufruf von Subroutinen

Der Rücksprung aus der Subroutine erfolgt durch die Instruktion „MOV pc, lr“. Hier wird der Inhalt des Link-Registers (die Rücksprungadresse) in den Program-Counter (PC) geladen. Anstelle von „MOV pc, lr“ kann man auch „MOV pc, r14“ oder „MOV r15, r14“ schreiben.

Weil die Rücksprungadresse im Link-Register gespeichert ist, darf die Subroutine dieses Register nicht verändern. Insbesondere darf die Subroutine keine weitere Subroutine aufrufen, ohne das Link-Register vorher zu retten. Dazu mehr im nächsten Unterkapitel.

9.2.2. Verschachtelte Aufrufe von Subroutinen, Stack

Eine Subroutine kann wiederum eine andere Subroutine aufrufen. Der Aufruf wird anhand eines einfachen Beispiels erklärt: Aus dem aufrufenden Programm soll eine Subroutine namens „subr_1“ aufgerufen werden. Innerhalb von „subr_1“ wird eine weitere Subroutine „subr_2“ aufgerufen.

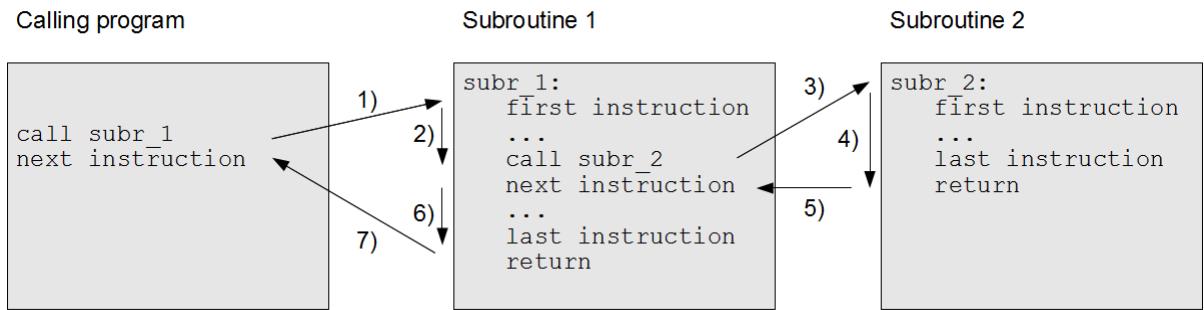


Abbildung 9.2.: Verschachtelter Aufruf von Subroutinen

Eine Subroutine kann sich auch selber aufrufen, was zu rekursivem Code führt. Bei rekursivem Code ist unbedingt darauf zu achten, dass ein Abbruchkriterium existiert, da sonst der Stack überläuft. Wenn immer möglich soll aber auf rekursive Aufrufe verzichtet werden.

Bei verschachtelten Aufrufen wird die Rücksprungadresse auf dem Stack gespeichert. Dies bedingt zwingend, dass wir:

1. Speicher für den Stack im Memory reservieren,
2. den Stack initialisieren.

Ein Stack (Stapel) ist eine dynamische Datenstruktur, die nach dem Prinzip LIFO (Last In First Out) arbeitet. Auf einem Stack können typischerweise folgende Operationen ausgeführt werden:

- Initialisierung
- Push
- Pop

Mit der Operation „Push“ wird ein neues Datenelement auf den Stack gelegt. Mit der Operation „Pop“ wird das zuletzt auf den Stack gelegte Datenelement wieder zurückgeholt. Die Grösse des Stacks wird dabei verändert.

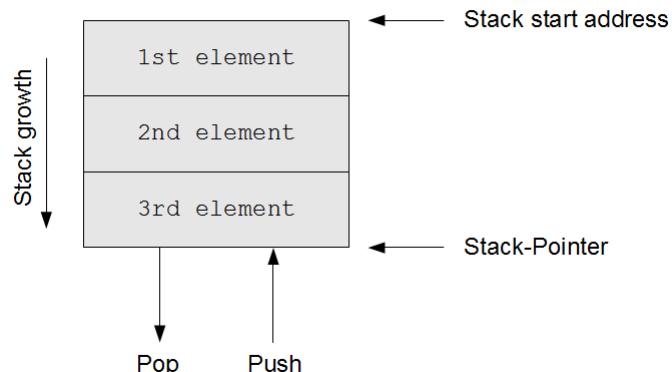


Abbildung 9.3.: Stack

Der Stack-Pointer zeigt auf das Ende des Stacks, d.h. dorthin, wo die nächste Operation ausgeführt wird. Bei ARM-Prozessoren ist das Register r13 der Stack-Pointer. Anstelle von r13 kann man im Code auch „sp“ oder „SP“ schreiben.

Es gibt verschiedene Ansätze, einen Stack zu implementieren:

Der Stack kann „nach unten“ wachsen („descending“), die Daten werden zu den tieferen Adressen hin gespeichert. Oder der Stack kann „nach oben“ wachsen („ascending“), die Daten werden zu den höheren Adressen hin gespeichert. Weiter kann der Stack-Pointer auf die Adresse zeigen, welche zuletzt verwendet wurde, d.h. wo das letzte Element des Stacks gespeichert ist („full“) oder der Stack-Pointer zeigt auf die nächste freie Adresse und somit auf die Stelle, wo das nächste Element gespeichert werden kann („empty“).

Es ist wichtig, dass sowohl der C-Compiler wie auch die Assembler-Programmierer*innen dieselbe Stack-Variante verwenden. In den „ARM Architecture Procedure Call Standards“ (AAPCS, siehe auch Kapitel 9.5) wird für ARM-Prozessoren spezifiziert, dass ein „full descending“ Stack verwendet wird. Die Push-Operation wird dann entweder mit der STR-Instruktion (nur das Link-Register auf dem Stack retten), der STMFD-Instruktion („Store Multiple Full Descending“, um mehrere Register gleichzeitig zu retten) oder der PUSH-Instruktion ausgeführt. Für die Pop-Operation wird entweder die LDR-Instruktion (das Link-Register vom Stack zurückholen), die LDMFD-Instruktion („Load Multiple Full Descending“, um mehrere Register gleichzeitig vom Stack zurück zu lesen) oder die POP-Instruktion verwendet. Anstelle von STMFD kann auch STMDB („Store Multiple Decrement Before“) programmiert werden, was dasselbe ist. Anstelle von LDMFD kann auch LDMIA („Load Multiple Increment After“) verwendet werden. In den nachfolgenden Beispielen werden immer die PUSH- und die POP-Instruktion verwendet.

Beispiel:

Der Code des aufrufenden Programms und der Subroutine 1 aus Abbildung 9.2 lautet wie folgt:

```
main:
    BL  subr_1           @ branch to subr_1, save return address in "lr" (r14)
    ...
    @ next instruction

subr_1:
    PUSH {lr}            @ push lr on stack
    ...
    BL  subr2             @ code subr_1
    ...
    POP {pc}              @ return, pop return address from stack to pc and increment sp
```

Listing 9.2: Aufruf von Subroutinen

Die PUSH-Instruktion reduziert zuerst den Stack-Pointer (SP) um 4 bytes (der Stack wächst negativ). Anschliessend wird die Rücksprungadresse im Link-Register (LR) auf dem Stack gespeichert. Dieser Sachverhalt wird nachfolgend noch etwas genauer erklärt. Als Ausgangslage für dieses Beispiel sollen folgende Registerwerte gelten: PC = 0x8000210, SP = 0x2002000, Subroutine subr_1 startet auf Adresse 0x800022C.

```
#Address:      Code:
0x08000210    BL subr_1           @ branch to subr_1, save return address in "lr" (r14)
0x08000214    ...
               ...
               ...
0x0800022C    PUSH {lr}          @ push lr on stack, return
               ...
```

Listing 9.3: Beispiel Instruktion PUSH

Die Stack-Operation „PUSH {lr}“ zu Beginn von subr_1 hat auf dem Stack folgende Änderungen zur Folge:

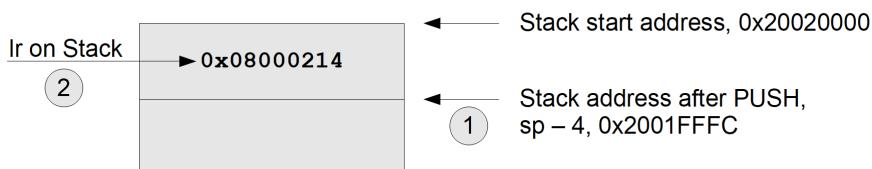


Abbildung 9.4.: Ablegen der Rücksprungadresse auf dem Stack

Nach Beendigung der Subroutine subr_1 erfolgt der umgekehrte Vorgang: Zuerst wird die Rücksprungadresse, die ja auf dem Stack liegt, in den PC kopiert. Anschliessend wird der Stack-Pointer wieder um 4 Bytes „nach oben“

korrigiert. Danach erfolgt der Rücksprung ins aufrufende Programm, d.h. auf die Instruktion nach dem Aufruf der Subroutine. Im Beispiel ist dies Adresse 0x08000214.

Die Initialisierung des Stacks erfolgt je nach CPU-Familie unterschiedlich. Beim ARM Cortex-Mx wird der Stack-Pointer mit dem ersten Eintrag in der Vektortabelle initialisiert. In der Datei „startup.s“ ist die Vektortabelle definiert. Die ersten Einträge sind im folgenden Listing zu sehen:

```
/*----- Section .isr_vector -----*/
/*
 * \brief      The minimal vector table for a Cortex Mx. Note that the
 *             proper constructs must be placed on this to ensure that it
 *             ends up at physical address 0x0000.0000.
 */
...
g_pfnVectors:
    .word _estack           @ start address stack pointer
    .word Reset_Handler     @ start address reset handler
    ...

```

Listing 9.4: Auszug aus der Vektortabelle, der erste Eintrag entspricht dem Initialisierungswert für den Stack-Pointer.

Der Wert von `_estack` wird im Linkerscript „stm32h743zitx_flash.ld“ definiert. Dies entspricht der ersten Adresse nach dem internen RAM des STM32F743. Der Stack wird somit am Schluss des internen RAMs angelegt und wächst danach hin zu tieferen Adressen im RAM:

```
/* Highest address of the user mode stack */
_estack = 0x20020000; /* end of 128K RAM */
```

Listing 9.5: Definition der Startadresse des Stack-Pointers im Linkerscript

Beim Aufrufen und Beenden von Subroutinen werden zusammengefasst folgende Instruktionen verwendet:

Anweisung	Syntax-Beispiel	Operation
Branch-and-Link	BL subr_name	$lr \leftarrow$ Rücksprungadresse $pc \leftarrow pc +$ Sprungdistanz
push single	PUSH {lr}	$sp \leftarrow sp - 4$ Stack \leftarrow Rücksprungadresse
push multiple	PUSH {r0, r2-r4, lr}	$sp \leftarrow sp - 5*4$ Stack \leftarrow Rücksprungadresse + Arbeitsregister
pop single + Rücksprung	POP {pc}	$pc \leftarrow$ Rücksprungadresse $sp \leftarrow sp + 4$ return
pop multiple + Rücksprung	POP {r0, r2-r4, pc}	$pc \leftarrow$ Rücksprungadresse Arbeitsregister \leftarrow Stack $sp \leftarrow sp + 5*4$ return

Tabelle 9.1.: Stackoperationen beim Aufrufen und Beenden von Subroutinen

Wenn innerhalb der Subroutine 2 aus Abbildung 9.2 die Rücksprungadresse ebenfalls auf dem Stack gerettet wird, so ergibt sich folgender Inhalt des Stacks:

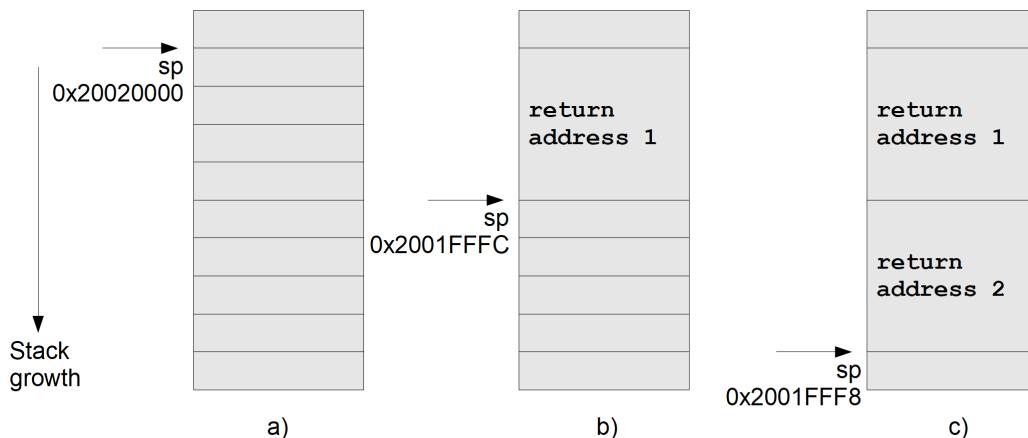


Abbildung 9.5.: Rücksprungadressen auf dem Stack

- a) Stack im aufrufenden Programm
- b) Stack innerhalb von „subr_1“
- c) Stack innerhalb von „subr_2“

9.3. Retten von Registern

Betrachten wir folgenden Code, welcher eine Schleife 10 mal ausführen soll, und aus dieser Schleife die Subroutine „subr_3“ aufruft:

```

start: MOV r4, #0          @ initialize loop variable r4
loop:  BL subr_3           @ branch to subroutine
      ADD r4, r4, #1        @ increment loop variable
      CMP r4, #10            @ check if 10 loops done
      BNE loop               @ no, branch to label loop
      ...
  
```

Listing 9.6: Aufrufendes Programm

```

subr_3: PUSH {lr}          @ push lr on the stack
...
MOV r4, #0                  @ modify r4
...
POP {pc}                   @ return
  
```

Listing 9.7: fehlerhafte Subroutine

Das Problem in dieser Codesequenz besteht darin, dass das Register r4, welches im aufrufenden Programm für die Schleife verwendet wird, in der Subroutine subr_3 verändert wird!

In diesem Falle gilt das Verursacherprinzip: Wenn eine Subroutine Register verändert, muss sie deren Inhalt zuerst retten und anschliessend zurückladen (Ausnahmen siehe Kapitel 9.5). Dazu wird ebenfalls der Stack verwendet. Zum Retten der Register auf dem Stack werden wieder die Instruktionen aus Tabelle 9.1 verwendet.

Die Subroutine subr_3 muss nicht nur das Link-Register, sondern auch das Register r4 auf dem Stack retten. Der Code für die Subroutine 3 muss deshalb wie folgt geändert werden:

```

subr3: PUSH {r4,lr}         @ push r4 and lr on the stack
...
MOV r4, #0                  @ modify r4
...
POP {r4,pc}                 @ pop register and return address
  
```

Listing 9.8: korrekte Subroutine

Beachten Sie, dass durch die „POP“-Instruktion die Rücksprungadresse direkt in den PC kopiert wird.

9.4. Parameterübergabe

Wenn zwischen dem aufrufenden Programm und der Subroutine Parameter übergeben werden müssen, gibt es folgende Möglichkeiten:

1. Parameterübergabe über Register
2. Parameterübergabe über den Stack

C-Funktionen und Assembler-Subroutinen können viele Parameter haben. Es ist sehr wichtig, dass sich sowohl die C-Compiler als auch die Assembler-Programmierer*innen an dieselben Richtlinien halten. Der ARM Architecture Procedure Call Standard AAPCS (siehe auch Kapitel 9.5) definiert deshalb, wie die Parameter bei ARM-Prozessoren übergeben werden müssen: **Die ersten vier Parameter werden in den Registern r0 bis r3 übergeben. Die übrigen Parameter werden in umgekehrter Reihenfolge auf dem Stack abgelegt.**

9.4.1. Parameterübergabe über Register

Grundsätzlich bestehen folgende Möglichkeiten für die Parameterübergabe:

Call by Value:

Als Parameter wird eine Kopie des Wertes in einem Register (r0 bis r3) übergeben. Die Subroutine kann nicht auf die ursprüngliche Variable zugreifen.

Beispiel:

```
LDR r0, =variable    @ r0 points to address of variable
LDR r0, [r0]          @ copy value of variable into r0
BL subr_4            @ call subroutine, parameter is passed in r0
```

Listing 9.9: Call by Value

Call by Reference:

Als Parameter wird die Adresse einer Variablen in einem Register (r0 bis r3) übergeben. Die Subroutine kann dadurch die ursprüngliche Variable verändern.

Beispiel:

```
LDR r0, =variable    @ r0 points to address of variable
BL subr_4            @ call subroutine, parameter is passed in r0
```

Listing 9.10: Call by Reference

Beispiel:

Wir betrachten die Subroutine „sum“ mit drei Parametern, welche in der Programmiersprache C wie folgt programmiert wird:

```
int v1 = 2;
int v2 = 5;
int res;

void sum(int p1, int p2, int* res){
    *res = p1 + p2;
}

main(void){
    sum(v1, v2, &res);
}
```

Listing 9.11: C-Funktion „sum“ mit 3 Parametern

Das aufrufende Programm wird in Assembler wie folgt implementiert:

```

    .data
v1: .word 2          @ 1st parameter
v2: .word 5          @ 2nd parameter
res: .word 0          @ result

    .text
main:
    ...
# calling subroutine sum
    LDR r0, =p1      @ copy address of v1 into r0
    LDR r0, [r0]      @ read value v1, 1st parameter in r0, passed by value
    LDR r1, =p2      @ copy address v2 into r1
    LDR r1, [r1]      @ read value v2, 2nd parameter in r1, passed by value
    LDR r2, =res      @ read address res, 3rd parameter in r2, passed by reference
    BL sum

```

Listing 9.12: Aufruf der Subroutine in Assembler

Die Subroutine „sum“ berechnet die Summe von 2 Parametern und gibt das Ergebnis im 3. Parameter by reference zurück. r3 wird als Zwischenergebnis verwendet.

```

ADD r3, r0, r1      @ temp = p1 + p2, p1 passed in r0, p2 in r1, by value
STR r3, [r2]         @ *res = temp, res in r2 by reference
MOV pc,lr           @ return

```

Listing 9.13: Subroutine „sum“ in Assembler

Bemerkung: Gemäss AAPCS ist es korrekt, Register r3 hier nicht auf dem Stack zu retten (siehe Kapitel 9.5).

9.4.2. Parameterübergabe über den Stack

Gemäss AAPCS muss die aufrufende Funktion die ersten vier Parameter in den Registern r0 bis r3 übergeben, die restlichen Parameter werden in umgekehrter Reihenfolge auf den Stack kopiert.

Beispiel:

Anhand der Subroutine super_sum, welche untenstehende C-Schnittstelle hat, soll dies erklärt werden.

C-Deklaration der Subroutine super_sum:

```
int super_sum(int p1, int p2, int p3, int p4, int p5, int p6);
```

Die Parameter p1 bis p4 werden in den Registern r0 bis r3 übergeben, p5 und p6 werden über den Stack übergeben. Das Resultat wird nicht als „call by value“, sondern als Rückgabewert zurück gegeben. Der Rückgabewert wird gemäss AAPCS in r0 übergeben. Die aufrufende Funktion wird somit in Assembler wie folgt programmiert:

```

main:
    ...
# calling subroutine super_sum
    LDR r0, =p1      @ copy address p1 into r0
    LDR r0, [r0]      @ read value p1, 1st parameter r0 by value
    LDR r1, =p2      @ copy address p2 into r1
    LDR r1, [r1]      @ read value p2, 2nd parameter r1 by value
    LDR r2, =p3      @ copy address p3 into r2
    LDR r2, [r2]      @ read value p3, 3rd parameter r2 by value
    LDR r3, =p4      @ copy address p4 into r3
    LDR r3, [r3]      @ read value p4, 4th parameter r3 by value
    LDR r12, =p6      @ copy address p6 into r12
    LDR r12, [r12]    @ read value p6, 6th parameter
    PUSH.w {r12}     @ push value of p6 on stack
    LDR r12, =p5      @ copy address p5 into r12
    LDR r12, [r12]    @ read value p5, 5th parameter
    PUSH.w {r12}     @ push value of p5 on stack
    BL super_sum     @ call subroutine
    LDR r1, =res      @ copy address of res into r1
    STR r0, [r1]      @ copy return value to variable res

```

Listing 9.14: Aufrufende Funktion von „super_sum“ in Assembler

Der Code der Subroutine super_sum wird wie folgt implementiert:

- r4 wird als temporäre Variable für das Resultat verwendet
- r5 wird als buffer zum speichern der Parameter auf dem Stack verwendet
- r4 und r5 müssen beide gerettet werden

```
super_sum:
    PUSH {r4, r5, lr}      @ push r4, r5 and lr on stack
    MOV r4, r0              @ temp = p1
    ADD r4, r1              @ temp += p2
    ADD r4, r2              @ temp += p3
    ADD r4, r3              @ temp += p4
    LDR r5, [sp, #12]       @ buffer = p5
    ADD r4, r5              @ temp += p5
    LDR r5, [sp, #16]       @ buffer = p6
    ADD r4, r5              @ temp += p6
    MOV r0, r4              @ copy return value to r0
    POP {r4, r5, pc}        @ pop r4, r5 and lr, return
```

Listing 9.15: Subroutine „super_sum“ in Assembler

Der Stack wird durch den Aufruf der Subroutine super_sum wie folgt modifiziert:

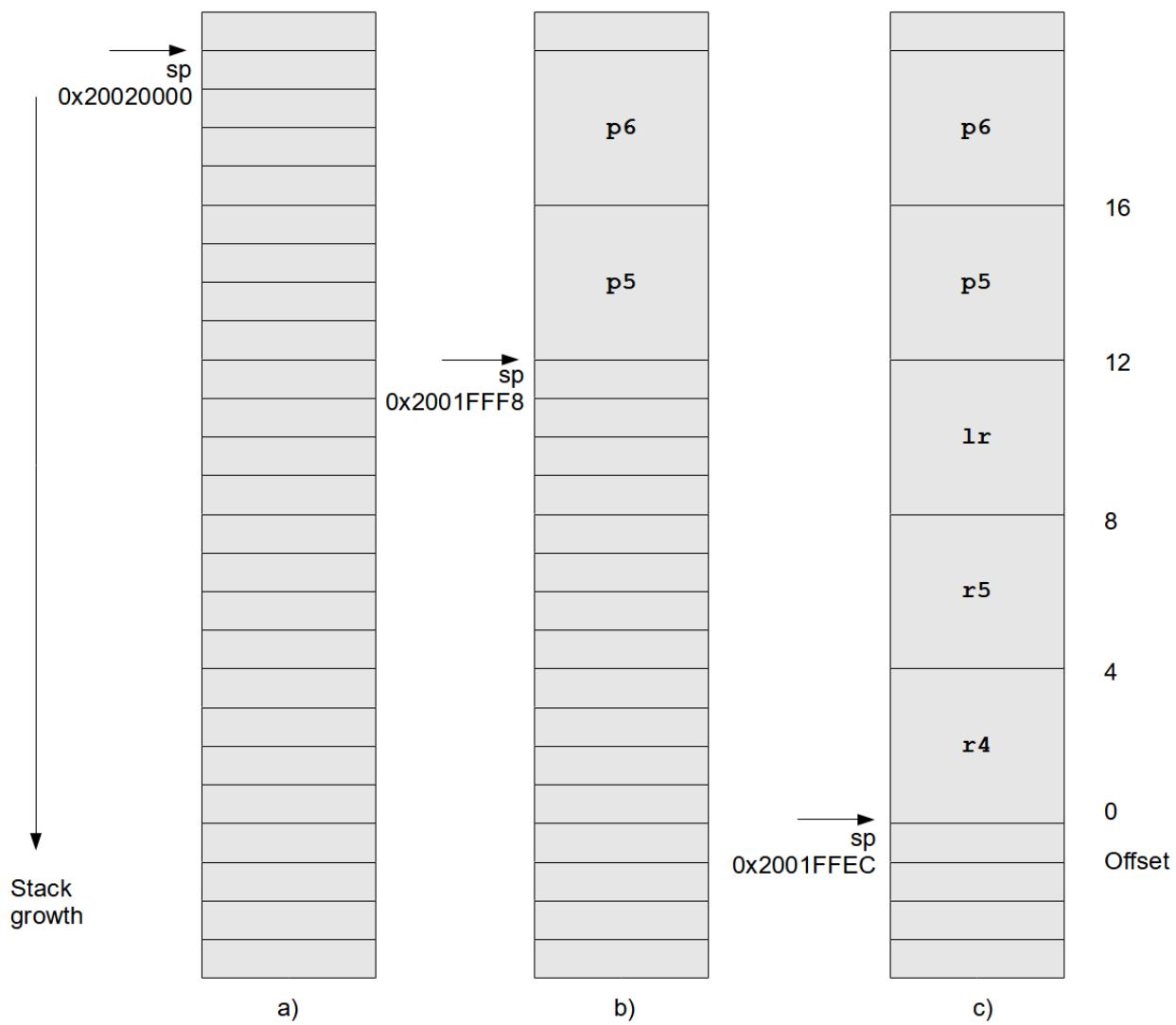


Abbildung 9.6.: Parameterübergabe auf dem Stack:

- vor dem Aufruf der Subroutine `super_sum` und dem Speichern der Parameter auf dem Stack
- nach dem Eintritt in die Subroutine, die Parameter **p5** und **p6** liegen auf dem Stack
- innerhalb der Subroutine nach der Instruktion `PUSH`

9.5. AAPCS

Der AAPCS (ARM Architecture Procedure Call Standard) legt fest, wie die Register der CPU zu verwenden sind. Diese Konvention ist notwendig damit unterschiedliche C-Compiler und Assembler dieselben Richtlinien für die Parameterübergabe, das Retten von Registern usw. einhalten.

Detaillierte Angaben zum AAPCS finden Sie im Anhang C. An dieser Stelle wird lediglich die Verwendung der Register bei Subroutinenaufrufen zusammengefasst.

Die Register einer ARM-CPU werden gemäss AAPCS wie folgt verwendet:

Register	AAPCS Synonym	Beschreibung
r0	a1	Argument 1 / Integer-Rückgabewert 32-Bit / Scratch-Register
r1	a2	Argument 2 / Integer-Rückgabewert 64-Bit / Scratch-Register
r2	a3	Argument 3 / Scratch-Register
r3	a4	Argument 4 / Scratch-Register
r4	v1	Variable Register 1
r5	v2	Variable Register 2
r6	v3	Variable Register 3
r7	v4	Variable Register 4
r8	v5	Variable Register 5
r9	v6 / SB / TR	Platformspezifisches Register / Variable Register 6
r10	v7	Variable Register 7
r11	v8	Variable Register 8
r12	ip	Intra-Procedure Call / Scratch-Register
r13	sp	Stack-Pointer
r14	lr	Link-Register
r15	pc	Program-Counter

Tabelle 9.2.: Verwendung der ARM CPU-Register gemäss AAPCS

Die Register r0 bis r3 werden für die Parameterübergabe an Subroutinen sowie für die Rückgabewerte verwendet. Diese Register können durch die aufgerufene Subroutine auch als Zwischenspeicher verwendete und dadurch modifiziert werden (Scratch-Register). Das Register r12 kann vom Linker als Scratch-Register für den Aufruf von Subroutinen verwendet werden. Die aufrufende Subroutine muss also davon ausgehen, dass nach dem Rücksprung aus der aufgerufenen Subroutine die Register r0 bis r3 sowie r12 verändert wurden.

r4 bis r8, sowie r10 und r11 können in einer Subroutine verwendet werden, um lokale Variablen zu speichern. Es gilt aber das Verursacherprinzip, d.h. dass die Register auf dem Stack gerettet und am Schluss der Subroutine wieder hergestellt werden müssen.

r9 wird plattformspezifisch verwendet. D.h. jede Plattform kann diesem Register eine spezielle Verwendung zuweisen. Gibt es keine spezielle Verwendung, so kann r9 als Variable Register eingesetzt werden.

Die Register r13 bis r15 haben spezielle Eigenschaften: r13 wird als Stack-Pointer verwendet (SP), r14 als Link-Register (LR) zum Speichern der Rücksprungadresse aus Subroutinen und r15 als Program-Counter (PC).

9.6. Lokale Variablen

Insbesondere in Hochsprachen werden oft lokale Variablen in einer Funktion benötigt.

Beispiel:

```
void sub_localvar(void)
{
    unsigned int myArray[10];           // 10 * 4 Byte local variable
    ...
    ...
}
```

}

Listing 9.16: Anlegen von lokalen Variablen in C

Werden nur wenige lokale Variablen benötigt, so können diese in den Registervariablen (r4 bis r8, r10 und r11) abgelegt werden. Wenn zusätzliche Variablen benötigt werden, so müssen diese auf dem Stack gespeichert werden. Eine Funktion legt deshalb einen lokalen Variablenbereich auf dem Stack an, dessen Größe den zu speichernden lokalen Variablen entspricht. Zusätzlich wird ein Register verwendet, welches auf den Bereich dieser lokalen Variablen zeigt: der Frame-Pointer. Als Frame-Pointer wird üblicherweise r11 verwendet. Der Code einer Subroutine mit lokalen Variablen kann in Assembler wie folgt implementiert werden:

```
sub_localvar:
    PUSH    {r11, lr}      @ push r11 and return address, (1)
    SUB     sp, sp, #40    @ allocate 10 * 4 bytes local variable space, (2)
    MOV     r11, sp        @ r11 points to first local variable, (3)
    ...
    ...
    ADD     sp, sp, #40    @ free local variable space
    POP    {r11, pc}       @ pop r11 and lr, return
```

Listing 9.17: Anlegen von lokalen Variablen in Assembler

Während dem Anlegen der lokalen Variablen verändert sich der Stack wie folgt:

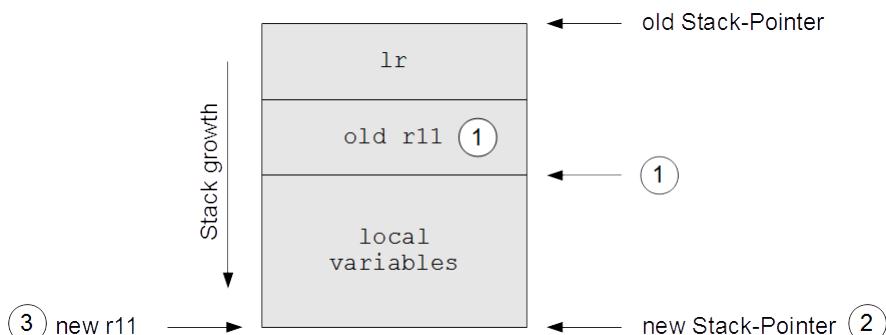


Abbildung 9.7.: Lokale Variablen auf dem Stack

Der entstehende lokale Variablenbereich kann nun über den Frame-Pointer r11 und einen positiven Offset adressiert werden. Wie schon für die Offsetberechnung bei den Übergabeparametern sind auch hier die Programmierer*innen für die korrekte Berechnung verantwortlich!

In Abbildung 9.7 zeigt der Frame-Pointer auf die tiefste Adresse des Bereiches mit den lokalen Variablen. Als Variante könnte man den Frame-Pointer auch so initialisieren, dass er auf die höchste Adresse des Bereiches zeigt. Der Offset zum Frame-Pointer für den Zugriff auf eine lokale Variable ist dann negativ. Weiter kann auch auf den Frame-Pointer verzichtet werden und über den Stack-Pointer auf die lokalen Variablen zugegriffen werden.

Werden lokale Variablen und Parameterübergabe über den Stack (vergleiche Kapitel 9.4.2) gleichzeitig verwendet, so kann die Berechnung des Parameter-Offsets auf dem Stack auch mit dem Frame-Pointer statt dem Stack-Pointer erfolgen. Bemerkung: Einige C-Compiler legen immer einen Frame-Pointer an, auch wenn in einer Subroutine gar keine lokalen Variablen verwendet werden. Dies ergibt einerseits einen zur Ausführungszeit langsameren Code, ist aber andererseits für den Compiler einfacher zur Berechnung der Offsets für die Parameter.

9.7. Fehlerquellen

Im Zusammenhang mit dem Aufrufen und Beenden von Subroutinen sowie den entsprechenden Stack-Operationen ist besondere Vorsicht geboten. Generell gilt:

- Subroutinen immer mit „BL“ (und nicht mit „B“) aufrufen!
- Subroutinen verlassen indem die Rücksprungadresse in den Program-Counter kopiert wird (entweder direkt aus dem Link-Register oder vom Stack holen)!

- Werden in einer Subroutine Register verändert (ausser Scratch-Register), so sind diese auf dem Stack zu retten (Reihenfolge beachten, d.h. zuletzt gerettet → zuerst entfernt). Die Anzahl der „Pop“ muss mit der Anzahl „Push“ übereinstimmen!
- Stack genügend gross dimensionieren, damit es keinen Überlauf gibt!
- Fehlerhafte Berechnung der Offsets bei der Parameterübergabe über den Stack!

Wenn Sie im Zusammenhang mit dem Stack Fehler machen, wird Ihr Programm unweigerlich abstürzen, da die Rücksprungadressen nicht mehr stimmen!

9.8. Vergleich Subroutinen und Makros

Der Code einer Subroutine ist nur einmal im Speicher abgelegt, auch wenn die Subroutine mehrmals von unterschiedlichen Stellen im Programm aufgerufen wird. Der Code eines Makros hingegen wird jedes mal anstelle des Makroaufrufs im Speicher abgelegt: Dadurch kann der Code wesentlich grösser werden als wenn eine Subroutine verwendet würde. Makros erlauben keine Rekursion.

Es stellt sich deshalb die Frage, wieso überhaupt Makros verwendet werden. Die Antwort liegt im zeitlichen Verhalten begründet: Ein Subroutinen-Aufruf mit „BL“ und anschliessendem Rücksprung benötigt Zeit (auch wenn dies nur 2 Instruktionen sind). In sehr zeitkritischen Applikationen kann dies den Ausschlag zu Gunsten eines Makros geben.

Generell gilt: Verwenden Sie im „Normalfall“ Subroutinen. Wenn Sie sehr zeitkritische Codesequenzen haben, dann verwenden Sie Makros.

Anmerkung: In der Regel holen Sie mit einem guten Softwaredesign und guten Algorithmen wesentlich mehr aus dem Prozessor heraus, als wenn Sie anstelle von Subroutinen Makros verwenden.

9.9. Heap

Der Heap ist ein Speicherbereich, aus dem zur Laufzeit (dynamisch) Speicher angefordert werden kann. Die Rückgabe dieses Speichers erfolgt in beliebiger Reihenfolge.

Der Heap wird je nach Programmiersprache unterschiedlich unterstützt. In ANSI-C gibt es dafür Funktionen wie malloc(), calloc() und realloc() für die Anforderung von Speicher sowie free() für die Rückgabe des Speichers. Die Applikation (und damit die Programmierer*innen) ist dafür verantwortlich, dass der Speicher wieder zurück gegeben wird. In C++ gibt es zusätzlich die Methoden new() und delete(). In Java läuft die Rückgabe des Speichers automatisch. Wenn Sie Assembler programmieren, müssen Sie die Heap-Funktionalität selber implementieren, oder Sie rufen die Standard-Funktionen aus C auf.

Die Verwaltung des Heap ist für das Laufzeitsystem aufwändig. Die Anforderungen an die Heap-Verwaltung sind insbesondere eine hohe Geschwindigkeit, eine effiziente Speichernutzung sowie ein kleiner Verwaltungsaufwand.

Durch die Anwendung von dynamischem Speicher ergeben sich viele Probleme und Fehlerquellen:

- Der Speicher kann mit zunehmender Laufzeit fragmentiert sein (d.h. der Speicher wird in viele kleine Blöcke unterteilt).
- Die Programmierer*innen „vergessen“, den angeforderten Speicher wieder freizugeben. Die Folge davon ist, dass es irgendwann zu wenig Speicher hat.
- Die Verwaltung des Heap, und somit die Suche nach freiem Speicher, ist nicht deterministisch, d.h. die Zeit für die Ausführung dieser Funktionen kann nicht vorausgesagt werden.

Aufgrund der oben genannten Probleme werden in Embedded Systems und insbesondere bei Echtzeitanwendungen keine dynamischen Speichern verwendet, oder diese werden nur eingeschränkt, z.B. während der Initialisierung des Systems, zugelassen.

10. Startup-Code

Die nächsten Kapitel beschäftigen sich damit, wie in einem Projekt C- und Assembler-Code kombiniert werden können.

10.1. Startup-Vorgang Übersicht

Als Startup-Vorgang wird der Ablauf bezeichnet, der zwischen Reset und Applikationsstart ausgeführt wird (Startup-Code). Der gesamte Startup-Vorgang ist in Abbildung 10.1 grafisch dargestellt. Die einzelnen Aufgaben werden in den folgenden Unterkapiteln beschrieben.

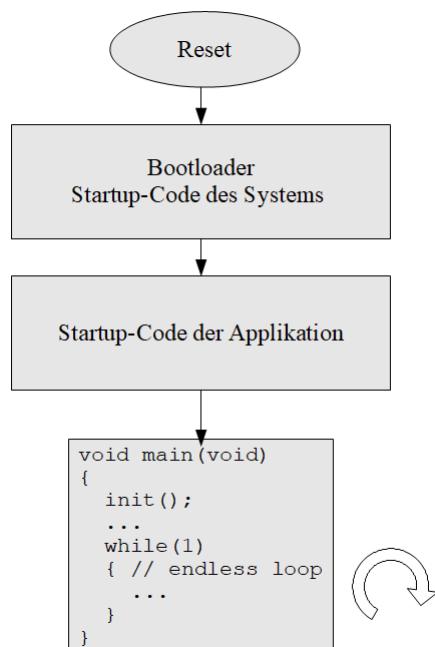


Abbildung 10.1.: Übersicht Startup-Vorgang

10.2. Reset

Der Reset kann durch folgende Ereignisse ausgelöst werden:

- Power up: Das Einschalten des Systems löst einen Reset aus, der das System in einen definierten Zustand bringt.
- Reset-Taster: Anwender können das System manuell zurücksetzen, indem sie den Reset-Taster betätigen.
- Watchdog: Eine Überwachungsschaltung, welche prüft, ob das System „hängt“ und in diesem Falle einen Reset auslöst. Für die Implementation des Watchdogs wird ein Timer verwendet, welcher durch die Applikation periodisch zurückgesetzt werden muss (siehe Kapitel 18). „Hängt“ die Applikation, so überläuft der Timer und löst einen Reset aus. Viele heutige Prozessoren haben einen Watchdog-Timer integriert.

- Spannungsüberwachung: Sie detektiert eine fehlerhafte Speisung (tiefe Spannung, kurze Spannungseinbrüche).
- Software: Bei einigen Prozessoren kann die Software einen Reset auslösen. Dies kann beispielsweise benutzt werden, um bei fehlerhaften Hardware-Tests das System neu zu starten.

Der Reset bringt den Prozessor in einen definierten Anfangszustand. Bei vielen Prozessoren wird der Reset-Vektor als erster Vektor in der Vektor-Tabelle eingetragen. Hier wird die Einsprungadresse des Codes definiert, der nach dem Reset-Ereignis ausgeführt wird. Das ist entweder der Bootloader oder, falls dieser nicht vorhanden ist, der Startup-Code der Applikation.

10.3. Bootloader

Der Bootloader ist diejenige Software, die ein System soweit initialisiert, dass überhaupt etwas läuft. Die Aufgaben des Bootloaders sind:

1. Initialisieren der wichtigsten Prozessor-Register, insbesondere der Chip>Selects, sodass externer Speicher korrekt angesprochen werden kann.
2. Bei leistungsfähigen Systemen wird die Applikation aus dem RAM und nicht aus dem Flash ausgeführt (weil der Zugriff auf das RAM schneller ist). In diesen Systemen wird die Applikation oder das Betriebssystem durch den Bootloader vom Flash (oder anderen nichtflüchtigen Speichermedien) in den Arbeitsspeicher (RAM) kopiert.
3. Start der Applikation oder des Betriebssystems.

Falls ein Betriebssystem eingesetzt wird, lädt der Bootloader zuerst dieses. Linux beispielsweise ist bei vielen Embedded Systems komprimiert im Flash abgelegt und wird für die Ausführung vom Bootloader ins RAM kopiert. Die Applikationen werden anschliessend vom Betriebssystem geladen.

10.4. Startup-Code der Applikation

10.4.1. Übersicht

Der Startup-Code wird von einer oder mehreren Assembler-Routinen ausgeführt, die sich in einer eigenen Startup-Datei befinden. Die Namen dieser Dateien sind abhängig von der Entwicklungsumgebung, heissen aber oft „crt0“ oder „startup“. In der Entwicklungsumgebung STM32CubeIDE, welche im Unterricht verwendet wird, finden Sie die Startup-Datei eines Projektes im Ordner „Core\Startup“, sie heisst „startup_stm32h743zitx.s“.

Wenn Sie in C programmieren, müssen Sie sich normalerweise nicht um den Startup-Vorgang kümmern, die Entwicklungsumgebung kopiert den auszuführenden Startup-Code automatisch, wenn Sie ein neues Projekt anlegen. Ihre Applikation beginnt dann mit der Funktion main(). Es gibt aber immer wieder Fälle, wo der Startup-Code modifiziert werden muss, dann müssen Sie Ihre Assembler-Kenntnisse zu Hilfe nehmen.

Der Startup-Code erledigt folgende Aufgaben:

- Anlegen der Vektor-Tabelle der Applikation
- Initialisieren diverser CPU-Register
- Initialisieren Cache, MMU
- Initialisieren der Stacks
- Initialisierte globale Variablen setzen
- Uninitialisierte Variablen auf 0 setzen
- Board-Support-Package (BSP) initialisieren

Bemerkung: Bei Systemen mit Bootloader übernimmt der Bootloader einen Teil der obigen Funktionen.

10.4.2. Vektortabelle

Die Vektortabelle wird in der Datei „startup.s“ definiert. Wenn für eine Applikation eigene Exception-Handler definiert werden, müssen diese in der Vektortabelle eingetragen werden. Einen Auszug aus dieser Vektortabelle finden Sie im folgenden Listing. Vergleichen Sie diese Einträge auch mit der in Kapitel 14.7 Tabelle 14.2 beschriebenen Vektortabelle für Cortex-Mx Prozessoren.

```
*****  
*  
* The minimal vector table for a Cortex M. Note that the proper constructs  
* must be placed on this to ensure that it ends up at physical address  
* 0x0000.0000.  
*  
*****  
.section .isr_vector,"a",%progbits  
.type g_pfnVectors, %object  
.size g_pfnVectors, .-g_pfnVectors  
  
g_pfnVectors:  
.word _estack  
.word Reset_Handler  
  
.word NMI_Handler  
.word HardFault_Handler  
.word MemManage_Handler  
.word BusFault_Handler  
.word UsageFault_Handler  
.word 0  
.word 0  
.word 0  
.word 0  
.word SVC_Handler  
.word DebugMon_Handler  
.word 0  
.word PendSV_Handler  
.word SysTick_Handler  
  
/* External Interrupts */  
.word WWDG_IRQHandler /* Window WatchDog */  
.word PVD_AVD_IRQHandler /* PVD/AVD through EXTI Line detection */  
.word TAMP_STAMP_IRQHandler /* Tamper and TimeStamps through the EXTI line */  
.word RTC_WKUP_IRQHandler /* RTC Wakeup through the EXTI line */  
.word FLASH_IRQHandler /* FLASH */  
.word RCC_IRQHandler /* RCC */  
.word EXTI0_IRQHandler /* EXTI Line0 */  
.word EXTI1_IRQHandler /* EXTI Line1 */  
.word EXTI2_IRQHandler /* EXTI Line2 */  
.word EXTI3_IRQHandler /* EXTI Line3 */  
.word EXTI4_IRQHandler /* EXTI Line4 */  
.word DMA1_Stream0_IRQHandler /* DMA1 Stream 0 */  
...
```

Listing 10.1: Anfang der Vektortabelle für den STM32F743

10.4.3. CPU Register

Die wichtigsten Register wurden bereits durch den Bootloader initialisiert. An dieser Stelle können weitere Register initialisiert werden, z.B. CPU-Speed usw.

10.4.4. Initialisierte globale Variablen setzen

In C können globale Variablen initialisiert werden, beispielsweise:

```
static int myInt = 5;
```

Listing 10.2: Initialisierte globale Variable in C

Haben Sie sich schon einmal überlegt, wer für die Initialisierung dieser Variablen verantwortlich ist? Der Code wird ja nicht etwa durch eine Funktion zur Laufzeit aufgerufen. Dies ist eine weitere Aufgabe des Startup-Codes. Die globalen Variablen werden durch das sogenannte ROM-Processing vom Startup-Code initialisiert.

Abbildung 10.2 zeigt, wie globale Variablen initialisiert werden:

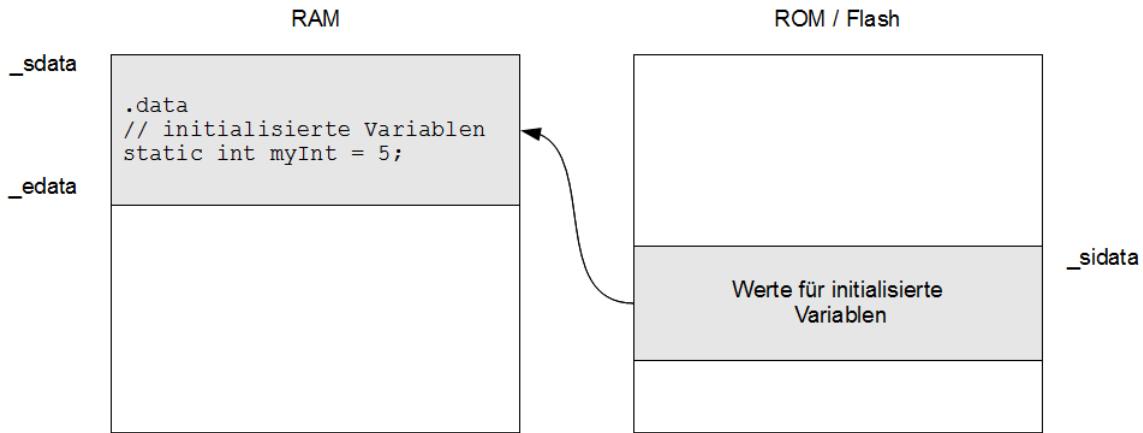


Abbildung 10.2.: Initialisierung von globalen Variablen

Der Startup-Code im File „startup.s“ kopiert nun die Werte für die zu initialisierenden Variablen vom Flash ins RAM. Im Code sieht das wie folgt aus:

```
/* Define Address spaces. defined in linker script. */
.word _sidata /* start address for the initialization values of the .data section. */
.word _sdata /* start address for the .data section. */
.word _edata /* end address for the .data section. */

...
/* Copy the data segment initializers from flash to SRAM */
    MOVS    r1, #0
    B      LoopCopyDataInit

CopyDataInit:
    LDR     r3, =_sidata
    LDR     r3, [r3, r1]
    STR     r3, [r0, r1]
    ADDS   r1, r1, #4

LoopCopyDataInit:
    LDR     r0, =_sdata
    LDR     r3, =_edata
    ADDS   r2, r0, r1
    CMP    r2, r3
    BCC    CopyDataInit
    ...
```

Listing 10.3: Startup-Code für die Initialisierung von initialisierten Variablen

`_sidata` ist die Startadresse des Bereichs im Flash, wo die Initialisierungs-Werte abgelegt sind. `_sdata` und `_edata` bezeichnen Anfang und Ende des `.data` Bereiches für die initialisierten Variablen. `_sidata`, `_sdata` und `_edata` werden im Linkerskript `stm32h743zitx_flash.ld` definiert.

Bemerkung: Wenn Sie mit einem Debugger arbeiten, laden Sie das Programm über die Debug-Schnittstelle ins Flash. In diesem Falle initialisiert Ihnen der Debugger automatisch auch die globalen Variablen im RAM. Bei marktfertigen Geräten haben Sie aber keinen Debugger mehr, der ihnen diese Initialisierung durchführt. In diesem Falle brauchen Sie zwingend den Startup-Code für die Initialisierung der globalen Variablen.

10.4.5. Uninitialisierte globale Variablen auf 0 setzen

Nicht initialisierte Variablen und der Stack werden in der .bss Section abgelegt. Alle diese Variablen werden mit 0 initialisiert, also gelöscht. Der Code ist wieder im File „startup.s“ zu finden und ist ähnlich aufgebaut wie für die initialisierten Variablen:

```
/* Define Address spaces. defined in linker script. */  
    .word _sbss      /* start address for the .bss section. */  
    .word _ebss      /* end address for the .bss section. */  
    ...  
    LDR    r2, =_sbss  
    B     LoopFillZeroBSS  
  
/* Zero fill the bss segment. */  
FillZeroBSS:  
    MOVS   r3, #0  
    STR    r3, [r2], #4  
  
LoopFillZeroBSS:  
    LDR    r3, =_ebss  
    CMP    r2, r3  
    BCC    FillZeroBSS  
    ...
```

Listing 10.4: Startup-Code für das Löschen von uninitialisierten Variablen

10.4.6. Weitere Initialisierungen

Der Startup-Code führt weitere Initialisierungen durch. Am Schluss des Starutup-Codes wird die Funktion main() aufgerufen. Sollte der Code aus der main-Funktion zurück springen, wird am Schluss des Startup-Codes eine Endlosschleife ausgeführt.

```
/* Call the clock system intitialization function.*/  
    bl SystemInit  
/* Call static constructors */  
    bl __libc_init_array  
/* Call the application's entry point.*/  
    bl main  
/* normally, there is no return from main in an embedded system */  
ProgramFinish:  
    B     ProgramFinish /* do nothing */
```

Listing 10.5: Initialisierung des BSP und Aufruf von main()

10.5. Hauptprogramm

Für einfachere Anwendungen benötigen Sie kein Betriebssystem. Das Hauptprogramm der Applikation ist etwa wie folgt aufgebaut:

In einem ersten Schritt initialisieren Sie Ihre Applikation:

- Die Hardwareperipherie über die dazugehörigen Register.
- Die Daten: Variablen, dynamische Datenstrukturen usw.

Die Initialisierung der Hardwareperipherie kann zeitkritisch sein, wenn Sie Ausgänge möglichst schnell auf definierte Pegel setzen müssen (z.B. die Ansteuerung von Motoren)! Eventuell muss dies schon im Startup-Code oder sogar hardwaremäßig gelöst werden.

In einem zweiten Schritt werden die einzelnen Funktionen der Applikation zyklisch aufgerufen. Dies können Sie durch eine Endlosschleife realisieren.

Daraus ergibt sich folgendes Nassi-Shneiderman-Diagramm für das Hauptprogramm:

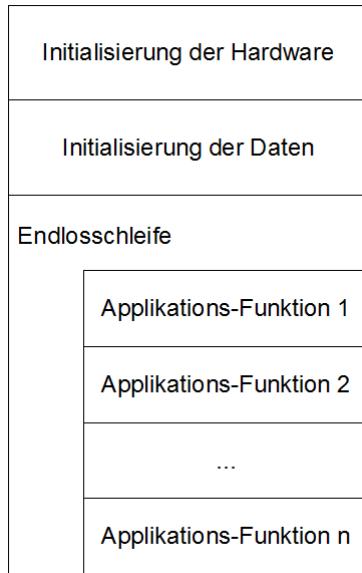


Abbildung 10.3.: Hauptprogramm

Wenn Sie mit ISR arbeiten, sollten Sie unbedingt darauf achten, in diesen Routinen nicht zu viel Zeit zu verlieren. Machen Sie in den ISR nur das Allernötigste, arbeiten Sie mit Flags oder State-Events, um den Rest der Arbeit ausserhalb der ISR zu machen! Beispiel: Wenn Sie über die serielle Schnittstelle ein Byte empfangen und dadurch ein Interrupt ausgelöst wird, sollten Sie in der ISR nur das empfangene Byte in einen Buffer kopieren. Die Auswertung der Daten sollte nicht in der ISR erfolgen (ausser es ist wirklich simpel).

Ein sehr kritischer Punkt ist der Datenaustausch zwischen ISR und den Funktionen des Hauptprogramms. Hier bleibt Ihnen keine andere Wahl als über globale Variablen zu kommunizieren (ausser Sie verwenden ein Betriebssystem). Aber Achtung! Die ISR kann Ihnen diese Daten überschreiben, ohne dass Sie das in den Applikationsfunktionen merken. Kritisch ist das vor allem bei strukturierten Daten. Nehmen wir an, Sie haben einen Empfangsbuffer von 10 Bytes, in welchen Sie in der ISR Daten schreiben und in einer Applikationsfunktion diese Daten wieder auslesen. Nehmen wir weiter an, Sie haben in der Applikationsfunktion gerade das fünfte Byte ausgelesen und es wird ein Interrupt ausgelöst, weil die Hardware Ihrer Schnittstelle die nächsten 10 Bytes empfangen hat. Die ISR kann Ihnen nun die 10 Bytes des Empfangsbuffers überschreiben und die Applikationsfunktion wird anschliessend das sechste Byte auslesen, als wäre nichts geschehen. Schlussendlich werden Sie mit inkonsistenten Daten arbeiten. Um dieses Problem zu umgehen, müssen Sie Mechanismen einführen, welche diese Operationen gegenseitig verriegeln, d.h. während Sie in der Applikation den Empfangsbuffer auslesen, darf kein Empfangs-Interrupt auftreten (Sie müssen also die Interrupts der Schnittstelle während dieser Zeit disablen). Das Ganze kann sehr komplex werden wenn Sie mehrere Abhängigkeiten haben. Wenn Sie diese Komplexität nicht im Griff haben (und das Problem haben Sie ganz sicher wenn Sie nachträglich viel Funktionalität implementieren müssen, die nicht geplant war), wird Ihr System zwangsläufig irgendwann abstürzen.

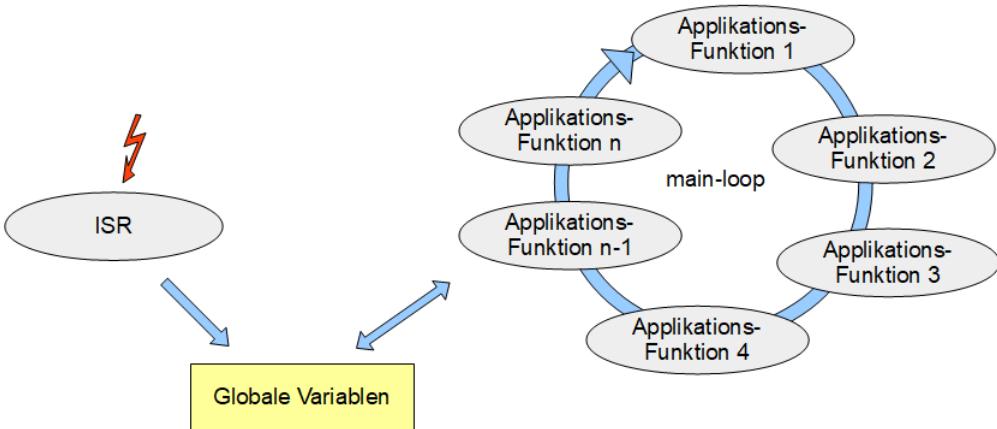


Abbildung 10.4.: Datenaustausch zwischen ISR und Applikationsfunktionen

Deshalb gilt: In einfachen Systemen mit zwei bis drei Interrupts können Sie durchaus ohne Betriebssystem arbeiten. Bei komplexeren Systemen und falls es die Hardware-Ressourcen erlauben (CPU, Speicher) ist ein Betriebssystem empfehlenswert. Mehr dazu erfahren Sie im Vertiefungsmodul „Echtzeit-Betriebssysteme“.

11. Projekte C / Assembler

11.1. Einleitung

Heute werden Microcontroller fast ausschliesslich in C oder C++ programmiert. Bei zeit- oder ressourcenkritischem Code wird aber oft ein Teil der Software in Assembler implementiert. Insbesondere gilt dies für folgende Funktionalität:

- Optimierter Zugriff auf die Hardware, speziell für Treiber
- Rechenintensive Algorithmen, um CPU und Co-Prozessoren (z.B. Multiply-Accumulate Einheit) optimal zu nutzen
- Häufig durchlaufene Funktionen, um die System-Performance zu erhöhen
- Zugriffe auf CPU-Register, die in C gar nicht adressiert werden können

In diesem Falle ist ein besonderes Augenmerk auf die Schnittstelle zwischen C und Assembler zu werfen. Diese Schnittstelle ist technologisch bedingt abhängig von der verwendeten CPU und vom Compiler. Insbesondere müssen folgende Punkte berücksichtigt werden:

- Aufruf von Assembler-Subroutinen aus C-Funktionen und umgekehrt
- Parameterübergabe und Rückgabewerte
- Verwendung von C-Variablen in Assembler und umgekehrt
- Inline-Assembler

Die in diesem Kapitel beschriebenen Mechanismen beziehen sich auf ARM Cortex-Mx CPUs sowie den GNU Compiler (gcc) und Assembler (as) für diese CPU-Familie.

11.2. Aufruf von Assembler-Subroutinen aus C

Assembler-Subroutinen können aus C-Code und C-Funktionen aus Assembler-Subroutinen aufgerufen werden. Dabei ist bei der Namensgebung zu beachten, dass die Namen identisch sind. Bemerkung: Bei einigen Entwicklungsumgebungen muss dem symbolischen Namen in Assembler zusätzlich ein „_“ vorangestellt werden. Dies ist jedoch bei der GNU-Toolchain nicht erforderlich.

Beispiel

Aufruf von Assembler-Subroutinen aus C-Code:

```
/* declare extern assembler subroutine */
extern int asmsub(int par1, int par2);

/* declare some variables */
int res;
int par1 = 3;
int par2 = 7;

/* C Function which is called from Assembler subroutine */
int cfunc(int par1, int par2){
    return(par1 + par2);
}
```

```
...
void main(void) {
    ...
    for (;;) {                                /* endless loop */
        res = asmsub(par1, par2);             /* call assembler subroutine */
    }
    ...
    return 0;
}
```

Listing 11.1: Aufruf der Subroutine asmsub aus dem C-Code

```
.global asmsub           @ gives symbol external linkage
.extern cfunc            @ specify that cfunc is defined in some other source code file
.thumb
.syntax unified
.text

asmsub:                 @ assembler subroutine
    PUSH    {lr}          @ save return address on stack
    BL     cfunc           @ call C-function
    POP    {pc}          @ return
```

Listing 11.2: Implementation von asmsub, Aufruf der C-Funktion cfunc() aus Assembler

11.3. Parameterübergabe an Assembler-Subroutinen

Im Kapitel 9.5 „Subroutinen“ wurde bereits beschrieben, wie der AAPCS die Übergabe von Parametern definiert. Daran müssen wir uns auch halten, wenn wir in Assembler und C gemischt programmieren.

Kurz zusammengefasst heisst dies:

- Die ersten vier Parameter werden in r0 bis r3 übergeben, unabhängig davon, ob wir eine C-Funktion oder eine Assembler-Subroutine aufrufen.
- Die restlichen Parameter werden auf dem Stack in umgekehrter Reihenfolge übergeben.
- Rückgabewerte werden im Register r0 zurückgegeben.

11.4. Vom C-Compiler verwendete Register

Ebenfalls von der AAPCS wird definiert, welche Register vom C-Compiler fix verwendet werden. Diese Register sind auch von Assembler-Programmierer*innen für diese Zwecke einzusetzen (siehe auch Kapitel 9.5):

- r13 / sp: Stack-Pointer
- r14 / lr: Link-Register
- r15 / pc: Program-Counter

11.5. Verwendete gemeinsame Variablen

Globale Variablen können in C und Assembler gemeinsam genutzt werden. Dabei ist wie schon bei der Namensgebung der Subroutinen zu beachten, dass die Namen von Variablen in Assembler und C identisch sind. Die gemeinsamen Variablen können entweder im C-Source oder in Assembler definiert werden.

Beispiel

Anlegen je einer globalen Variablen im C-Code (c_var) und im Assembler-Code (asm_var) sowie der Zugriff darauf.

```
int c_var = 0;           // reserve memory for word-variable
extern int asm_var;     // specified in assembler source
...
void ctest_var(void){
    asm_var = 0x22;      // modify assembler variable
    c_var = 0x55;        // modify C variable
}
```

Listing 11.3: Globale Variablen, C-Code

```
.global asm_var          @ give asm_var external linkage
.extern c_var            @ specified in C source
...
.data
asm_var:
.word 0                 @ global asm_var, initialized with 0
...
.text
asmtest_var:
PUSH {lr}                @ save return address on stack
LDR r0, =c_var           @ r0 holds address of variable c_var
MOV r1, #0x44             @ r1 holds value to write to c_var
STR r1, [r0]              @ c_var = 0x44
LDR r0, =asm_var          @ r0 holds address of variable asm_var
MOV r1, #0x11             @ r1 holds value to write to asm_var
STR r1, [r0]              @ asm_var = 0x11
POP {pc}                 @ return
```

Listing 11.4: Globale Variablen, Assembler-Code

11.6. Inline-Assembler

Mittels Inline-Assembler ist es möglich, Assembler-Zeilen direkt in den C-Code einzubinden. Dadurch können ARM-Instruktionen ausgeführt werden, die vom C-Compiler nicht unterstützt werden. Oder es können Register verändert werden, auf welche man aus dem C-Code nicht zugreifen kann; das sind alle CPU-Register sowie Coprozessor-Register die nicht über die Memory-Map ansprechbar sind.

Die Verwendung von Inline-Assembler ist manchmal hilfreich. Allerdings können ungewollte Seiteneffekte auftreten! Vorsicht ist geboten.

Um Inline-Assembler im C-Code für den GNU-Compiler gcc einzubinden wird das Statement „asm“ verwendet. Inline-Assembler ist nach ANSI-C nicht definiert und somit ist der zu verwendende Ausdruck abhängig von der Entwicklungsumgebung. Andere Compiler verwenden auch Ausdrücke wie ASM, __asm oder ASMLINE.

Die allgemeinste Form des asm-Statements ist für den GNU-Compiler wie folgt definiert:

```
asm volatile(code : output operand list : input operand list : clobber list);
```

Listing 11.5: Inline Assembler in der allgemeinsten Form

Beispiel

```
asm volatile("AND %0,%1,#0xFF" : "=r" (result) : "r" (value));
```

Listing 11.6: Inline Assembler Anweisung mit Output- und Input-Operanden

Das sieht auf den ersten Blick etwas kompliziert aus, wird aber gleich ausführlich besprochen. „volatile“ ist notwendig, damit der C-Compiler das asm-Statement nicht wegoptimiert. Das Statement selbst besteht aus vier Spalten:

1. code, beinhaltet die Assembler-Instruktion als String definiert. Im Beispiel: „AND %0, %1, #0xFF“

2. output operand list, mehrere Operanden werden durch Kommata getrennt. Im Beispiel: “=r” (result). Mit Hilfe dieser Liste kann innerhalb des asm-Statements auf C-Variablen zugegriffen werden. %0 aus dem Code bezieht sich auf das erste Element aus dieser Liste. “=r” bedeutet, dass der Compiler dafür irgend ein Register verwenden darf.
3. input operand list, mehrere Operanden werden durch Kommata getrennt. Im Beispiel: “r” (value). Da dieses der zweite Operand ist (Aufzählung output und input Operanden) wird im Beispiel mit %1 auf die Variable value zugegriffen. “r” bedeutet, dass der C-Compiler hier wieder irgend ein Register verwenden soll.
4. clobber list, teilt dem Compiler mit, welche Register, Speicher oder Statusregister durch den Assembler-Code verändert werden. Im Beispiel ist die clobber list leer.

Beispiel

```
asm volatile("MOV<ur0,ur0"); // NOP (No Operation)
```

Listing 11.7: Inline-Assembler Anweisung ohne Operanden, ohne clobber list

Beispiel

```
asm volatile(
    "MOV<ur0,ur0"    "\n\t"
    "MOV<ur0,ur0"
);
```

Listing 11.8: Inline Assembler Anweisung ohne Operanden, zwei Assembler-Instruktionen

Durch “\n\t“ wird das Assembler-Listing, welches der Compiler generiert, besser lesbar.

Wenn eine C-Variable als Ausgabeparameter verwendet werden soll, kann dies in der output operand list angegeben werden. Das nachfolgende Beispiel hat keine Eingabeparameter, die input operand list ist somit leer. Weiter zeigt das Beispiel sehr schön, wie auf ein Register zugegriffen wird (xPSR), was in reinem C-Code nicht möglich wäre:

Beispiel

```
asm volatile("MRS<%0,xPSR" : "=r" (status)); // load variable status with value of xPSR
```

Listing 11.9: Inline-Assembler Anweisung mit Zugriff auf das Status Register

Der Compiler wird aus diesem asm-Statement folgenden Code erzeugen:

```
MRS r2, xpsr
LDR r3, =status
STR r2, [r3]
```

Listing 11.10: Erzeugte Assembler-Instruktionen für obiges Listing

In der output operand list wurde für die Variable status ein “=r” angegeben. Es wird dem Compiler überlassen, welches Register er verwenden will, im Beispiel hat er sich für r2 und r3 entschieden.

Wenn die Assembler-Instruktion ein Register verändert, muss dies dem Compiler mitgeteilt werden. Der Compiler hat keine Möglichkeit, dies selber festzustellen, ist aber auf diese Information angewiesen, damit es keine Probleme mit den von ihm verwendeten Registern gibt. Die Angabe der geänderten Register erfolgt in der clobber list.

Beispiel

```
asm volatile("MOV<r1,ur0" : : : "r1"); // register r1 modified
```

Listing 11.11: Inline-Assembler Anweisung mit clobber list

Zum Abschluss nochmals einige Worte zum Beispiel eingangs des Kapitels.

```
asm volatile("AND<%0,%1,>#0xFF" : "=r" (result) : "r" (value));
```

Listing 11.12: Inline Assembler Anweisung mit Output- und Input-Operanden

Dieser Code hat als Eingangsparameter die Variable `value`. `value` wird mit dem Wert `#0xFF` AND-verknüpft, d.h. die höheren 3 Byte werden gelöscht. Das Ergebnis wird in der Variablen `result` abgelegt. Das Beispiel zeigt, wie der Assembler-Code erweitert werden und mit C-Variablen ergänzt werden kann. Das ist natürlich keine reine Assembler-Instruktion mehr, eher schon eine Mischung aus C und Assembler. Der vom C-Compiler generierter Code lautet:

```
LDR r3, =value
LDR r3, [r3]
AND r2, r3, #255
LDR r3, =result
STR r2, [r3]
```

Listing 11.13: Erzeugte Assembler-Instruktionen für obiges Listing

Da der Compiler diesen Code selber generiert hat, müssen die modifizierten Register `r2` und `r3` nicht in der clobber list angegeben werden.

Teil III.

Peripheriebausteine

12. Zugriff auf die Peripherie

12.1. Übersicht

Die Peripherie bildet die Schnittstelle zwischen der CPU und der Umgebung. Die Peripherie-Einheiten werden über Memory-Mapped Register am Adress-/Daten-Bus der CPU angeschlossen. Als Peripherie werden Komponenten bezeichnet, welche:

1. Daten von extern einlesen und an die CPU weiterleiten. Diese Daten können beispielsweise von Sensoren oder Kommunikations-Schnittstellen stammen.
2. Daten, welche die CPU erzeugt hat, an Aktoren, Kommunikations-Schnittstellen oder andere Interfaces ausgeben.

Die am häufigsten verwendeten Peripherie-Einheiten werden in den Folgekapiteln detailliert beschrieben. Es sind dies:

- Digitale Ein- und Ausgänge, GPIO, siehe Kapitel 13.
- EXTI-Controller für die Interrupt-Programmierung, siehe Kapitel 14
- UART, siehe Kapitel 15
- I2C-Bus, siehe Kapitel 16
- SPI-Bus, siehe Kapitel 17
- Timer, siehe Kapitel 18
- A/D-Wandler (Kapitel 19) und D/A-Wandler (Kapitel 20)

12.2. Zugriff auf Memory-Mapped Register

12.2.1. Einleitung

Bei der hardwarenahen Programmierung besteht das Bedürfnis, Adressen im Memory-Map anzusprechen (Hardware-Register des Microcontrollers oder der Peripherie). Dazu stehen folgende Möglichkeiten zur Verfügung:

- #define nach ANSI-C
- const Pointer
- Absolute Sections
- herstellerspezifische Schlüsselwörter

12.2.2. #define nach ANSI-C

Mit dieser Variante wird ein Pointer sowie seine absolute Adresse als Makro definiert. Es ist eine verbreitete Möglichkeit, in der Programmiersprache C auf eine fixe Adresse zuzugreifen. Dabei geht man folgendermassen vor:

1. Das Makro wird nach folgendem Schema definiert:

```
// fuer eine 8-bit Adresse 0x20010000
#define reg_8    *((volatile unsigned char *) 0x20010000)

// fuer eine 16-bit Adresse 0x20010004
#define reg_16   *((volatile unsigned short *) 0x20010004)

// fuer eine 32-bit Adresse 0x20010008
#define reg_32   *((volatile unsigned long *) 0x20010008)
```

Listing 12.1: Definition der Makros für den Zugriff auf absolute Adressen

2. Der Zugriff auf die Adresse im Code lautet wie folgt:

```
reg_8 = 0x12;
reg_16 = 0x1234;
reg_32 = 0x12345678;
```

Listing 12.2: Zugriff auf die absoluten Adressen

12.2.3. Const Pointer

Anstelle der Makros `#define` können auch `const` Pointer verwendet werden. Dabei geht man folgendermassen vor:

1. Die Variable wird nach folgendem Schema definiert:

```
int* const cp_32 = (int *)0x2001000C;
```

Listing 12.3: Definition der `const` Pointer für den Zugriff auf absolute Adressen

Beachten Sie die Position des Schlüsselwortes „`const`“. Dieses gibt an, dass der Pointer selbst nicht geändert werden kann, wohl aber der Wert, auf den er zeigt.

2. Der Zugriff auf die Variable im Code lautet wie folgt:

```
*cp_32 = 0x12345678;
```

Listing 12.4: Zugriff auf die absoluten Adressen

12.2.4. Absolute Sections

Entwicklungsumgebungen bieten die Möglichkeit, eigene Sections zu definieren und diesen absolute Adressen zuzuweisen. Beim GNU-Compiler wird dazu das Schlüsselwort „`__attribute__`“ verwendet, mit welchem spezifische Attribute für Variablen definiert werden können. Nach dem Schlüsselwort „`__attribute__`“ wird das Attribut selbst definiert, wobei dieses Attribut in einer doppelten Klammer angegeben wird. Um eine Section anzugeben wird das Attribut „`section`“ verwendet.

Beispiel

Es soll ein Array „sevenSeg[]“ definiert werden, welches auf dem Leguan-Board den vier 7-Segment Anzeigen entspricht. Die 7-Segment-Anzeigen werden ab Adresse 0xC8000010 angesprochen.

```
enum {SEG0=0, SEG1, SEG2, SEG3, SEVEN_HEX, SEVEN_BCD, SEVEN_DP, SEVEN_NONE};
uint16_t sevenSeg [8] __attribute__ ((section ("._SEVENSEG")));
...
sevenSeg[SEVEN_HEX] = 0x12AB;
```

Listing 12.5: Definition einer Variablen in einer eigenen Section

In der ersten C-Code-Zeile werden die Index des Arrays definiert (siehe auch Memory-Map für die 7-Segment Anzeige). Die zweite Code Zeile legt ein Array von 16-Bit Werten fest, welches den Adressen der 7-Segment Anzeige entspricht. Mit `__attribute__ ((section ("._SEVENSEG")))` wird angegeben, dass sich das Array „sevenSeg“ in der Section „._SEVENSEG“ befindet. In der nächsten Zeile wird mit „sevenSeg[SEVEN_HEX] = 0x12AB“ ein Bitmuster auf die 7-Segment Anzeige geschrieben.

Im Linkerskript „stm32f743zitx.flash.ld“ muss nun die Section .SEVENSEG noch definiert werden. Dazu sind folgende zwei Einträge notwendig:

- Der Memorybereich der Hardware-Peripherie muss spezifiziert werden (im Beispiel „LEGUAN_PERIPH“).

```
/* Specify the memory areas */
MEMORY
{
    FLASH (rx)      : ORIGIN = 0x08000000, LENGTH = 2048K
    RAM (xrw)       : ORIGIN = 0x20000000, LENGTH = 128K
    RAM_D1 (xrw)    : ORIGIN = 0x24000000, LENGTH = 512K
    RAM_D2 (xrw)    : ORIGIN = 0x30000000, LENGTH = 288K
    RAM_D3 (xrw)    : ORIGIN = 0x38000000, LENGTH = 64K
    ITCMRAM (xrw)   : ORIGIN = 0x00000000, LENGTH = 64K
    LEGUAN_DRAM (xrw) : ORIGIN = 0x60000000, LENGTH = 32k
    LEGUAN_PERIPH (xrw) : ORIGIN = 0xC8000000, LENGTH = 0x800
}
```

Listing 12.6: Definition

- Die Section „LEGUAN_PERIPH“ muss spezifiziert werden.

```
.LEGUAN_PERIPH :
{
    . = ALIGN(4);
    _leguan_periph = .;           /* create a global symbol at leguan periphery start */
    *(.FPGA_RESET)              /* FPGA reset address */
    . += 0x10;                   /* gap */
    *(.SEVENSEG)                /* 7 segment display */
    . += 0x10;                   /* gap */
    ...
} >LEGUAN_PERIPH
```

Listing 12.7: Definition

Der Eintrag „*(.SEVENSEG)“ legt fest, dass sich die 7-Segment Anzeige ab der Adresse 0xC8000010 ansprechen lässt. Diese Adresse berechnet sich aus der Basisadresse der Leguan-Peripherie (0xC8000000) sowie dem Offset von 0x10, eingetragen in der Zeile „. += 0x10;“

12.2.5. Spezielle Schlüsselwörter

Einige Compilerhersteller definieren eigene Schlüsselwörter, um Variablen auf eine definierte Adresse zu legen. Diese Schlüsselwörter lauten oft „_at“ oder „_at“.

Beispiel

```
volatile unsigned short led _at(0x80000000);
```

Listing 12.8: Absolute Adressierung mit Hilfe von Schlüsselwörtern

Für den GNU-Compiler gibt es leider keine solchen Schlüsselwörter.

12.3. Libraries

Der Zugriff auf die Hardware erfolgt in der Programmierung entweder direkt auf die Hardware-Register oder mit Hilfe von Library-Funktionen. Der direkte Zugriff auf die Hardware-Register hat den Vorteil, dass er sehr schnell ist, bedingt aber viel Know-How über die Hardware-Architektur und muss bei der Portierung der Software auf einen anderen Prozessor angepasst werden. Deshalb stellen viele CPU- und Board-Hersteller Libraries für den Zugriff auf die Hardware zur Verfügung. Dadurch wird die Entwicklungszeit für die Software reduziert und der Code ist portabler, andererseits wird der Code grösser und die Ausführungszeiten werden länger.

Abbildung 12.1 zeigt die verschiedenen Schichten (Layerung) von Hardware, Libraries und Applikation.

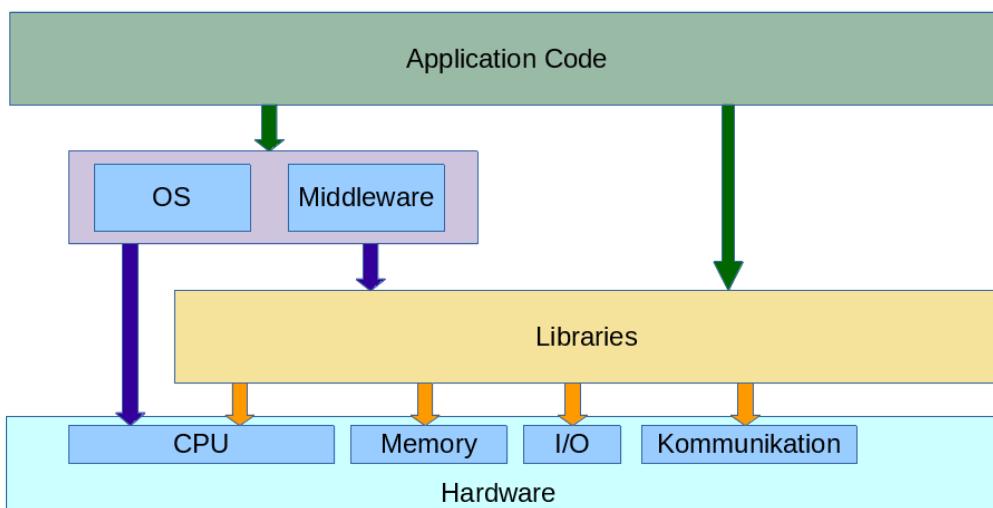


Abbildung 12.1.: Library Layerung

12.4. CMSIS

12.4.1. Die Vorteile von CMSIS

CMSIS (Cortex Microcontroller Software Interface Standard) [6] definiert ein standardisiertes Software-Interface für Cortex-Mx Prozessoren. Dieser Standard wurde von ARM und verschiedenen Chip- und Tool-Herstellern definiert, um die Interoperabilität verschiedener Softwarekomponenten zu gewährleisten.

Die Ziele von CMSIS sind insbesondere:

- Software-Kompatibilität: Die API-Funktionen des CMSIS ermöglichen die Zusammenarbeit verschiedener Software-Komponenten unterschiedlicher Hersteller.
- Wiederverwendbarkeit der Software: Die Software kann wesentlich einfacher auf andere Cortex-Mx Prozessoren portiert werden, was den Entwicklungsaufwand erheblich reduziert.
- Einfacher Lernprozess: Verschiedenen Prozessor-Features können mit Hilfe von C-Funktionen verwendet werden, ohne Details zu kennen. Das erarbeitete Know-How kann einfach auf andere Cortex-Mx Prozessoren angewendet werden.

- Unabhängigkeit von der Toolchain: Die CMSIS API-Funktionen sind unabhängig von der Toolchain und ermöglichen dadurch eine freie Wahl der besten Entwicklungsumgebung für das jeweilige Projekt.

Aus Sicht der Software werden mit CMSIS folgende Bereiche standardisiert:

- Standardisierung der Peripherie eines Prozessors. Insbesondere umfasst diese Standardisierung den NVIC (Nested Vector Interrupt Controller), den SysTick (System Tick), optional eine MPU (Memory Protection Unit) und verschiedene Register des SCB (System Control Block).
- Standardisierung der API-Funktionen für den Zugriff auf die Peripherie. Diese API-Funktionen erhöhen die Portabilität von Code.
- Diverse standardisierte API-Funktionen für den Zugriff auf spezielle Instruktionen wie WFI (Wait for Interrupt).
- Eine Standardisierung der Namen für Exception Handler.
- Standardisierte Namen für die Initialisierung des Systems, z.B. „SystemInit()“.
- Standardisierte Namen für wichtige Systemvariablen wie „SystemCoreClock“.

12.4.2. Die Layerung von CMSIS

Die Software eines Systems besteht aus verschiedenen Layern, welche in Abbildung 12.2 dargestellt sind. CMSIS setzt direkt auf der Hardware auf. Die Applikation verwendet CMSIS-Funktionen und greift in der Regel nicht direkt auf die Hardware zu. Optional kann auch Software von Drittanbietern eingekauft und verwendet werden.

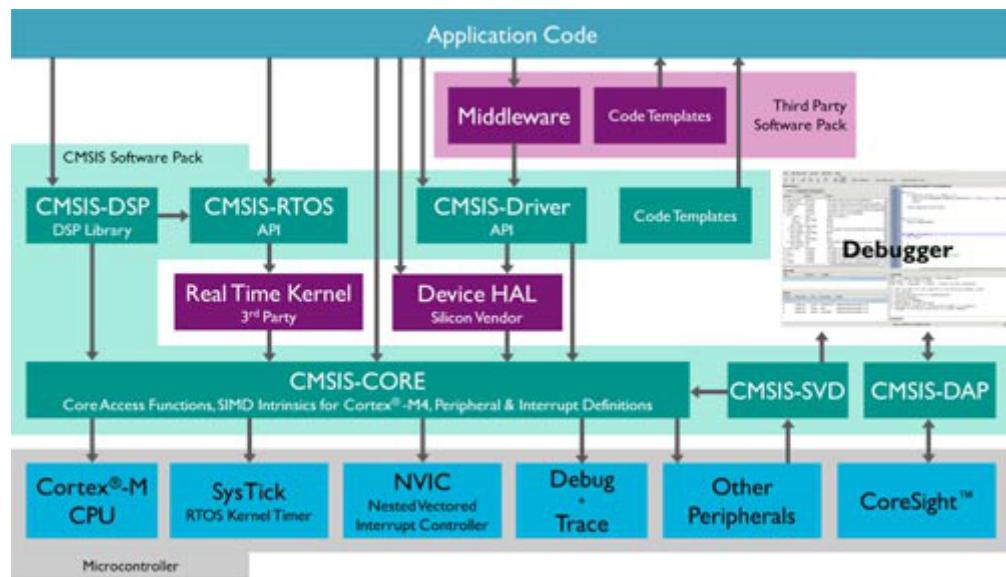


Abbildung 12.2.: CMSIS Layerung. Quelle: [6]

CMSIS besteht aus verschiedenen Modulen. Dies sind:

- **CMSIS-CORE**: Bietet ein Standard-Interface für den Zugriff auf Core und Peripherie von Cortex-Mx Prozessoren und ist unabhängig vom Chiphersteller oder von der Toolchain.
- **CMSIS-Driver**: Definiert ein generisches Interface für die Peripherie verschiedener Chiphersteller und findet Verwendung als Schnittstelle zur Applikation sowie zu Middleware wie Kommunikations-Stacks, Dateisysteme oder GUI.
- **CMSIS-DSP**: Diese DSP-Library bietet eine Vielzahl an API-Funktionen für Floatingpoint-Operationen. Die DSP-Library für den Cortex-M7 verwendet die Funktionalität des DSP Co-Prozessors.

- **CMSIS-RTOS API:** Bietet ein Standard-Interface für Real Time Operating Systems. Für viele RTOS gibt es Portierungen auf CMSIS-RTOS, so dass Applikationen ohne (grossen) Aufwand auf ein anderes RTOS portiert werden können.
- **CMSIS-SVD:** System View Description, beschreibt die Peripherie des Prozessors in Form einer XML-Datei. Sie dient als Hilfe für das Debugging, um den Zustand der Peripherie im Debugger darzustellen.
- **CMSIS-DAP:** Debug Access Port ist ein Referenz-Design für ein Debug Interface, welches USB und JTAG-/Serial unterstützt.

12.4.3. Die Einbindung von CMSIS in ein Projekt

Sobald in einem Projekt die Device Driver Library eines Chipsetstellers verwendet wird, werden auch die Dateien der CMSIS-Library eingebunden. Abbildung 12.3 zeigt, welche Dateien in ein Projekt eingebunden werden müssen, um die Libraries zu verwenden. In Klammern unterhalb der entsprechenden Dateien werden jeweils die Dateinamen angegeben, welche für das Leguan-Board eingebunden werden.

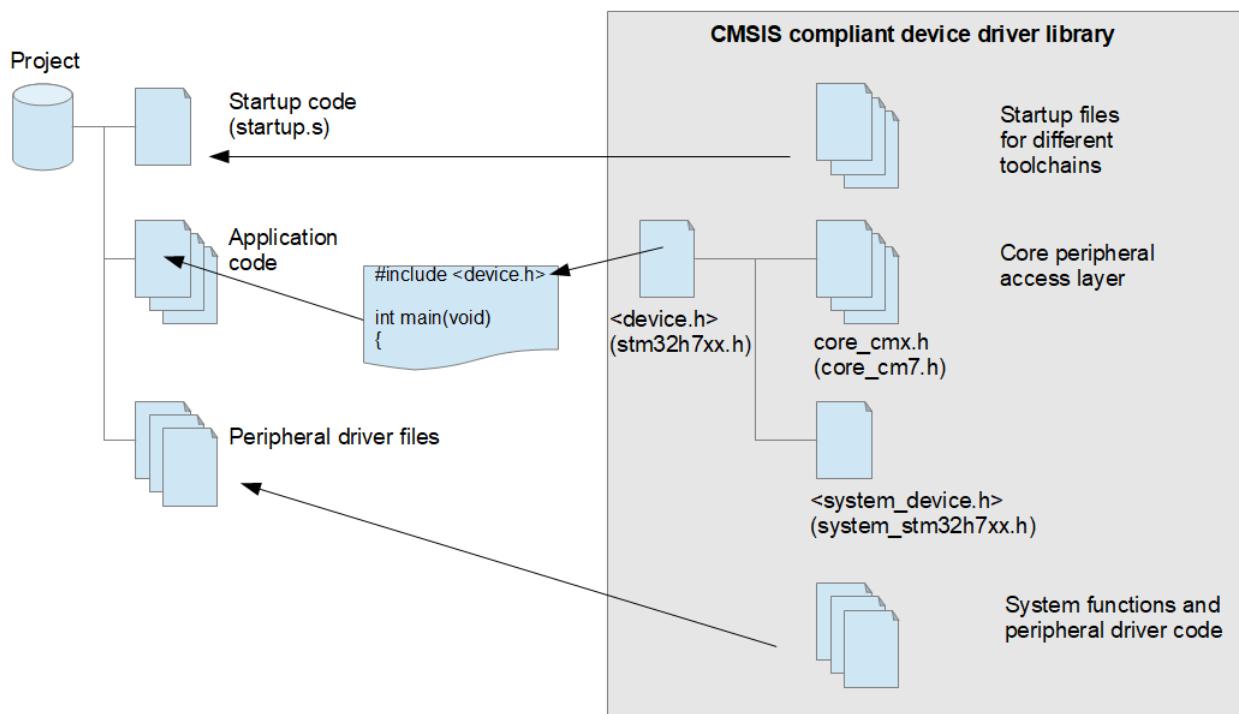


Abbildung 12.3.: Einbindung der Device Driver Library und der CMSIS Library in ein Projekt. Quelle: [13]

Folgende Source-Dateien sind in ein Projekt einzubinden:

- Die Startup-Datei, abhängig vom verwendeten Prozessor und der Toolchain. Sie wird beim Anlegen eines neuen C-Projektes von der IDE automatisch in das Projekt integriert.
- Prozessorspezifischer Initialisierungscode (z.B. system_<device>.c). Wird von der IDE automatisch über eine Bibliothek eingebunden.
- Chipspezifische Source-Dateien. Wird von der IDE automatisch über eine Bibliothek eingebunden.

Folgende Header-Dateien sind in ein Projekt einzubinden:

- Chipspezifische Header-Datei. Wird beim Anlegen eines neuen C-Projektes von der IDE automatisch in der Datei main.c eingefügt.

```
/*----- Header-Files -----*/
#include <stm32h7xx.h> /* Processor STM32h7xx */
```

Listing 12.9: Einbinden der Prozessor-Headerdatei

- Eine zusätzliche chipspezifische Header-Datei (system_<device>.h respektive system_stm32h7xx.h) mit Funktionen für die Initialisierung. Diese wird von der Datei <device>.h automatisch eingebunden.
- Eine zusätzliche Datei mit Funktionalitäten für den Prozessor Core, typischerweise mit Namen wie core_cm7.h. Auch diese Datei wird von <device>.h automatisch eingebunden.

12.4.4. Beispiel

In CMSIS wird beschrieben, wie symbolische Namen definiert werden:

1. Grossbuchstaben für Core-Register, Peripherie-Register oder CPU-Instruktionen.
2. „Camel-Case“ (Mix Gross- und Kleinbuchstaben) für Funktionen.
3. Prefix für Funktionen mit Angabe des Moduls (<name>_Function).

Weiter wird in CMSIS empfohlen, Doxygen für die Kommentare zu verwenden.

```
/** 
 * @brief Enable Interrupt in NVIC Interrupt Controller
 *
 * @param IRQn_Type IRQn specifies the interrupt number
 * @return none
 *
 * Enable a device specific interrupt in the NVIC interrupt controller.
 * The interrupt number cannot be a negative value.
 */
static __INLINE void NVIC_EnableIRQ(IRQn_Type IRQn)
{
    ...
}
```

Listing 12.10: Beispiel CMSIS Kommentare und Namensgebung

12.5. Leguan BSP

Zur Programmierung des Leguan-Boards werden diverse Libraries zur Verfügung gestellt. Dieses erleichtern die Programmierung der Peripherie und sind hierarchisch aufgebaut:

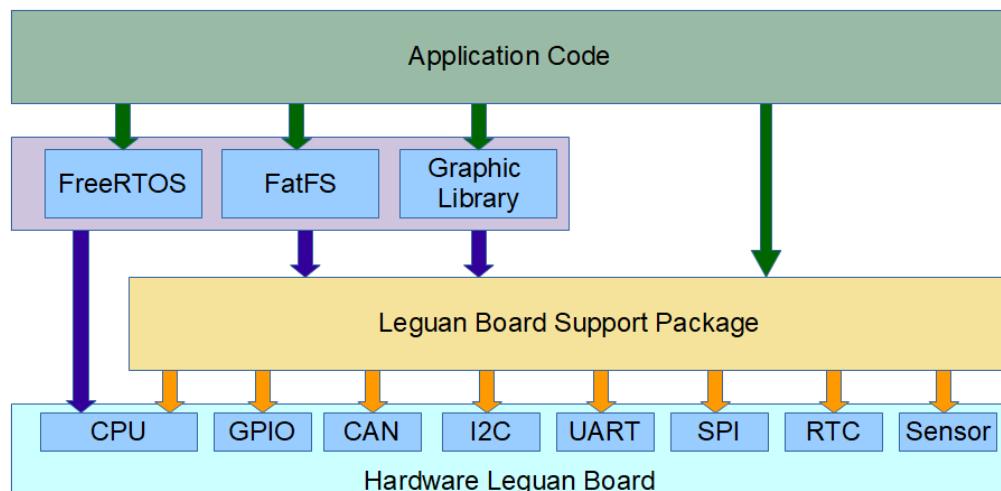


Abbildung 12.4.: Layerung Leguan Libraries

12.5.1. Board Support Package

Funktionsnamen des Leguan BSP (Board Support Package) beginnen immer mit „LEGUAN_“. Mit ihnen kann die Hardware des Leguan-Boards angesteuert werden. Des weiteren übernimmt das BSP auch einen Teil der Initialisierung im Startup Prozess, so werden beispielsweise alle RCC Clocks der GPIOs eingeschaltet, da diese für den Adress- und Datenbus gebraucht werden.

Die Funktionen des BSP sind auf der Leguan-Webseite beschrieben.

12.5.2. FatFs

Das FatFs [7] stellt einfache Funktionen zur Verfügung, um die SD Karte anzusteuern und Dateien in einem FAT Filesystem zu erstellen, zu löschen und zu bearbeiten.

Das FatFs wurde von ChaN entwickelt. Es enthält keine Low Level Treiber für die Kommunikation mit dem Speicher, sondern nur das Fat Filesystem. Der Treiber zur Ansteuerung des Speichers muss selber programmiert werden. ST Microelectronics hat einen Treiber für ihre Prozessoren erstellt, welche über SDIO mit der SD Karte kommuniziert. Dieser Treiber wurde für das Leguan-Board angepasst und ins BSP integriert.

Die Funktionen des FatFs sind nicht kompatibel mit der stdlib, so heisst zum Beispiel die Funktion zum Öffnen einer Datei nicht fopen() sondern f_open(). Auch die Parameter stimmen nicht ganz mit der stdlib überein: Es werden eigene Datenstrukturen definiert und nicht die Datenstrukturen aus der stdlib verwendet.

Die FatFs Library ist Open-Source und darf für eigene Projekte genutzt werden. Das FAT Filesystem von Microsoft ist jedoch lizenpflichtig und muss für den kommerziellen Einsatz von Microsoft lizenziert werden.

Beispiel FatFs

Nachfolgendes Listing zeigt, wie mit Hilfe der FatFs-Library aus einer Datei auf der SD-Karte gelesen werden kann:

```
FIL f;
char buff[20]; /* Buffer to store string */
UINT totalread = 0; /* Pointer for next read position */
UINT tmpread; /* number of read characters */

/* open the read.txt file to read the next string */
if (!f_open(&f, "read.txt", FA_OPEN_ALWAYS | FA_READ)) {
    /* set the cursor next position to read */
    f_lseek(&f, totalread);
    f_read(&f, buff, sizeof(buff), &tmpread);
    f_close(&f);
    totalread += tmpread;
}
```

Listing 12.11: Beispiel FatFs Library

12.5.3. HAL Driver Library

Zur Programmierung des Leguan Boards kann anstelle des Leguan BSPs auch die HAL Driver Library verwendet werden. HAL bedeutet im Allgemeinen “Hardware Abstraction Layer“, dient also zur Abstrahierung der Hardware für die oberen Software-Layer. Dadurch wird Code portabel, indem bei einem Wechsel der Hardware nur der HAL Driver angepasst werden muss, nicht aber der Applikationscode. ST Microelectronics hat für ihre Cortex-Mx Microcontroller eine HAL Driver Library entwickelt, welche sehr universell ist. Sie kann für alle Boards mit einem Cortex-Mx Microcontroller von ST Microelectronics verwendet werden. Somit ist die HAL Driver Library eine gute Wahl, wenn Sie Programme für andere Boards wie ein Nucleo-Board oder ein Discovery-Board entwickeln. Die HAL Driver Library bietet Funktionen zur Ansteuerung der CPU inklusive deren Peripherie. Sie bietet aber keine Middleware zur Ansteuerung von Displays usw.

Von der HAL Driver Library wird die On-Chip Peripherie der Cortex-Mx Microcontroller unterstützt:

- ADC / DAC
- Timer
- GPIO
- Watchdog
- DMA
- SPI
- I2C
- USART
- CAN
- ...

Die Liste der Hardware, welche mit der HAL Driver Library angesteuert werden kann, ist sehr lang. Ausführliche Angaben sind im Datenblatt des Prozessors oder in den Header Dateien der Library zu finden. Alle Funktionsnamen, welche mit "HAL_" beginnen, gehören zur HAL Driver Library.

Im Namen der Library-Funktionen wird ebenfalls die anzusteuernde Hardware angegeben. So heissen beispielsweise die Funktionen zur Ansteuerung der GPIOs immer „HAL_GPIO_*“ (Beispiel für die Initialisierung: HAL_GPIO_Init). Zu beachten ist, dass vor dem Gebrauch einer Komponente der Clock auf diese Peripherie weitergeleitet werden muss. Dies wird mit Hilfe des Moduls RCC gemacht. Beispiel: Soll auf den GPIO Port A zugegriffen werden, muss zuerst der RCC für GPIO Port A mit der Funktion __HAL_RCC_GPIOA_CLK_ENABLE() aktiviert werden. Danach kann die Konfiguration des Moduls vorgenommen und das Modul eingeschaltet werden.

Beispiel für die Initialisierung des GPIO Port A Pin 0:

```
GPIO_InitTypeDef GPIO_InitStruct;

/* GPIO PortA Clock Enable */
__HAL_RCC_GPIOA_CLK_ENABLE();

/*Configure GPIO pin Output Level */
HAL_GPIO_WritePin(OUTO_GPIO_Port, OUTO_Pin, GPIO_PIN_RESET);

/*Configure GPIO pin : OUTO_Pin */
GPIO_InitStruct.Pin = OUTO_Pin;
GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
GPIO_InitStruct.Pull = GPIO_NOPULL;
GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_VERY_HIGH;
HAL_GPIO_Init(OUTO_GPIO_Port, &GPIO_InitStruct);
```

Listing 12.12: Beispiel HAL Driver Library, Initialisierung GPIO

Soll der GPIO als Alternate Function betrieben werden, muss der GPIO-Mode in GPIO_InitStruct.Mode auf GPIO_AFxx gesetzt werden. Beispiel:

```
GPIO_InitStruct.Mode = GPIO_AF8_UART4;
```

Listing 12.13: Beispiel HAL Driver Library, Zugriff auf Alternate Function Register

Welche GPIOs welche Alternate Function unterstützen, kann im Datenblatt des Prozessors [12] nachgeschlagen werden. Anhand dessen kann auch geprüft werden, welche Funktion innerhalb des Moduls ein Pin übernimmt.

Obiges Beispiel zeigt, wie ein GPIO initialisiert wird. Damit die Alternate Function funktioniert, muss auch die UART initialisiert werden. Zudem müssen bei Verwendung von Interrupts auch der NVIC und im Falle der Verwendung der DMA auch diese initialisiert werden.

Es können auch Peripherie-Komponenten initialisiert werden, welche keinem GPIO zugewiesen werden. Dies ist zum Beispiel bei den Timern der Fall. Diese können entweder als reine Software Timer oder für Capture Compare Einheiten, usw. verwendet werden.

Details zur HAL Driver Library von ST Microelectronics sind in [10] zu finden.

12.5.4. CubeMX

Viele Microcontroller-Hersteller bieten Tools an, um die Entwicklungzeit für die Software zu reduzieren, insbesondere für die Initialisierung der Peripherie. Die Tools unterstützen Entwickler*innen darin, die einzelnen Register der Microcontroller mit grafischen Hilfsmitteln zu konfigurieren, ohne sich um Details zu kümmern.

Ein solches Tool ist CubeMX von STMicroelectronics. Das Tool läuft entweder eigenständig oder in STM32CubeIDE integriert. Mit Hilfe von CubeMX können beispielsweise einzelne GPIO des Microcontrollers grafisch konfiguriert werden. CubeMX erzeugt anschliessend den für die Initialisierung notwendigen C-Code. Sie können entweder alle Pins, Timer usw. Ihres Microcontrollers bei Projektstart mit CubeMX konfigurieren und anschliessend ein Projekt erstellen lassen, oder Sie können den von CubeMX erzeugten Code bedarfsweise (einzelne Funktionen) kopieren und in Ihrem Projekt integrieren.

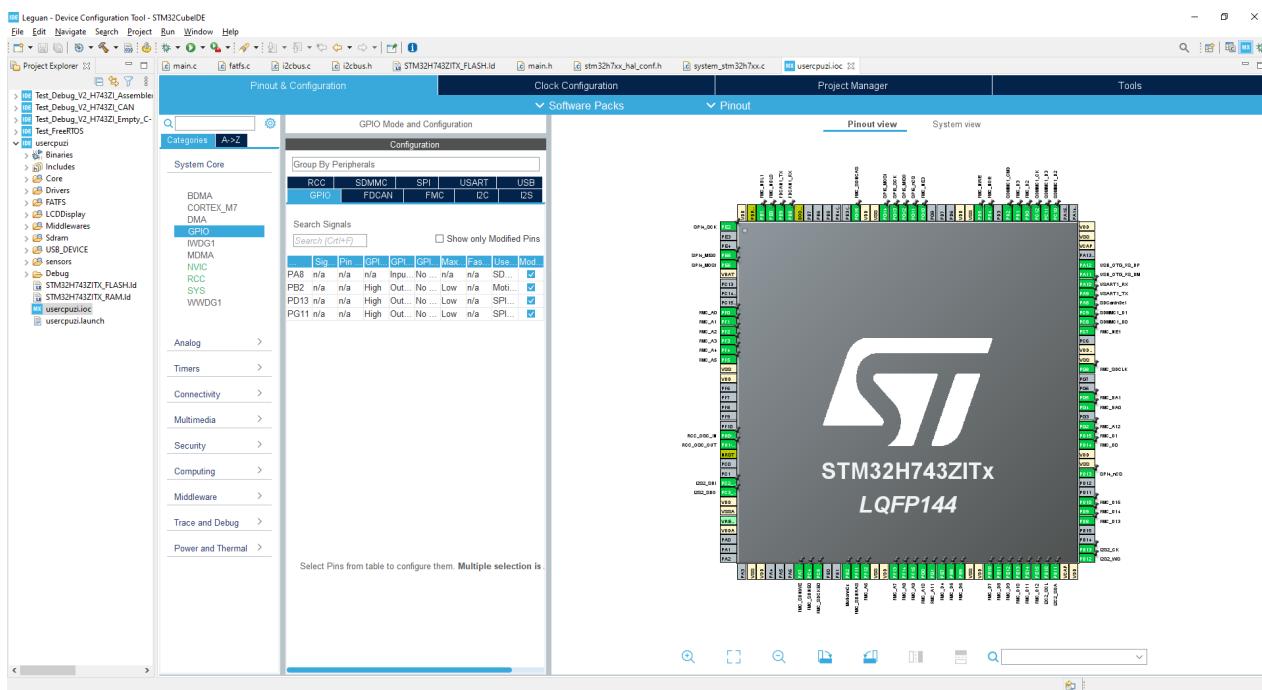


Abbildung 12.5.: Konfiguration des Microcontrollers mit CubeMX

13. GPIO

13.1. Anwendungen

GPIOs (General-Purpose Input Output) sind multifunktionale Ein- und Ausgänge, welche für die Ein- und Ausgabe von analogen und digitalen Signalen verwendet werden. Es gibt zwei unterschiedliche Ansätze um I/Os bereitzustellen:

1. Die Verwendung von Microcontroller-Pins (GPIOs). Je nach verwendeter Microcontroller-Familie ist die Zahl der zur Verfügung stehenden GPIOs unterschiedlich. In der Software werden diese Pins über ein GPIO-Register angesprochen (memory-mapped).
2. Die Verwendung zusätzlicher Peripherie-ICs (FPGA, PIO, ...), welche verschiedene I/Os bereitstellen und vom Microcontroller über den Adress-/Datenbus angesprochen werden.

Typische Einsatzgebiete für GPIO Inputs sind das Einlesen von Schaltern, Tastern, analogen Eingängen usw. Aber auch die Empfangssignale von Kommunikationsschnittstellen wie UART, I2C, SPI usw. werden über GPIOs eingelesen, erhalten dann aber eine spezielle Funktionalität (Alternate Function). Digitale Inputs werden häufig noch gefiltert (RC-Glieder und Schmitt-Trigger) und EMV-mässig gegen Überspannungen geschützt (beispielsweise mit Transzorb-Dioden, Varistoren, Gasableitern).

Typische Einsatzgebiete für GPIO Outputs sind die Ansteuerung von Anzeigeelementen (Lampen, LEDs), Relais, Motoren, Ventilen, analoge Ausgänge aber auch Ausgänge von Kommunikationsschnittstellen usw. Digitale Outputs werden teilweise noch über Verstärkerstufen geführt, damit sie die geforderten Ströme liefern können.

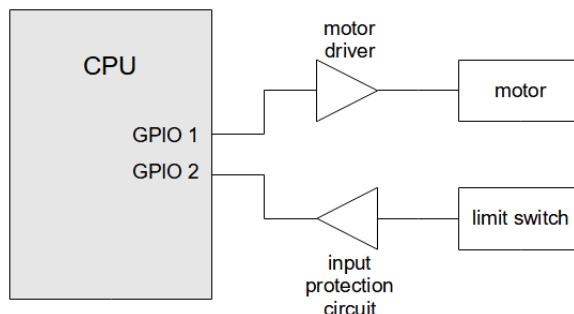


Abbildung 13.1.: Beispiel Verwendung von GPIOs

GPIOs werden auf einem Microcontroller in Gruppen zusammengefasst. Bei ARM Cortex-Mx sind das Ports mit jeweils 16 Pins, beispielsweise Port A mit PA0 (bit0) bis PA15 (bit 15). Auf dem STM32H743 hat es je nach Pinout bis zu 11 Ports mit je 16 bit (Port A bis Port K).

13.2. Aufbau

Der Aufbau eines GPIO Port Pins auf dem STM32H7xx ist in Abbildung 13.2 dargestellt:

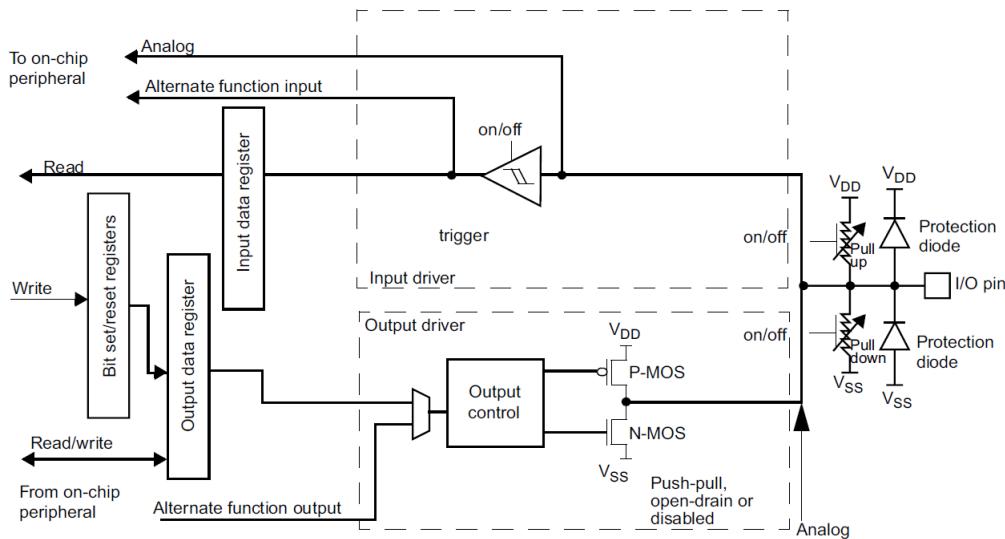


Abbildung 13.2.: Aufbau eines GPIO Port Pins, [11]

Die Software kann über Memory-Mapped Hardware-Register das GPIO konfigurieren, die Ausgänge setzen oder die Eingänge einlesen. Die einzelnen Register werden in [11] ausführlich beschrieben und haben folgende Namen / Funktionalität:

- **Port Mode Register**
Ein 32-Bit Register, welches für jedes Pin den Mode definiert (Input, Output, Alternate Function oder Analog). Für jedes Pin sind dafür zwei Bits im Port Mode Register reserviert.
- **Port Output Type Register**
Die unteren 16 Bit dieses 32-Bit Registers definieren für jedes Pin eines Ports den Ausgangstreiber (Push-Pull oder Open-Drain).
- **Port Output Speed Register**
Ein 32-Bit Register, welches für jedes Pin die mögliche Geschwindigkeit definiert (Low, Medium, High, Very High). Für jedes Pin sind dafür zwei Bits im Port Output Speed Register reserviert. Je höher die Geschwindigkeit gewählt wird, desto grösser ist auch der Stromverbrauch.
- **Port pull-up/pull-down Register**
Ein 32-Bit Register, welches für jedes Pin die pull-up / pull-down Widerstände definiert (no, pull-up, pull-down, reserviert). Für jedes Pin sind dafür zwei Bits im Port pull-up/pull-down Register reserviert.
- **Port Output Data Register**
Ein 32-Bit Register, über welches die Daten an den Ausgang geschrieben werden können. Die unteren 16 Bit des Registers werden 1:1 auf die einzelnen Pins des Ports gemappt (Pin 0 bis Pin 15). Die oberen 16 Bit des Registers werden nicht gebraucht.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ODR15	ODR14	ODR13	ODR12	ODR11	ODR10	ODR9	ODR8	ODR7	ODR6	ODR5	ODR4	ODR3	ODR2	ODR1	ODR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:16 Reserved, must be kept at reset value.

Bits 15:0 **ODR[15:0]**: Port output data I/O pin y (y = 15 to 0)

These bits can be read and written by software.

Note: For atomic bit set/reset, the ODR bits can be individually set and/or reset by writing to the GPIOx_BSRR register (x = A..F).

Abbildung 13.3.: Port Output Data Register, [11]

- Port Bit set/reset Register

Um ein einzelnes Bit eines 16-Bit Ports zu schreiben (setzen oder löschen) könnte der Wert des Ports zurückgelesen, das entsprechende Bit modifiziert und der ganze Port wieder geschrieben werden. Da dies umständlich ist, gibt es das set/reset Register. Dies ist ein 32-Bit Register, welches für jedes Pin eines Ports ein Bit-Set Bit sowie ein Bit-Reset Bit hat. Das Schreiben einer „1“ an die entsprechende Bitposition führt dazu, dass das Bit gesetzt (BSx) oder gelöscht (BRx) wird. Ist eine „0“ an der entsprechenden Bitposition, wird das Bit nicht geändert.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
BR15	BR14	BR13	BR12	BR11	BR10	BR9	BR8	BR7	BR6	BR5	BR4	BR3	BR2	BR1	BR0
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BS15	BS14	BS13	BS12	BS11	BS10	BS9	BS8	BS7	BS6	BS5	BS4	BS3	BS2	BS1	BS0
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w

Bits 31:16 **BR[15:0]**: Port x reset I/O pin y (y = 15 to 0)

These bits are write-only. A read to these bits returns the value 0x0000.

0: No action on the corresponding ODRx bit

1: Resets the corresponding ODRx bit

Note: If both BSx and BRx are set, BSx has priority.

Bits 15:0 **BS[15:0]**: Port x set I/O pin y (y = 15 to 0)

These bits are write-only. A read to these bits returns the value 0x0000.

0: No action on the corresponding ODRx bit

1: Sets the corresponding ODRx bit

Abbildung 13.4.: Port Bit set/reset Register, [11]

- Port Input Data Register

Ein 32-Bit Register, über welches die Daten des Ports eingelesen werden können. Die unteren 16 Bit des Registers enthalten die Werte der 16 Pins eines Ports.

13.3. Programmierung

Für die Programmierung der GPIO gibt es zwei Ansätze: Entweder Sie greifen direkt auf die Hardware-Register zu, oder Sie verwenden Library-Funktionen. Der Zugriff über die Hardware-Register erfordert Wissen über den Aufbau eines Ports sowie über die zu programmierenden Register, ist dafür aber sehr schnell. Die Verwendung von Library-Funktionen kapselt den Zugriff und erfordert von den Programmierer*innen kein vertieftes Wissen über den Hardware-Aufbau, dafür ist der Zugriff langsamer und es braucht mehr Programmspeicher.

Ein Beispiel für den Zugriff auf die einzelnen Register in C ist in folgendem Listing dargestellt:

```
/**  
 * GPIOB initialization for decimal point 7-segment display Leguan, PB15, low active  
 * for details see STM32H7xx reference manual  
 * Port:  GPIOB  
 * Pin:    15  
 * Mode:   Output push-pull  
 * Speed:  very high  
 * PuPd:   No PU/PD  
 */  
  
/* GPIO port mode register, value 01 for GPIOB pin 15, general purpose output */  
GPIOB->MODER &= ~(1<<31);      // clear bit 31  
GPIOB->MODER |= (1<<30);       // set bit 30  
  
/* GPIO port output type register, type 0 (push-pull) */  
GPIOB->OTYPER &= ~(1<<15);     // clear bit 15 for push-pull  
  
/* GPIO port output speed register, value 11 for very high */  
GPIOB->OSPEEDR |= 0xC0000000; // set bit 30 and 31  
  
/* GPIO port pull-up/pull-down register, no pull-up/pull-down */  
GPIOB->PUPDR &= ~0xC0000000; // clear bit 30 and 31  
  
/* clear and set pin */  
GPIOB->ODR &= ~(1 << 15);      // clear bit 15, set decimal point  
GPIOB->ODR |= (1 << 15);       // set bit 15, clear decimal point
```

Listing 13.1: Beispiel für die Initialisierung von GPIO Port B

13.4. Ausgangstreiber

Die Ausgangssignale eines ICs werden über Treiberstufen auf die Pins ausgegeben. Je nach Art der Ausgänge werden unterschiedliche Ausgangstreiber eingesetzt:

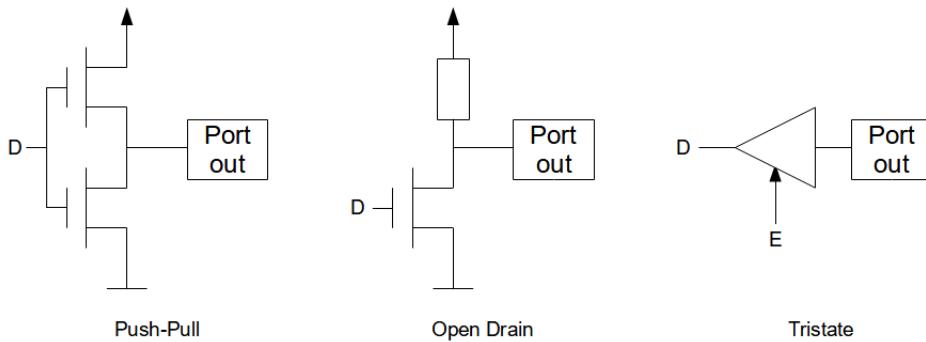


Abbildung 13.5.: Ausgangstreiber

13.4.1. Push-Pull

Push-Pull Stufen geben immer ein Signal auf den Ausgang. Sie dürfen deshalb nicht parallel geschaltet werden!

1. logisch low (oder „0“)
2. logisch high (oder „1“)

13.4.2. Open Drain

Open Drain (bei FET) oder Open Collector (bei Bipolar) Ausgänge haben einen einzigen Transistor am Ausgang, welcher einen Pullup-Widerstand gegen Masse ziehen kann. Der Pullup-Widerstand kann entweder im IC integriert oder extern beschalten sein. Open Drain respektive Open Collector Ausgänge können parallel geschaltet werden.

Beispiel: Parallelschaltung von Ausgängen verschiedener Interrupt-Quellen, welche auf einen interruptfähigen Pin des Microcontrollers (IRQx) geführt werden:

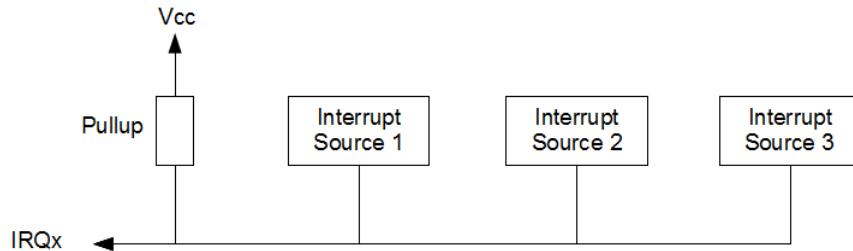


Abbildung 13.6.: Interrupt-Quellen mit Open-Drain-Ausgängen

13.4.3. Tristate

Tristate-Ausgänge können einen hochohmigen Zustand einnehmen. Der Zustand „Ausgang aktiv“ oder „Tristate“ wird durch eine zusätzliche Steuerleitung (E, enable) definiert. Im allgemeinen sind Ausgänge, welche Signale auf einen Databus ausgeben, mit Tristate-Treibern ausgestattet (z.B. Daten-I/O eines RAM-Bausteines).

14. Exceptions und Interrupts

14.1. Einleitung

In diesem Kapitel wird die Exception-Verarbeitung beschrieben, insbesondere auch, was auf der Hardware bei einer Exception abläuft, und wie Exceptions in der Software programmiert werden.

Der Begriff Exception ist ein Oberbegriff für Ereignisse, welche während der Ausführung eines Programms auftreten können und eine Ausnahmebehandlung erfordern. Das kann ein Hardware-Fehler auf der CPU oder auch eine externe Quelle für ein Ereignis (Interrupt) sein. Bei ARM Cortex-Mx Prozessoren lassen sich Exceptions in folgende Kategorien einteilen:

1. System Exceptions: Diese werden vom Prozessor-Core erzeugt, beispielsweise durch einen Reset, Hardware-Fehler, durch den Debug-Monitor oder durch den SysTick Timer.
2. Interrupts: Diese werden durch die Hardware-Peripherie erzeugt, beispielsweise Timer, Kommunikations-Schnittstellen oder I/O-Ports.

Die verschiedenen Quellen von Exceptions sind in Abbildung 14.1 dargestellt. Links sind verschiedene Peripherie-Bausteine ersichtlich, welche einen Interrupt auslösen können. Diese Peripherie-Bausteine können auf demselben Chip sein wie der Prozessor-Core, oder sie können in einem externen Baustein integriert sein. GPIOs können je nach Konfiguration einen Interrupt auslösen, beispielsweise ein Taster, welcher auf ein GPIO geführt ist. Der SysTick Timer wird insbesondere bei Betriebssystemen verwendet, kann aber auch in anderen Systemen als Zeitbasis dienen. Der SysTick löst periodisch eine System-Exception aus, welche vom System dazu verwendet wird, die Zeit nachzuführen.

Der NMI (Non Maskable Interrupt) ist ein Interrupt, welcher nicht deaktiviert werden kann. Der NMI wird typischerweise durch folgende Ereignisse getriggert:

- Durch den Watchdog (überwacht den Systemzustand und löst bei Problemen den NMI oder den Reset aus).
- Durch einen Brownout-Detector (überwacht die Speisespannung und gibt eine Warnung aus, wenn diese unter ein gewisses Niveau fällt).

Der NVIC (Nested Vectored Interrupt Controller) wird in Kapitel 14.6 beschrieben.

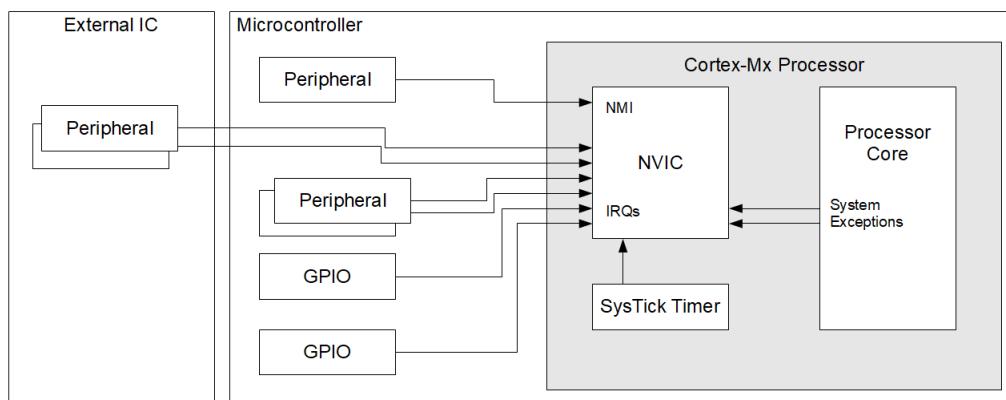


Abbildung 14.1.: Quellen von Exceptions, [13]

14.2. Eigenschaften

Tritt eine Exception auf, so wird eine spezielle „Subroutine“ aufgerufen, welche das entsprechende Ereignis bearbeitet. In diesem Zusammenhang spricht man von einem „Exception Handler“ oder im Falle von Interrupts von Interrupt Handler und ISR (Interrupt Service Routine).

Der Programmablauf ist für alle Exceptions derselbe: Das ausgeführte Programm wird unterbrochen und das Ereignis wird durch den entsprechenden Exception Handler bearbeitet. Eine genauere Beschreibung des Ablaufs finden Sie in Kapitel 14.9.

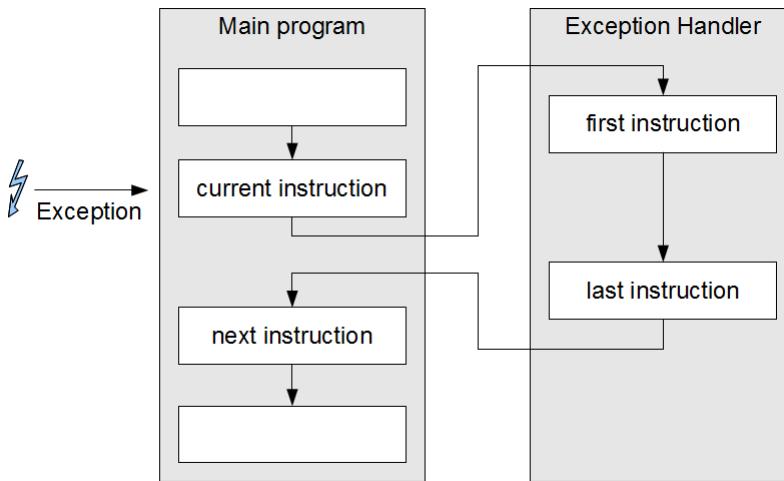


Abbildung 14.2.: Programmablauf beim Eintreffen einer Exception

Exceptions besitzen weiter folgende Eigenschaften:

- Im Unterschied zu Subroutinen, welche durch die Branch-and-Link Instruktion aufgerufen werden, werden Exception Handler durch ein Ereignis indirekt aufgerufen. Die „Vektor-Tabelle“ definiert für jede Exception, wo der entsprechende Exception Handler zu finden ist (siehe auch Kapitel 14.7).
- Das Auftreten einer Exception führt dazu, dass die CPU in den Handler Mode wechselt (siehe Kapitel 5.3).
- Exceptions können aktiviert und deaktiviert werden, was auch Maskierung genannt wird (siehe Kapitel 14.4).
- Exceptions haben unterschiedliche Prioritäten, diese können konfiguriert werden (siehe Kapitel 14.5).

14.3. Interrupts in Embedded Systems

Interrupts sind für Embedded Systems sehr wichtig. Sie ermöglichen es einer Applikation, auf Ereignisse schnell zu reagieren. Dies wird auch Echtzeitfähigkeit genannt. Solche Systems haben ein deterministisches Verhalten, d.h. Sie sind in der Lage, auf Ereignisse in einer voraussagbaren Zeit zu reagieren.

Interrupts können in zwei Kategorien aufgeteilt werden: Hardware-Interrupts und Software-Interrupts.

14.3.1. Hardware-Interrupts

Hardware-Interrupts treten asynchron zum Programmfluss auf. Hardware Interrupts werden immer durch ein CPU-externes Hardware-Ereignis ausgelöst. Dies kann beispielsweise ein Taster, eine serielle Schnittstelle oder ein A/D-Wandler sein. Die möglichen Quellen für Interrupts sind abhängig vom Prozessor und von der angeschlossenen Peripherie.

Bei ARM Cortex-Mx Prozessoren existieren sehr viele Interrupt-Quellen, deshalb werden diese Interrupts durch einen Interrupt-Controller verwaltet: Der NVIC (Nested Vectored Interrupt Controller). Mehr zum NVIC im Kapitel 14.6.

Beispiel für Hardware-Interrupts:

Die Software eines Microcontrollers steuert ein Gerät mit einem Display, Tastern und einer seriellen Schnittstelle. Um die Taster einzulesen oder Daten von der seriellen Schnittstelle zu empfangen gibt es zwei Ansätze:

1. Die Eingänge und die serielle Schnittstelle werden periodisch gepollt. Die Software prüft zyklisch, z.B. alle 10 Millisekunden, ob ein Taster gedrückt wurde oder ob ein Zeichen über die serielle Schnittstelle empfangen wurde. Dieser Ansatz hat verschiedene Nachteile:
 - Es wird viel Rechenzeit verschwendet, um die Eingänge dauernd zu pollen. Im „Normalfall“ wird die Software feststellen, dass kein Taster gedrückt und kein Zeichen empfangen wurde.
 - Ist die Zeit zwischen zwei Poll-Zyklen zu lange, kann es sein, dass ein Ereignis verpasst wird. Beispielsweise wenn die Taster jede Sekunde gepollt werden, der Anwender einen Taster aber nur 500 Millisekunden betätigt.
 - Die Zeit, die verstreicht, bis auf das Ereignis reagiert wird, ist im worst case gleich der Zeit, die zwischen zwei Poll-Zyklen liegt.
2. Die Taster lösen einen Interrupt Request aus, wenn sie betätigt werden. Die serielle Schnittstelle löst ebenfalls einen Interrupt aus, wenn sie ein Zeichen empfängt. Dieser Ansatz hat folgende Vorteile:
 - Die Reaktionszeit auf ein Ereignis ist sehr kurz.
 - Die CPU wird nur belastet falls wirklich eine Reaktion erforderlich ist.

14.3.2. Software-Interrupts

Software-Interrupts werden bei der Abarbeitung durch die CPU wie Hardware-Interrupts behandelt. Sie werden aber synchron aus dem Programmcode aufgerufen und sind somit eine Mischung aus Subroutinen-Aufruf und Interruptverarbeitung. Software-Interrupts werden häufig verwendet, um Betriebssystemfunktionen aufzurufen: Durch den Software-Interrupt wechselt der Prozessor in den Handler-Mode, welcher privilegierten Zugriff auf die Hardware hat. Das Betriebssystem hat dadurch sämtliche Rechte für den Zugriff auf alle Register usw.

Die Assembler-Instruktion SVC löst eine SVC-Exception aus und dadurch wird der SVC-Handler aufgerufen.

14.4. Maskierung

14.4.1. Übersicht

Exceptions können gesperrt (maskiert, disabled) und freigegeben (enabled) werden. Die Anwendung kann dadurch definieren, welche Exceptions zu welchem Zeitpunkt aktiv oder inaktiv sind. Die Maskierung erfolgt über spezifische Register. Bei Cortex-Mx Prozessoren erfolgt die Maskierung mehrstufig:

1. In einer ersten Stufe können Exceptions global gesperrt oder freigegeben werden. Dies ist oft notwendig, um kritische Codesequenzen am Stück auszuführen, ohne dass diese unterbrochen werden können. Die Programmierung erfolgt über die Register PRIMASK und BASEPRI im NVIC. Für den Zugriff auf diese Register muss die CPU privilegierte Rechte haben. Mit Hilfe des Registers PRIMASK werden alle Exceptions gesperrt, außer Reset, NMI und HardFault. Mit Hilfe des Registers BASEPRI können Exceptions mit Prioritäten kleiner als ein gewisser Wert gesperrt werden, alle anderen Exceptions sind aktiv. BASEPRI wird oft im Zusammenhang mit Betriebssystemen verwendet.
2. In einer zweiten Stufe kann jede einzelne Vektor-Adresse im NVIC aktiviert oder deaktiviert werden. Dazu werden die Register ISER[x] (Interrupt Set Enable Register) und ICER (Interrupt Clear Enable Register) des NVIC verwendet.

3. In einer dritten Stufe kann jede einzelne Interrupt-Quelle aktiviert oder deaktiviert werden. Die entsprechenden Register sind im Datenblatt des Prozessors unter den Peripherie-Beschreibungen zu finden. Beim STM32F743 müssen zusätzlich Register im EXTI-Controller (siehe Kapitel 14.8) initialisiert werden.

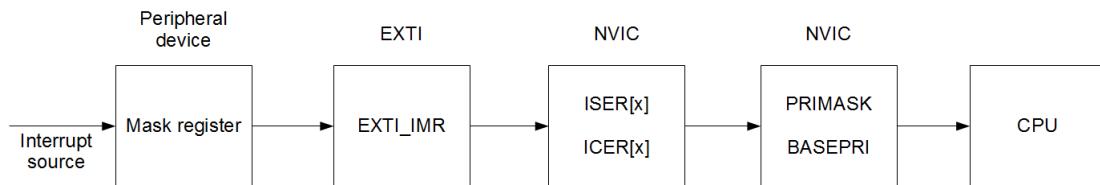


Abbildung 14.3.: Mehrstufige Maskierung von Interrupts beim STM32F743

14.4.2. Programmierung in C mit CMSIS

In der Programmiersprache C können Interrupts global mit Hilfe von CMSIS (siehe Anhang 12.4) sehr einfach maskiert werden. Dazu werden Funktionen zur Verfügung gestellt, welche wie folgt definiert sind:

```

void __enable_irq(void);           // clear PRIMASK
void __disable_irq(void);          // set PRIMASK
...
__set_BASEPRI(uint32_t x);         // disable interrupts with priority x and lower
uint32_t __get_BASEPRI(void);      // read BASEPRI value
  
```

Listing 14.1: Maskieren der Interrupts in C

14.4.3. Programmierung in Assembler

In Assembler müssen Sie direkt auf die Register PRIMASK oder BASEPRI des NVIC zugreifen, um Interrupts global zu maskieren. Dazu brauchen Sie die Instruktionen MRS und MSR, welche den Datentransfer zwischen CPU und speziellen Registern wie PRIMASK erlauben. Ein direkter Zugriff von der CPU auf das PRIMASK- oder BASEPRI-Register ist nicht möglich.

```

MOV r0, #1
MSR PRIMASK, r0      @ write 1 to PRIMASK to disable all interrupts
...
MOV r0, #0
MSR PRIMASK, r0      @ write 0 to PRIMASK to enable all interrupts
...
MOV r0, #0x40
MSR BASEPRI, r0       @ disable interrupts with priority 0x40 and lower
  
```

Listing 14.2: Maskieren der Interrupts in Assembler mit PRIMASK

Anmerkung: In der Programmiersprache C ist es nicht möglich, auf die CPU-Register zuzugreifen. Dies gilt für die Standardregister Rx aber auch für die Spezialregister wie PRIMAK oder BASEPRI. Funktionen wie `__enable_irq(void)` sind deshalb Assembler-Soubrutinen oder C-Funktionen, welche sogenannte Inline-Assembler Instruktionen 11.6 brauchen.

14.5. Prioritäten

14.5.1. Übersicht

Treten gleichzeitig mehrere Exceptions auf, so stellen sich folgende Fragen:

- Welche Exception wird zuerst bedient?
- Können Exception Handler durch andere Exceptions unterbrochen werden?

Bei den meisten Prozessoren sind Exceptions priorisiert oder können teilweise sogar zur Laufzeit anwendungsspezifisch priorisiert werden. Bei ARM Cortex-M3, Cortex-M4 und Cortex-M7 Prozessoren ist dies auch der Fall. Die Applikation kann für alle Interrupts und einen Teil der System Exceptions die Priorität dynamisch festlegen.

Eine Exception mit einer hohen Priorität (tiefe Zahl der Priorität) kann eine Exception mit einer tiefen Priorität unterbrechen. Dies bedeutet, dass der Exception Handler, welcher die Exception mit tiefer Priorität bearbeitet, beim Auftreten einer Exception mit hoher Priorität unterbrochen wird. Der zeitliche Ablauf dieses Szenarios ist in Abbildung 14.4 dargestellt. Dieses Szenario entspricht einer Verschachtelung von Exceptions (Nested Exceptions / Interrupts).

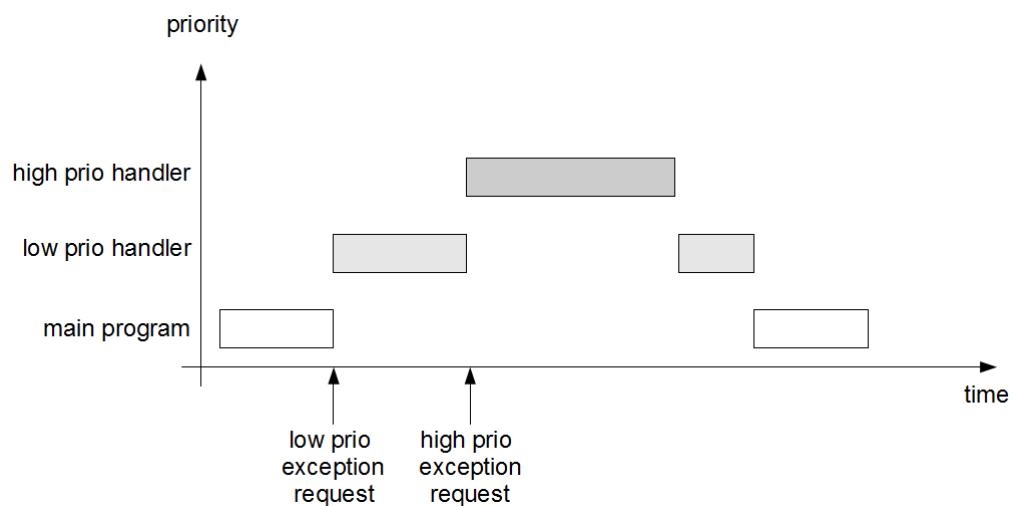


Abbildung 14.4.: Priorisierung und Verschachtelung von Exceptions

Bei Cortex-M3, Cortex-M4 und Cortex-M7 stellt der NVIC für die Programmierung der Priorität die Register IP[x] zur Verfügung, d.h. für jede Exception existiert ein eigenes Register. Die IP[x] Register sind Byte-Register, welche für die Definition der Priorität verwendet werden. Weiter Informationen finden Sie in [2].

14.5.2. Programmierung in C mit CMSIS

CMSIS stellt auch für den Zugriff auf die Register des NVIC diverse Funktionen zu Verfügung (siehe auch Kapitel 14.6). Um lediglich die Priorität zu modifizieren oder abzufragen, können folgende Funktionen oder Datenstrukturen verwendet werden:

```
// using CMSIS functions
void NVIC_SetPriority(IRQn_Type IRQn, uint32_t priority);
uint32_t NVIC_GetPriority(IRQn_Type IRQn);

// direct register access using NVIC typedef
NVIC->IP[0] = 0x80;
```

Listing 14.3: Definition der Priorität von Interrupts in C

14.5.3. Programmierung in Assembler

In Assembler kann direkt auf das IP[x]-Register zugegriffen werden.

```
LDR r0, =0xE000E400    @ load address of NVIC-IP[0] register to r0
LDR r1, =0x80            @ load value (prio) to be stored in IP[0], value 0b10000000
STR r1, [r0]             @ set IP[0] register with value 0x80
```

Listing 14.4: Definition der Priorität von Interrupts in Assembler

14.6. NVIC

14.6.1. Übersicht

Der Interrupt Controller ist für die Verarbeitung der Exceptions und Interrupts zuständig. Bei den Cortex-M3, Cortex-M4 und Cortex-M7 Prozessoren ist der NVIC (Nested Vectored Interrupt Controller) ein Bestandteil des Cores. Der NVIC besitzt folgende Eigenschaften:

- Unterstützung von bis zu 240 Interrupt-Eingängen, einem NMI sowie 15 System Exceptions.
- Alle Interrupts und einige System Exceptions können einzeln priorisiert werden.
- Interrupts können einzeln maskiert werden (enable und disable).
- Verschachtelte Interrupts (nested interrupts) werden automatisch verarbeitet, abhängig von deren Priorität.
- Vectored Interrupt: Jeder Exception Handler erhält einen Eintrag (Interrupt-Vektor) in der Vektortabelle (siehe Kapitel 14.7). Beim Eintreffen einer Exception liest der NVIC den zur Exception gehörigen Interrupt-Vektor, wo die Adresse des Exceptions Handlers abgelegt ist, und ruft den Handler auf.
- Die Startadresse der Vektortabelle (Vector Table Offset) kann dynamisch konfiguriert werden. Defaultmäßig startet die Vektortabelle auf Adresse 0 (Einsprung nach einem Reset).
- Der Wechsel vom Hauptprogramm in den Exception Handler ist mit 12 Zyklen sehr kurz. Während dieser Zeit werden einige Register auf dem Stack gespeichert (Stacking, siehe Kapitel 14.9).
- Interrupts und Exceptions werden durch Hardware-Ereignisse getriggert. Beim NVIC können diese Ereignisse zusätzlich aus der Software getriggert werden. Das ist für Tests sehr praktisch.

Interrupt-Quellen können als Puls (pulse triggered) oder zustandsgetriggert (level triggered) auftreten. Ein Puls muss aber mindestens einen Clockzyklus lang dauern.

14.6.2. Zustände

Interrupts können verschiedene Zustände annehmen, dies sind:

- Enabled oder Disabled.
- Pending (der Interrupt liegt an und wartet auf die Verarbeitung durch den Interrupt Handler) oder „not pending“.
- Active (der Interrupt wird durch den Handler verarbeitet) oder inactive.

Tritt ein Interrupt-Request (ein Ereignis, welches einen Interrupt auslöst) ein, so kann der Ablauf wie folgt dargestellt werden:

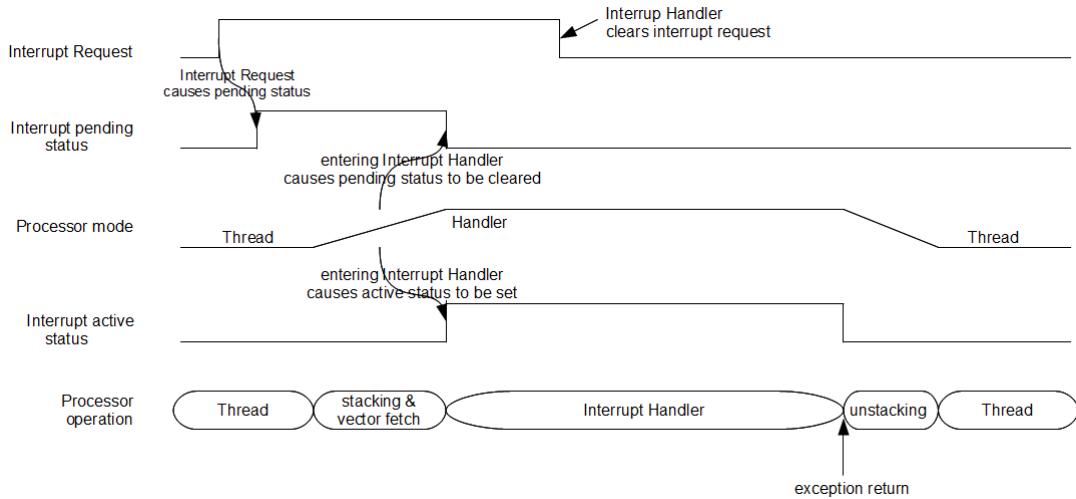


Abbildung 14.5.: Interrupt pending und Interrupt aktiv, [13]

Stacking bedeutet, dass einzelne Register der CPU auf dem Stack abgelegt werden. Unstacking bedeutet, dass diese Register am Schluss des Handlers wieder zurückgeholt werden. Mehr dazu im Kapitel 14.9.

14.6.3. Register

Der NVIC besitzt eine Anzahl Register, über die er konfiguriert werden kann. Diese sind über das Memory-Map zugänglich [2].

Adress	Register	CMSIS-Core	Function
0xE000E100 to 0xE000E13C	Interrupt Set Enable Register	NVIC→ISER[0] to NVIC→ISER[15]	Write 1 to set enable
0xE000E180 to 0xE000E1BC	Interrupt Clear Enable Register	NVIC→ICER[0] to NVIC→ICER[15]	Write 1 to clear enable
0xE000E200 to 0xE000E23C	Interrupt Set Pending Register	NVIC→ISPR[0] to NVIC→ISPR[15]	Write 1 to set pending status
0xE000E280 to 0xE000E2BC	Interrupt Clear Pending Register	NVIC→ICPR[0] to NVIC→ICPR[15]	Write 1 to clear pending status
0xE000E300 to 0xE000E33C	Interrupt Active Bit Register	NVIC→IABR[0] to NVIC→IABR[15]	Active status bit, read only.
0xE000E400 to 0xE000E5EC	Interrupt Priority Register	NVIC→IP[0] to NVIC→IP[123]	Level (8 bit) for each interrupt

Tabelle 14.1.: Übersicht Register NVIC

14.6.4. Programmierung in C mit CMSIS

Das CMSIS-Interface stellt diverse Funktionen für den Zugriff auf die Register des NVIC zur Verfügung.

```
// using CMSIS functions to access each register
void NVIC_EnableIRQ(IRQn_Type IRQn); // enable interrupt
void NVIC_DisableIRQ(IRQn_Type IRQn); // disable interrupt
void NVIC_SetPendingIRQ(IRQn_Type IRQn); // set pending status of an interrupt
void NVIC_ClearPendingIRQ(IRQn_Type IRQn); // clear pending status of an interrupt
uint32_t NVIC_GetPendingIRQ(IRQn_Type IRQn); // get pending status of an interrupt
```

```

void NVIC_SetPriority(IRQn_Type IRQn, uint32_t priority); // set the priority level of an
    interrupt
uint32_t NVIC_GetPriority(IRQn_Type IRQn); // Get the priority level of an interrupt

// using function NVIC_Init() to set an interrupt (example: EXTI9_5_IRQHandler)
NVIC_InitTypeDef NVIC_InitStructure;
...
NVIC_InitStructure.NVIC_IRQChannel = EXTI9_5_IRQHandler;
NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 8;
NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
NVIC_InitStructure.NVIC_IRQChannelCmd = 1;
NVIC_Init(&NVIC_InitStructure);

// direct register access using NVIC typedef
NVIC->ISER[0] = 0x01; // Enable Interrupt number 0

```

Listing 14.5: Zugriff auf den NVIC in C

14.6.5. Programmierung in Assembler

In Assembler kann ebenfalls auf jedes einzelne Register des NVIC zugegriffen werden. Nachfolgend wird ein Beispiel für das Set-Enable-Register ISER[0] (Adresse 0xE000E100) gezeigt, wo Bit 0 gesetzt werden soll:

```

LDR r0, =0xE000E100      @ load address of ISER[0] register into r0
LDR r1, =0x01              @ load value (interrupt number) to be stored in ISER[0]
STR r1, [r0]                @ set ISER[0] register with value 0x01

```

Listing 14.6: Zugriff auf das ISER-Register des NVIC in Assembler

14.7. Vektortabelle

Wird ein Exception Request ausgelöst, so wird der Code des zugehörigen Exception Handlers ausgeführt. Die Verknüpfung zwischen Request und Handler erfolgt über die Vektortabelle. Eine Vektortabelle ist eine Ansammlung von Vektor-Adressen, wobei in jeder Vektor-Adresse die Adresse eines Exception-Handlers eingetragen wird. Jede Exception hat ihre eigene Nummer und dadurch auch einen festen Platz in der Vektortabelle. Weitere Einzelheiten zum Ablauf finden Sie in Kapitel 14.9.

Die Basisadresse der Vektortabelle liegt bei vielen Prozessoren auf Adresse 0, weil sich an dieser Stelle das Flash (der Programmspeicher) befindet und diese Adresse nach einem Interrupt angesprungen wird. Soll sich die Vektortabelle im RAM befinden, wodurch die Einträge in die Vektor-Adressen zur Laufzeit verändert werden können, kann der Start der Vektortabelle zur Laufzeit auch verändert werden (VTOR, Vector Table Offset Register). Bei ARM-Cortex Mx Prozessoren ist dieser Offset defaultmäßig auf 0x08000000 definiert. D.h. das Vector Table Offset Register hat nach einem Reset den Wert 0x08000000 und das Flash respektive die Vektortabelle beginnt auf 0x08000000.

Es gibt zwei Varianten, was in einer Vektor-Adresse eingetragen wird. Bei einigen Prozessoren ist dies eine Instruktion, üblicherweise ein Branch zu einer bestimmten Adresse. Bei den ARM Cortex-M3, Cortex-M4 und Cortex-M7 Prozessoren wird die Adresse des Exception Handlers in die Vektor-Adresse eingetragen. Die CPU wird diese Adresse lesen und einen Sprung auf diese Adresse ausführen, was gleichbedeutend ist mit dem Aufruf des entsprechenden Exception-Handlers.

Die Einträge der Vektortabelle für ARM Cortex-M3, Cortex-M4 und Cortex-M7 Prozessoren sind wie folgt gegliebert:

1. Eintrag Position 0: Initialisierungswert für den Main Stack-Pointer (MSP)
2. Einträge Positionen 1 - 15: System Exceptions
3. Einträge Positionen 16 und grösser: Interrupts

Exception Number	CMSIS Interrupt Number	Table Address Offset	Exception Type	Priority	Description
-	-	0x00	-	-	Initial value of Main Stack-Pointer
1	-	0x04	Reset	-3(höchste)	Reset
2	-14	0x08	NMI	-2	Non-maskable Interrupt
3	-13	0x0C	HardFault	-1	All classes of fault when handler cannot be activated
4	-12	0x10	MemManagement	programmable	Memory Management fault
5	-11	0x14	BusFault	programmable	Bus-Systems fault
6	-10	0x18	Usage Fault	programmable	Invalid instruction or state transition
7	-	0x1C	-	-	reserved
8	-	0x20	-	-	reserved
9	-	0x24	-	-	reserved
10	-	0x28	-	-	reserved
11	-5	0x2C	SVC	programmable	supervisor call
12	-4	0x30	Debug Monitor	programmable	software based debug
13	-	0x34	-	-	reserved
14	-2	0x38	PendSV	programmable	pendable request for system service
15	-1	0x3C	SysTick	programmable	System Tick Timer
16	0	0x40	IRQ	programmable	IRQ Input # 0
...	IRQ Input # n
255	239	0x3FF	IRQ	...	IRQ Input # 239

Tabelle 14.2.: Vektortabelle ARM Cortex-M3, Cortex-M4 und Cortex-M7

Welche Interrupts in einer Vektortabelle wirklich implementiert sind, können Sie dem Datenblatt des entsprechenden Microcontrollers entnehmen.

14.8. External Interrupt/Event Controller (EXTI)

14.8.1. Übersicht

Der EXTI (External Interrupt/Event Controller) ist ein zusätzlicher Interrupt-Controller des STM32H7xx. Er überwacht bis zu 16 GPIO und weitere Interrupt-Quellen des Prozessors und löst bei Bedarf beim NVIC einen Interrupt Request aus. Detaillierte Infos zum EXTI-Controller finden Sie in [11]. Die wichtigsten Eigenschaften des EXTI sind:

- Jedes Bit kann einzeln maskiert werden (Interrupt aktivieren, deaktivieren).
- Statusabfrage für jedes einzelne Bit.
- Flankentriggierung (steigend, fallend oder beides) konfigurierbar.
- Jede dieser Interruptquellen kann auch softwaremäßig getriggert werden.

Das Blockdiagramm des External Interrupt/Event Controllers ist in Abbildung 14.6 dargestellt:

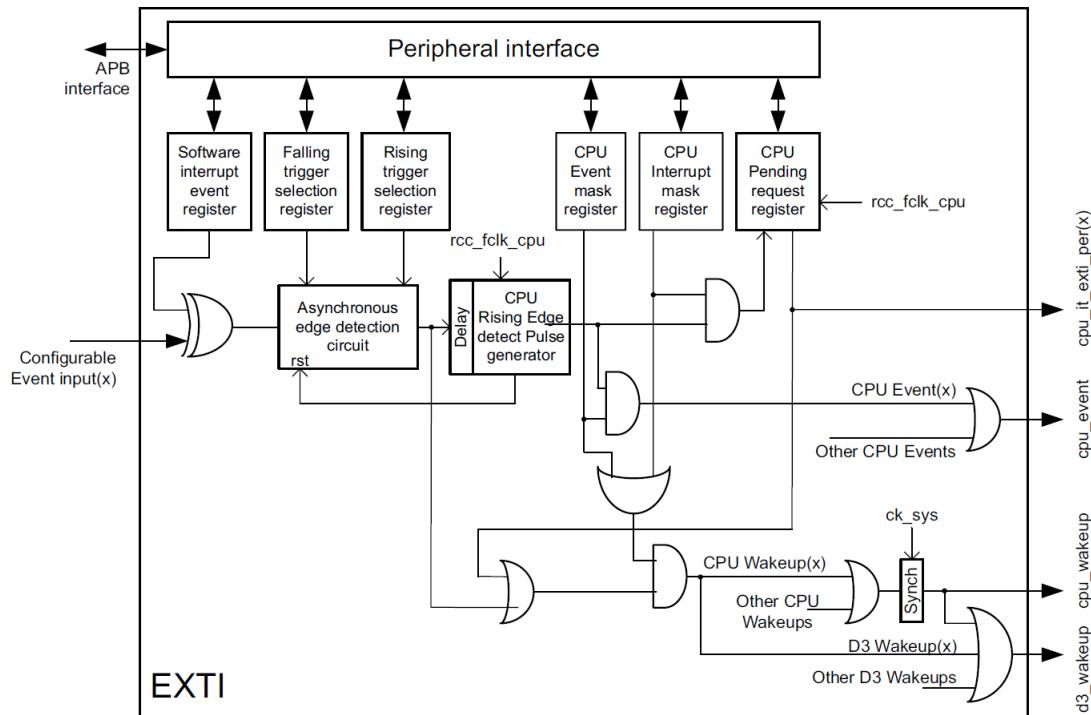


Abbildung 14.6.: Blockschaltbild EXTI STM32H7xx, [11]

Jedes GPIO Port des STM32H7xx (Port A bis Port K) hat jeweils 16 Ein-/Ausgänge. Alle Pins Nummer 0 (PA0 bis PK0) werden mit Hilfe eines Multiplexers auf die Line EXTI0 gemappt, alle Pins Nummer 1 (PA1 bis PK1) werden auf EXTI1 gemappt usw. In den Registern SYSCFG_EXTICR1 bis SYSCFG_EXTICR4 kann definiert werden, welche GPIO-Pins auf welche Line gemappt werden (siehe Reference Manual des STM32H743, [11]). Die restlichen EXTI-Lines (Line 16 bis Line 88) werden für RTC, I2C, UART, SPI usw. verwendet.

Der External Interrupt/Event Controller stellt folgende Register zur Verfügung (vergleiche auch Abbildung 14.6):

Register	CMSIS	Function
CPU Interrupt Mask Register	EXTI→IMR	0: Write 1 to activate
CPU Event Mask Register	EXTI→EMR	0: Write 1 to activate
Rising Trigger Selection Register	EXTI→RTSR	Write 1 to enable rising edge
Falling Trigger Selection Register	EXTI→FTSR	Write 1 to enable falling edge
Software Interrupt Event Register	EXTI→SWIER	Write 1 to generate interrupt request
CPU Pending Request Register	EXTI→PR	1 if trigger request occurred

Tabelle 14.3.: Übersicht Register EXTI-Controller

Weitere Angaben zu den Registern des EXTI-Controllers finden Sie im Reference Manual des STM32H743, Kapitel 20, [11].

Um die Lines des EXTI-Controllers als Interrupt zu initialisieren müssen Sie wie folgt vorgehen:

1. Konfiguration des Registers EXTI_IMR für die entsprechende Line.
2. Flankentriggerung (Rising und Falling) konfigurieren (EXTI_RTSR und EXTI_FTSR).
3. Konfiguration des NVIC, dass die externen Interrupts aktiv sind (Enable und Mask Bits).

14.8.2. Programmierung in C mit der HAL-Library

Die HAL-Library von ST stellt diverse Funktionen für den Zugriff auf die Register des EXTI-Controllers zur Verfügung.

```
// using function HAL_EXTI_SetConfigLine() to config EXTI line 11 for GPIO D Pin 11
EXTI_ConfigTypeDef EXTI_ConfigStructure;

EXTI_HandleTypeDef myHandle;
HAL_EXTI_GetHandle(&myHandle, EXTI_LINE11);

/* Configure EXTI line port D pin 11 */
EXTI_ConfigStructure.Line = EXTI_LINE11;
EXTI_ConfigStructure.Mode = EXTI_MODE_INTERRUPT;
EXTI_ConfigStructure.Trigger = EXTI_TRIGGER_RISING;
EXTI_ConfigStructure.GPIOSel = EXTI_GPIOD;
HAL_EXTI_SetConfigLine(&myHandle, &EXTI_ConfigStructure);
```

Listing 14.7: Zugriff auf den EXTI-Controller mit Hilfe der HAL-Library

14.8.3. Programmierung in C der EXTI-Register

In der Programmiersprache C kann ebenfalls auf jedes einzelne Register des EXTI-Controllers zugegriffen werden. Nachfolgend wird ein Beispiel für das Interrupt Mask Register gezeigt, wo Bit 11 gesetzt werden soll:

```
// direct register access using EXTI typedef
EXTI->IMR1 |= 1 << 11; // line 11 is unmasked
```

Listing 14.8: Zugriff auf die EXTI-Register in C

14.9. Ablauf einer Exception

Der Ablauf beim Eintreffen einer Exception Request ist prozessorspezifisch. Die nachfolgenden Beschreibungen gelten für ARM Cortex-M3, Cortex-M4 und Cortex-M7 Prozessoren.

14.9.1. Exception Eingangssequenz

Während der Eingangssequenz werden diverse Operationen automatisch durch die Hardware ausgeführt. Dies sind für den Betrieb ohne FPU:

- Die aktuelle Instruktion wird in der Regel fertig ausgeführt. Bei Instruktionen, welche mehrere Clock-Zyklen dauern, können diese auch unterbrochen und nach der Exception-Behandlung fertig ausgeführt werden.
- Retten diverser Prozessor-Register auf dem Stack (R0 bis R3, R12, LR), retten der Rücksprungadresse und des Statusregisters auf dem Stack (siehe Abbildung 14.7). Dieser Ablauf wird „stacking“ genannt und bietet den Vorteil, dass Exception Handler in C-Code geschrieben werden können. Im Thread-Mode wird der Process Stack-Pointer (PSP) für das Retten der Register verwendet, sonst der Main Stack-Pointer (MSP).
- Aus der Vektortabelle wird aus dem entsprechenden Exception-Vektor die Startadresse des Exception Handlers gelesen.
- Aus dem Exception Handler wird die erste Instruktion geladen.

- Es werden diverse Register des NVIC aktualisiert (Pending Status, Active Status) sowie diverse Register der CPU (PSR, LR, PC, SP).

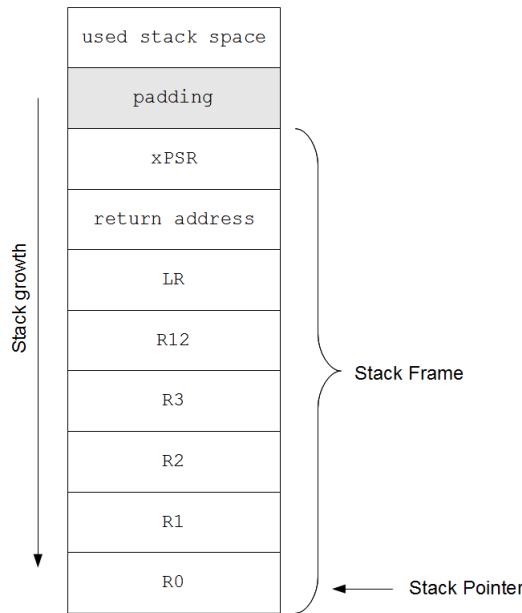


Abbildung 14.7.: Exception Stack nach Aufruf des Handlers (ohne FPU)

Bemerkung: Gemäss AAPCS wird der Stack-Pointer beim Aufruf eines Handlers Double-Word ausgerichtet. Dies ist beim Cortex-M3, Cortex-M4 und Cortex-M7 Prozessor defaultmäßig aktiviert. Je nach Stack-Zustand wird deshalb ein Padding-Word auf dem Stack eingefügt.

14.9.2. Ausführung des Exception Handler

Der Exception Handler wird im Handler-Mode ausgeführt. Dadurch wird der Main Stack-Pointer (MSP) verwendet und die CPU arbeitet im privilegierten Modus. Der laufende Exception Handler kann durch eine höher priorisierte Exception unterbrochen werden, nicht aber durch eine Exception gleicher oder tieferer Priorität. Am Schluss des Exception Handlers wird ein EXC_RETURN ausgeführt (siehe Kapitel 14.9.3).

14.9.3. Exception Rücksprung

Bei einigen Prozessoren müssen spezielle Rücksprung-Instruktionen aus einem Exception Handler verwendet werden (RTI, Return from Interrupt). Diese müssen entweder in Assembler oder durch spezielle Compiler-Direktiven programmiert werden.

Bei ARM Cortex-M3, Cortex-M4 und Cortex-M7 Prozessoren wird der Rücksprung aus einem Handler durch den Rückgabewert EXC_RETURN ausgelöst. EXC_RETURN wird beim Aufruf des Handlers automatisch in das Link-Register (LR) kopiert. Wird dieser Wert beim Rücksprung in den Program-Counter (PC) kopiert (Instruktion BX lr), so wird von der Hardware ein Rücksprungmechanismus aus einer Exception ausgeführt. Dabei werden die Register, welche bei der Eingangssequenz auf den Stack gelegt wurden, wieder zurück gelesen („unstacking“). Der C-Compiler kann den Wert EXC_RETURN im Link-Register wie eine normale Rücksprungadresse behandeln.

Der Rückgabewert EXC_RETURN ist wie folgt definiert:

Bits	Beschreibung	Wert
31 - 28	EXC_RETURN indicator	0xF
27 - 5	reserved	23 Bits value 1
4	Stack Frame Type	1 (8 Words, without FPU) 0 (26 Words, with FPU)
3	Return Mode	1 (return to Thread) 0 (return to Handler)
2	Return Stack	1 (return with PSP) 0 (return with MSP)
1	reserved	0
0	reserved	1

Tabelle 14.4.: Die einzelnen Bits von EXC_RETURN

14.9.4. Zusammenfassung

Der Ablauf einer Exception-Anforderung kann graphisch wie folgt zusammengefasst werden:

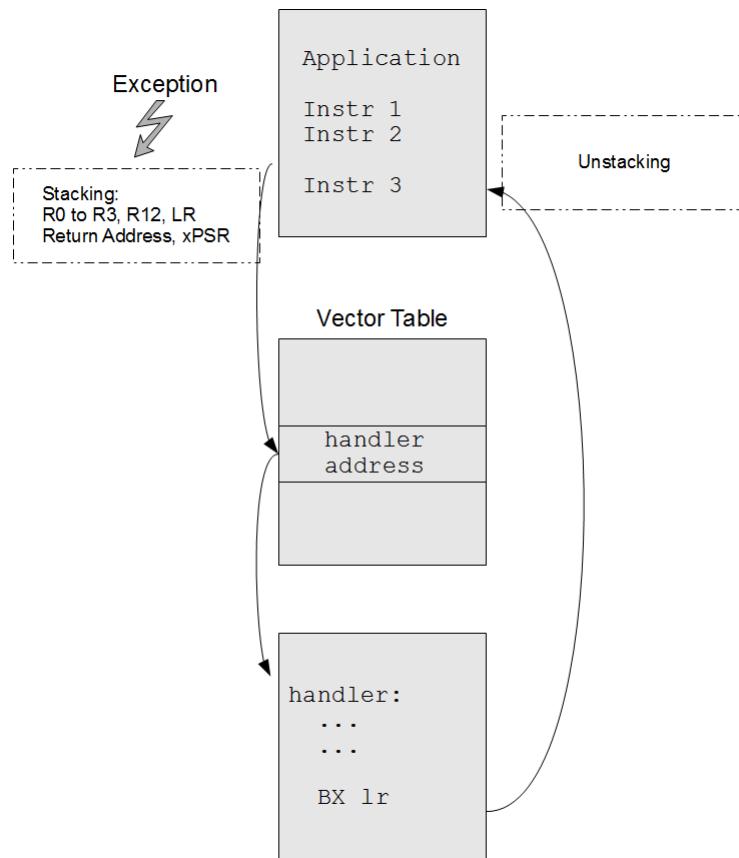


Abbildung 14.8.: Ablauf einer Exception-Bearbeitung

Im Beispiel in Abbildung 14.8 werden die Instruktionen 1 und 2 im Hauptprogramm sequentiell ausgeführt. Nun tritt ein Exception Request auf und das Hauptprogramm wird unterbrochen. Anschliessend werden einige Register und die Rücksprungadresse auf dem Stack abgelegt (stacking) und die Adresse des Exception Handlers wird aus der

Vektortabelle gelesen. Der Handler wird ausgeführt und bearbeitet die Exception. Am Schluss des Handlers erfolgt der Rücksprung und es wird der ursprüngliche Zustand wieder hergestellt (unstacking). Das Programm wird in der Applikation bei der nächsten Instruktion fortgesetzt.

14.9.5. Spezialfälle

Beim Ablauf von Exceptions gibt es einige Spezialfälle. Die wichtigsten sind:

- Wenn mehrere Exceptions hintereinander ausgeführt werden müssen, so wird das Unstacking und das anschliessende Stacking übersprungen. Am Schluss des ersten Handlers wird der zweite Handler mit einer Verzögerung von nur sechs Clock-Zyklen aufgerufen. Dieser Vorgang wird „Tail chaining“ genannt.
- Ist die FPU aktiv, so werden auch die Register S0 bis S15 sowie das Register FPSCR auf dem Stack gerettet. Dieser Vorgang würde zusätzlich 17 Clock-Zyklen dauern, was sehr lange ist. Beim Aufruf eines Exception Handlers wird deshalb erst der Speicher für die FPU-Register reserviert, die Register werden aber erst kopiert, wenn sie innerhalb des Handlers modifiziert werden. Dieser Vorgang wird „Lazy stacking“ genannt.

14.10. Zeitverhalten von Exceptions

Bei der Verarbeitung von Exceptions können Verzögerungen eintreten. Diese sind teilweise durch die Hardware gegeben (beispielsweise stacking), teilweise aber auch softwarebedingt (beispielsweise das Disablen von Interrupts).

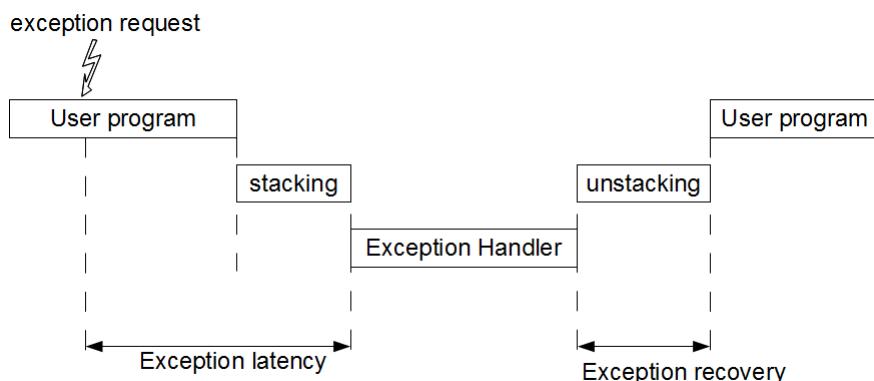


Abbildung 14.9.: Verzögerungszeiten bei der Verarbeitung von Exceptions

Die Exception Latenzzeit (Exception Latency, bei Interrupts entsprechend Interrupt Latency genannt) ist die Zeit zwischen dem Eintreffen der Exception-Anforderung (Exception Request) und dem Start der ersten Anweisung im Exception Handler:

$$t_{ELat} = t_{dis} + t_{stack} + t_{cmod}$$

t_{ELat} : Exception Latency

t_{dis} : Maximale Zeit, während der die Exception disabled sein können

t_{stack} : Zeit für das Retten der Register auf dem Stack, Stacking

t_{cmod} : Zeit für den Wechsel in den Exception Handler (Handler Mode)

Die Exception Recovery Zeit ist diejenige Zeit, die es braucht, um am Schluss eines Exception Handlers wieder in das Anwenderprogramm zu springen.

$$t_{ERec} = t_{unstack} + t_{cmodt}$$

t_{ERec} : Exception Recovery

$t_{unstack}$: Zeit um die geretteten Register auf dem Stack zurückzulesen, unstacking

t_{cmodt} : Zeit für Rücksprung in die Applikation (Thread Mode)

14.11. Varianten von Interrupt-Handlern

Die System Exceptions der ARM Cortex-M3, Cortex-M4 und Cortex-M7 Prozessoren haben nur eine mögliche Quelle. Bei Interrupts können jedoch für einen Eintrag in der Vektortabelle und dadurch für einen Interrupt Handler verschiedene Quellen in Frage kommen (z.B. der Handler „EXTI9_5_IRQHandler“ für die EXTI Line 5 bis 9). Der Interrupt Handler muss deshalb prüfen, welche Quelle für den Interrupt Request ist. Es gibt verschiedene Varianten, einen Interrupt-Handler in der Software zu implementieren. So kann beispielsweise die Verarbeitung im Handler in zwei Teile aufgeteilt werden. In einem ersten Teil wird das absolut Notwendige gemacht, anschließend werden die Interrupts wieder freigegeben und es wird der zweite Teil des Handlers ausgeführt. So können auch Interrupts gleicher oder tieferer Priorität den aktuellen Handler unterbrechen. Alle möglichen Handler werden ausführlich in [1] beschrieben. In diesem Kapitel wird der „Nonnested Interrupt Handler“ beschrieben. Dieser hat folgende Eigenschaften:

- Verarbeitet die Interrupts sequenziell. Während der Bearbeitung eines Interrupts kann kein weiterer Interrupt gleicher Priorität bearbeitet werden.
- Vorteil: Einfach zu implementieren und zu testen
- Nachteil: Hohe Interrupt-Latenzzeiten, was für komplexe Embedded Systems mit mehreren Interrupt-Quellen nachteilig ist.

Der zeitliche Ablauf ist für den „Nonnested Interrupt Handler“ wie folgt definiert:

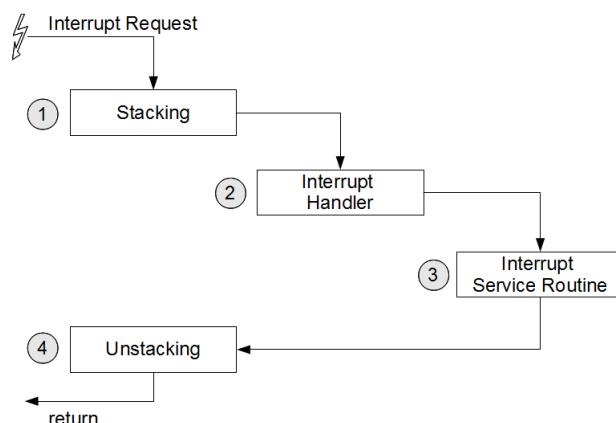


Abbildung 14.10.: Nonnested Interrupt Handler

Der „Nonnested Interrupt Handler“ wird wie folgt ausgeführt:

1. Stacking: Es werden diverse Register auf dem Stack gerettet (siehe Kapitel 14.9.1). Der Prozessor liest die Startadresse des Handlers aus der Vektortabelle und wechselt in den Handler-Mode.
2. Der Handler wird ausgeführt. Oft muss ein Handler verschiedene Interrupt-Quellen verarbeiten. In diesem Falle prüft der Handler die Quellen und ruft die entsprechenden Interrupt Service Routinen (ISR) auf.
3. In der Interrupt Service Routine wird das jeweilige Ereignis bearbeitet und der Interrupt wird zurück gesetzt. Falls nur eine Quelle für einen Interrupt möglich ist, kann diese auch direkt im Handler verarbeitet werden.
4. Unstacking: Die Register werden vom Stack zurück gelesen und der Prozessor springt wieder in den Thread-Mode (siehe Kapitel 14.9.3).

Beispiel: In C-Code kann der Nonnested Interrupt Handler für EXTI Line 11 (Taster auf Leguan-Board) mit Hilfe der HAL-Library wie folgt programmiert werden:

```

//----- Header-Files -----
#include "stm32h7xx_exti.h"
...
/********************* Function : EXTI15_10_IRQHandler ********************/

```

```
/** \brief      External interrupt on line 10 to 15
 *           Check if line 11 (button pd11) is active and call ISR
 *
 * \type      global
 *
 * \return    void
 *
 ****
void EXTI15_10_IRQHandler(void) {
    /* Check if line 11 is active */
    if((EXTI->PR1 & (1 << 11)) != 0){
        /* call IRQ 11 Interrupt Service Routine */
        IRQ11_ISR();
        /* clear pending interrupt */
        EXTI->PR1 = (1 << 11);
    }
    /* check other interrupt sources */
    ...
}
```

Listing 14.9: Interrupt Handler in C

15. RS232

15.1. Übersicht Serielle Schnittstellen

Serielle Schnittstellen werden zur Übertragung von Daten in einem seriellen Bitstrom eingesetzt (Bsp. RS232, I2C, SPI, FireWire, USB, Feldbusse ...). Ein Schieberegister wird zur Umwandlung des parallelen Datenformates in den seriellen Bitstrom (Senden) und umgekehrt (Empfangen) verwendet. Die Umwandlung Parallel / Seriell und umgekehrt erfolgt in sogenannten UARTs (Universal Asynchronous Receiver Transmitter).

Die wichtigsten Eigenschaften einer seriellen Schnittstelle sind:

- Baudrate
- Protokoll
- Spannungspegel

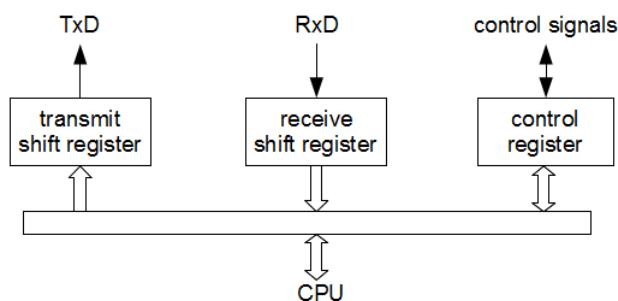


Abbildung 15.1.: Prinzipieller Aufbau einer seriellen Schnittstelle

15.2. Einleitung RS232

Die RS232 (Recommended Standard 232) ist bei Embedded Systems sehr verbreitet. Typische Anwendungen sind die Kommunikation mit einem PC für Konsolen-Applikationen um das System zu konfigurieren oder Daten auf dem PC darzustellen (z.B. ein Log). Hardwaremäßig haben viele Microcontroller bereits eine oder mehrere RS232-Schnittstellen integriert.

Die RS-232 ist eine serielle, asynchrone Schnittstelle. Es existiert somit kein Taktsignal, die Synchronisation erfolgt durch den Empfänger auf die Signalflanke des Startbits (siehe Kapitel 15.7). Die Verbindung ist vollduplex, weil für Senden und Empfangen getrennte Datenleitungen zur Verfügung stehen (RXD und TXD).

Aus traditionellen Gründen wird die RS-232 oft auch als „serielle Schnittstelle“ benannt, weil sie früher im PC-Bereich die einzige übliche war. Die Schnittstelle wurde von der CCITT normiert: V.24 für die Beschreibung der Stecker und V.28 für die Beschreibung der Signale und des Protokolls. In den USA wird die Schnittstelle EIA RS-232C genannt.

15.3. Leitungslänge und Übertragungsrate

Die Signalübertragung erfolgt asymmetrisch (also nicht differentiell). Somit beinhaltet das Signal einen Gleichspannungsanteil, was es anfällig für Störungen macht. So können beispielsweise induktive Einkopplungen über eine Schleife von Signalleitung und GND entstehen. Dies ist insbesondere in Applikationen in EMV-gestörter Umgebung zu beachten. Einen Ausweg bietet hier die RS-485 an, welche die Signale differentiell überträgt.

RS-232 wird an den Leitungsenden nicht mit dem Wellenwiderstand abgeschlossen, was zu Reflexionen führt. Dies hat schlussendlich auch einen Einfluss auf die maximale Kabellänge, welche abhängig von der Übertragungsrate ist. Gängige Übertragungsraten und deren maximale Kabellänge sind in Tabelle 15.1 zusammengefasst.

max. Baudrate [kbps]	max. Länge [m]
2'400	900
4'800	300
9'600	150
19'200	15
57'600	5
115'200	2

Tabelle 15.1.: Maximale Kabellänge bei RS-232

15.4. Steckerbelegung

In der ursprünglichen Norm wurden 25-polige DSUB-Stecker spezifiziert. Viele der Leitungen wurden für die Steuerung von Druckern und Terminals verwendet. Heute verwendet man diese Steuerleitungen nicht mehr, weshalb in der Regel 9-polige DSUB-Stecker verwendet werden.

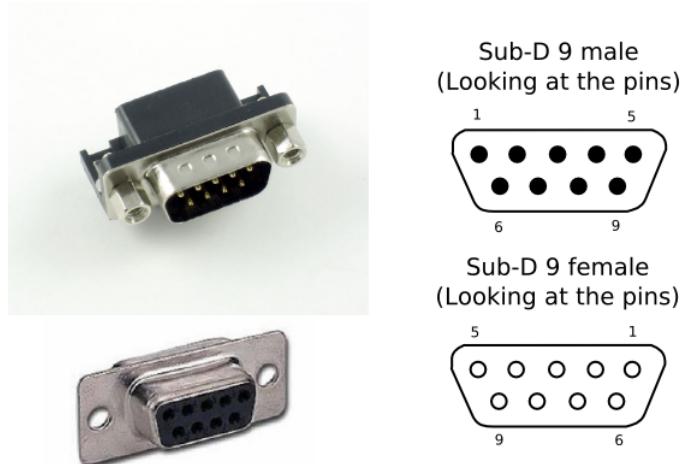


Abbildung 15.2.: RS-232-Stecker, oben Stecker (male), unten Buchse (female)

Pin-Nr	Abkürzung	Name	Beschreibung
1	DCD	Data Carrier Detect	Daten werden auf der Leitung erkannt.
2	RxD	Receive Data	Eingehende Daten.
3	TxD	Transmit Data	Ausgehende Daten.
4	DTR	Data Terminal Ready	Signalisation der Betriebsbereitschaft an die Gegenstation.
5	GDN	Ground	Signalmasse, Bezugspotential für die Signale.
6	DSR	Data Set Ready	Gerät ist einsatzbereit.
7	RTS	Request to Send	Sendeanforderung, Gerät möchte Daten senden.
8	CTS	Clear to Send	Sendeerlaubnis, Gerät kann Daten entgegennehmen.
9	RI	Ring Indicator	Ankommender Anruf, Gegenstation wünscht Datenverbindung aufzubauen.

Tabelle 15.2.: Pinbelegung RS-232 DSUB-9

15.5. Verkabelung

Werden zwei Systeme über RS-232 verbunden, müssen die Receive- und Transmit-Leitungen (RxD und TxD) sowie die Handshake-Leitungen korrekt verdrahtet werden.

- Kommuniziert ein Embedded System mit einem PC (auch Modem-Verbindung genannt), so ist ein 1:1 Kabel notwendig.
- Kommunizieren zwei Systeme gleichen Typs miteinander, so sind die Leitungen zu kreuzen (Nullmodem-Kabel). Im einfachsten Fall sind nur RxD und TxD zu tauschen, je nach System aber auch die Handshake-Leitungen CTS/RTS.

15.6. Signalpegel

Die Zustände der einzelnen Signale werden durch folgende Spannungen dargestellt:

- zwischen -3V und -15V für eine logische 1 für Daten oder passiver Zustand für Steuerleitungen („MARK“).
- zwischen +3V und +15V für eine logische 0 für Daten oder aktiver Zustand für Steuerleitungen („SPACE“).

Die Signalpegel der RS-232 werden durch Treiber-IC's (z.B. MAX3245) erzeugt. Diese arbeiten auf der Seite des Microcontrollers mit 3.3V oder 5V und transformieren diese auf die geforderte Busspannung auf der RS-232 Seite. Sie beinhalten DC/DC-Wandler sowie Bustreiber.

Abbildung 15.3 zeigt den Schaltplan eines RS232 Transceivers. Es kommt ein MAX3245 von Maxim Integrated zum Einsatz, der auch für die Handshake-Signale die erforderlichen Pegel erzeugt. Die beiden Kondensatoren zu je 1 μ F werden vom internen DC/DC-Wandler verwendet.

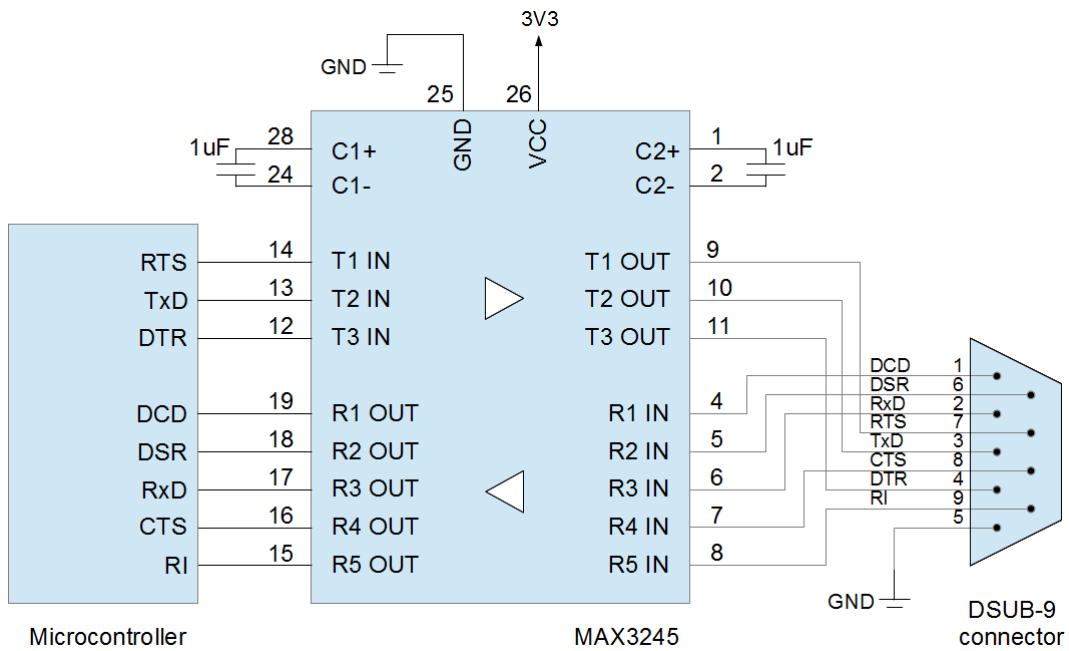


Abbildung 15.3.: MAX3245, RS-232 Bus-Transceiver

15.7. Bitcodierung

Ein RS-232 Frame beginnt mit dem Start-Bit, gefolgt von einigen Daten-Bits, einem optionalen Parity-Bit und abschliessend ein bis zwei Stop-Bits.

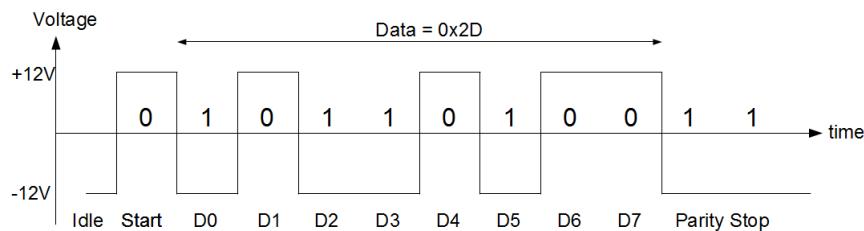


Abbildung 15.4.: RS-232 Frame

Bit	Beschreibung
Start	Es wird vom Empfänger für die Synchronisation verwendet. Durch das Start-Bit erfolgt ein Pegelwechsel auf dem Bus.
Data	Die Anzahl der Datenbits kann zwischen 5 und 8 konfiguriert werden. Das LSB wird zuerst übertragen.
Parity	Das Parity-Bit erlaubt eine einfache Fehlererkennung, jedoch keine Fehlerrichtigung. Das Parity-Bit wird von der Sender-Hardware automatisch so gesetzt, so dass die Anzahl "1" aller Datenbits und des Parity-Bits gerade ist (even-Parity) oder ungerade ist (odd-Parity). RS232-Bausteine können so konfiguriert werden, dass kein Parity, even-Parity oder odd-Parity unterstützt wird.
Stop	Das oder die Stop-Bits kennzeichnet das Ende des RS-232 Frames. Es handelt sich dabei nicht um ein Bit, sondern um einen Zeitraum, während dem sich die Leitung auf logisch 1 befindet. Typisch sind 1, 1.5 oder 2 Stop-Bits.

Tabelle 15.3.: RS-232 Bitcodierung

15.8. Handshake-Signale

Langsame Empfänger müssen die Datenübertragung unterbrechen können, um Datenverluste zu vermeiden. Dafür werden Handshake-Signale verwendet. Diese können entweder softwaremäßig über Steuercodes oder hardwaremäßig über spezielle Steuerleitungen realisiert werden:

Software-Handshake: Der Empfänger steuert den Datenfluss mit Steuercodes, die in den Nutzdaten nicht vorkommen dürfen. Typischer Vertreter ist das Xon/Xoff-Protokoll (Xon = 11h, Xoff = 13h). Für die Datenübertragung werden nur die drei Leitungen RxD, TxD und GND benötigt.

Hardware-Handshake: Das Handshake erfolgt über zusätzliche Steuerleitungen. Typischerweise sind dies RTS (Request to Send) und CTS (Clear to Send).

Bei Embedded Systems wird sehr oft auf die Handshake-Signale verzichtet. Somit müssen nur die Leitungen RxD, TxD und GND verdrahtet werden.

15.9. RS232 zu USB Konverter

Da heutige Laptops und PCs nur selten über einen RS-232 Anschluss verfügen, werden RS-232 zu USB Konverter eingesetzt. Diese können entweder als Spezialkabel mit integriertem Konverter in einem Stecker gekauft werden, oder die entsprechenden Chips werden direkt im Embedded System integriert, sodass nach aussen ein USB-Stecker verwendet wird. Die PC-seitigen Treiber für diese Konverter werden von den heutigen Betriebssystemen standardmäßig installiert, sodass der Anwender diese nicht nachinstallieren muss.

Abbildung 15.5 zeigt den Schaltplan des RS232 zu USB Konverters FT230XQ der Firma FTDI.

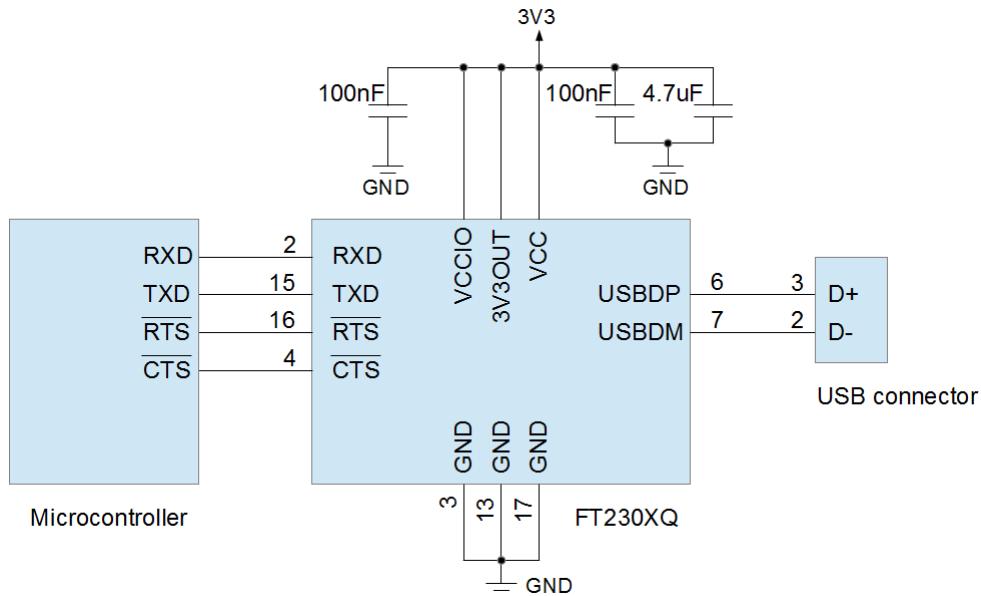


Abbildung 15.5.: Beispiel RS-232 zu USB converter FT230XQ

15.10. Blockschaltbild

Hardwaremäßig haben viele Microcontroller bereits eine oder mehrere RS232-Schnittstellen integriert. Für die Generierung der Baudrate wird oft ein Standard-Timer verwendet. Einige RS232-Schnittstellen haben jedoch eigene Baudrate-Generatoren integriert. Dadurch können die Standard-Timer für die Applikation verwendet werden.

Aus Sicht der Software wird die RS232-Schnittstelle über Register angesprochen:

- Sendebuffer und Empfangsbuffer, jeweils 1 Byte. Oft liegen beide Buffer auf derselben Adresse, weil in den Sendebuffer nur geschrieben und vom Empfangsbuffer nur gelesen wird.
- Controlregister zum Definieren des Betriebsmodus (Baudrate, Daten, Parity ...) und der Interrupts.

Ein generisches Blockschaltbild für eine RS232-Schnittstelle ist in Abbildung 15.6 dargestellt.

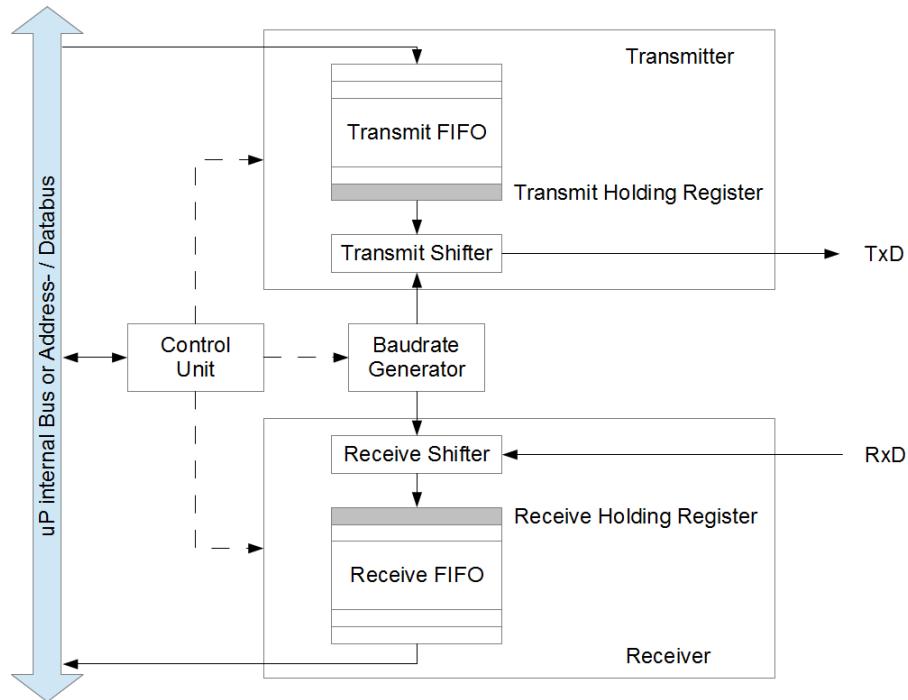


Abbildung 15.6.: Generisches Blockschaltbild RS232-Schnittstelle mit FIFO

Die Umsetzung für eine RS232-Schnittstelle im STM32H7xx ist in Abbildung 15.7 zu sehen. Dazu wird eine UART/USART des Microcontrollers verwendet. In der Mitte der Abbildung sind die beiden FIFOs sowie die Shift-Register für den Sende- und Empfangs-Pfad ersichtlich. Auf der linken Seite sind die Hardware-Register sichtbar, über welche die Software mit der USART/USART kommuniziert. Um einen Wert zu senden, muss die Software ins Register USART_TDR schreiben, um einen Wert zu empfangen vom Register USART_RDR lesen.

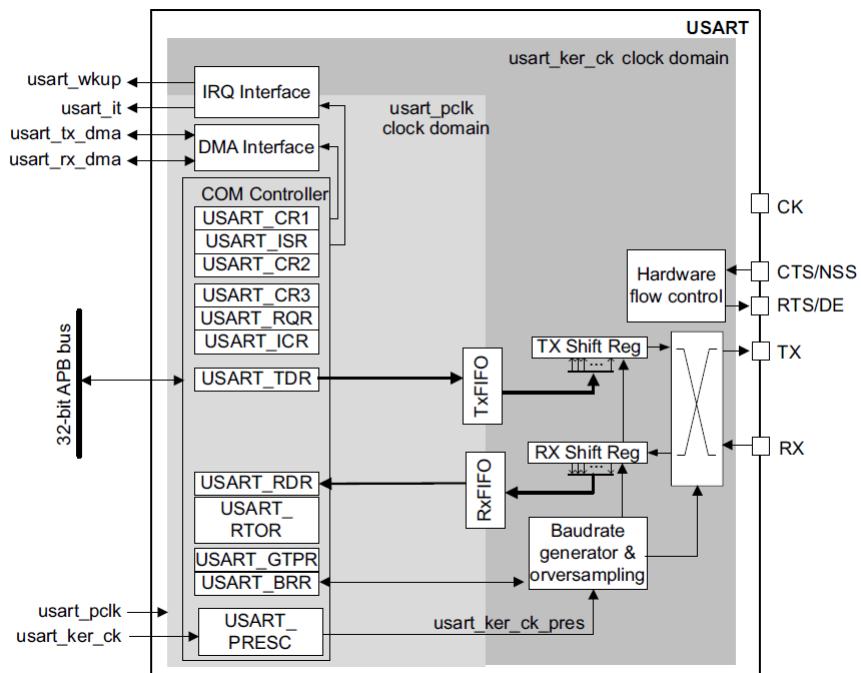


Abbildung 15.7.: USART/USART im STM32H7xx

15.11. Programmierung der RS232-Peripherie auf dem STM32H7xx

Die Programmierung der RS232/UART/USART-Peripherie erfolgt wahlweise direkt auf die Register oder über die HAL-Library von ST [10].

Die Register der Peripherie sind in [11] ausführlich beschrieben. Die wichtigsten Register sind:

Register	CMSIS	Funktionsbeschreibung
USART control register x	USARTx→CRx	Aktiviert verschiedene Funktionen und Interrupts
USART baud rate register	USARTx→BRR	Definition der Baudrate
USART interrupt and status register	USARTx→ISR	Interrupt Status
USART interrupt flag clear register	USARTx→ICR	Zurücksetzen der Interrupts
USART receive data register	USARTx→RDR	Daten lesen
USART transmit data register	USARTx→TDR	Daten schreiben

Tabelle 15.4.: Übersicht Register UART-Controller

Für die Programmierung mit der HAL-Library von ST werden die folgenden Funktionen häufig verwendet:

Funktions-Prototyp	Funktion
HAL_StatusTypeDef HAL_UART_Init(UART_HandleTypeDef * huart))	Initialisiert die USART-Peripherie
HAL_StatusTypeDef HAL_UART_Transmit(UART_HandleTypeDef * huart, uint8_t * pData, uint16_t Size, uint32_t Timeout))	Daten senden
HAL_StatusTypeDef HAL_UART_Receive(UART_HandleTypeDef * huart, uint8_t * pData, uint16_t Size, uint32_t Timeout))	Daten empfangen

Tabelle 15.5.: Übersicht HAL-Funktionen USART-Controller

Beispiel:

```
// define handle for uart peripheral no 1
UART_HandleTypeDef huart1;
...

// initialize uart peripheral no 1
huart1.Instance = USART1;
huart1.Init.BaudRate = 115200;
huart1.Init.WordLength = UART_WORDLENGTH_8B;
huart1.Init.StopBits = UART_STOPBITS_1;
huart1.Init.Parity = UART_PARITY_NONE;
huart1.Init.Mode = UART_MODE_TX_RX;
huart1.Init.HwFlowCtl = UART_HWCONTROL_NONE;
huart1.Init.OverSampling = UART_OVERSAMPLING_16;
huart1.Init.OneBitSampling = UART_ONE_BIT_SAMPLE_DISABLE;
huart1.Init.ClockPrescaler = UART_PRESCALER_DIV1;
huart1.AdvancedInit.AdvFeatureInit = UART_ADVFEATURE_NO_INIT;
if (HAL_UART_Init(&huart1) != HAL_OK)
{
    // Add error code implementation
}

...
// define a buffer for a log message
```

```
char cBuffer[100]; /* buffer for sprintf for log msg */
...
/* prepare log message */
// send log message
HAL_UART_Transmit(&huart1, (uint8_t *) cBuffer, strlen(cBuffer), 10);
```

Listing 15.1: Codebeispiel HAL-Library von ST für die RS232

16. I2C

16.1. Einleitung

Der I2C-Bus (Inter-IC Bus) wurde von der Firma Philips entwickelt und ist heute ein viel verwendeter Standard um Peripherie-Elemente wie EEPROM, LCD oder RTC's mit einem Microcontroller zu verbinden. Der Bus wird vorwiegend für die Kommunikation auf einem Print oder zwischen mehreren Prints im selben Gehäuse verwendet.

I2C ist ein Zweidrahtbus, was den Hardwareaufwand sehr reduziert. Er besteht aus einer Takteleitung SCL (Serial Clock) und einer Datenleitung SDA (Serial Data). Die Kommunikation ist wegen der Takteleitung synchron. Der Bus wird im Master-Slave Mode betrieben. Es wird auch Multimasterfähigkeit unterstützt, jedoch existiert pro Zeiteinheit nur eine logische Verbindung.

Ein mögliches Beispiel für eine I2C-Bus Applikation zeigt Abbildung 16.1:

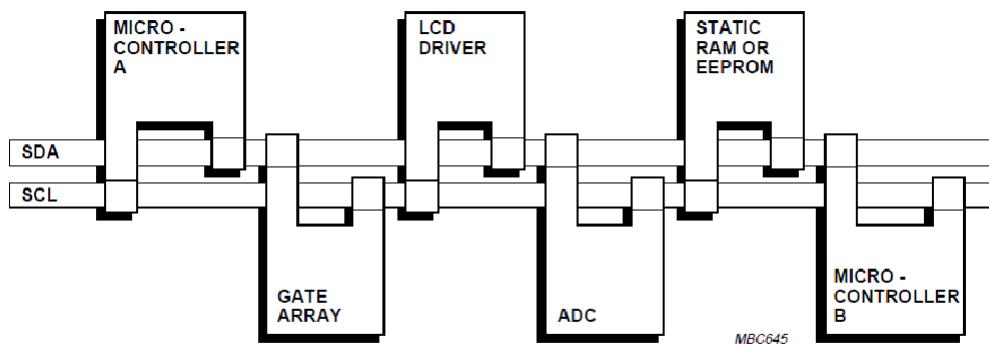


Abbildung 16.1.: Applikation mit einem I2C-Bus, Quelle: [8]

16.2. Busprotokoll

Das Busprotokoll ist wie folgt definiert:

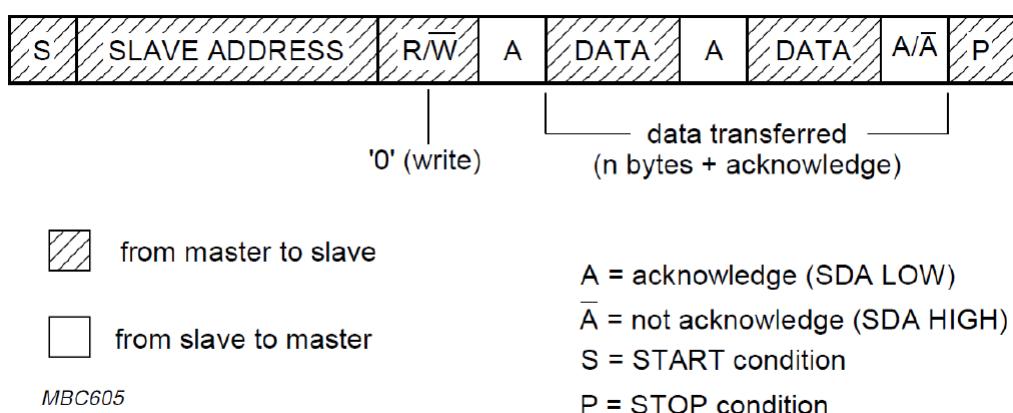


Abbildung 16.2.: Beispiel Protokoll mit Slaveadresse und 2 Byte Daten, Quelle: [8]

Die Kommunikation zwischen Master und Slave läuft gemäss Abbildung 16.2 wie folgt ab:

1. Der Master startet die Übertragung, indem er die sogenannte **Start-Condition** sendet. Dadurch werden alle Slaves am Bus in einen passiven Abhörzustand versetzt.
2. Der Master sendet anschliessend die **Adresse** des Slaves, mit dem er kommunizieren will. Die Adresse besteht aus 7 Bits.
3. Danach wird vom Master das **R/W-Bit** gesendet. Durch dieses Bit wird die Richtung der zu übertragenden Daten festgelegt: Write vom Master zum Slave oder Read vom Slave zum Master.
4. Das I2C-Protokoll sieht vor, dass jeweils nach der Übermittlung von 8 Bits ein **Acknowledge** (Bestätigung) stattfinden muss. In diesem ersten Acknowledge-Bit bestätigt der Slave dem Master, dass er bereit ist für den Datenaustausch.
5. Nun findet der Datenaustausch zwischen Master und Slave statt. Auch hier wird jeweils nach 8 **Datenbits** ein Acknowledge ausgetauscht. Im Write-Mode erfolgt dieses vom Slave, im Read-Mode vom Master.
6. Die Übertragung wird mit einer **Stop-Condition** abgeschlossen.

Je nach verwendetem Baustein ist die Anzahl der übermittelten Daten und die Datenrichtung unterschiedlich. Dies ist aus den jeweiligen Datenblättern zu entnehmen. Bei einem 8-Bit D/A-Wandler schreibt der Master nur ein Byte. Bei einem EEPROM, welches einen internen Adressraum besitzt, wird mit einem Offset zuerst die Adresse für den folgenden Speicherzugriff definiert. Anschliessend folgt der Datentransfer bis die Stop-Condition die Übertragung abschliesst.

Je nachdem, ob der Master lesen oder schreiben will, und wie viele Datenbytes übertragen werden, sehen die Protokolle etwas unterschiedlich aus. Einige Beispiele sind in Tabelle 16.1 aufgeführt. Das Acknowledge ist in diesen Beispielen nicht aufgeführt, um die Darstellung zu vereinfachen.

Beispiel	Protokoll									
	Start	Slave-Adresse	Wr	Daten	Stop					
D/A schreiben										
EEPROM schreiben	Start	Slave-Adresse	Wr	Offset	Daten	Stop				
EEPROM lesen	Start	Slave-Adresse	Wr	Offset	Stop	Start	Slave-Adresse	Rd	Daten	Stop

Tabelle 16.1.: Beispiele I2C-Protokolle

16.3. Bitübertragung

Auf Bitebene ist das I2C-Protokoll wie folgt definiert:

Ruhepegel:

Der Ruhepegel ist High.

Start-Condition:

Die Start-Condition ist als fallende Flanke auf SDA, gefolgt von einer fallenden Flanke auf SCL, definiert. Dieses Signal ist eindeutig und kommt während der normalen Datenübertragung nicht vor.

Stop-Condition:

Die Stop-Condition ist eine steigende Flanke von SCL, gefolgt von einer steigenden Flanke von SDA. Auch dieses Signal ist wieder eindeutig und kommt während der normalen Datenübertragung nicht vor.

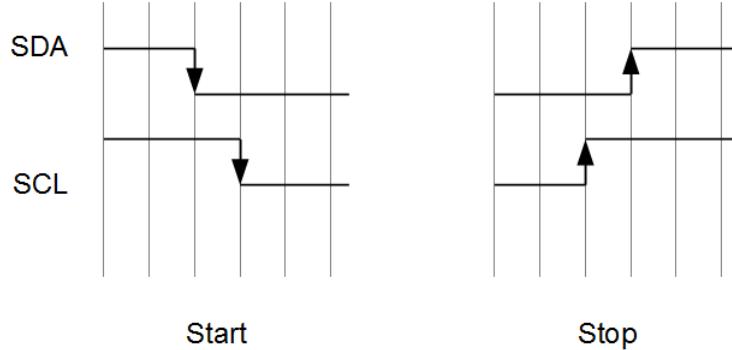


Abbildung 16.3.: Start- und Stop-Condition

Datenbit:

SDA wird entsprechend dem Bitwert gesetzt: 0 = low, 1 = high. Anschliessend wird die Validität mit einer steigenden SCL Flanke signalisiert. Das Lesen und Schreiben von Datenbits geschieht immer während SCL high ist.

ACK:

Das Acknowledge-Bit deutet korrekten Empfang bzw. Empfangsbereitschaft für ein weiteres 8-Bit-Wort an. Aus Sicht des Masters werden geschriebene Bytes vom Slave bestätigt. Andererseits muss der Master die gelesenen Bytes bestätigen, sofern er noch weitere empfangen will. Ansonsten wird der Lesevorgang mit einer Stop-Condition beendet.

NACK:

Not Acknowledged zeigt einen Übertragungsfehler oder ein Ende der Empfangsbereitschaft an.

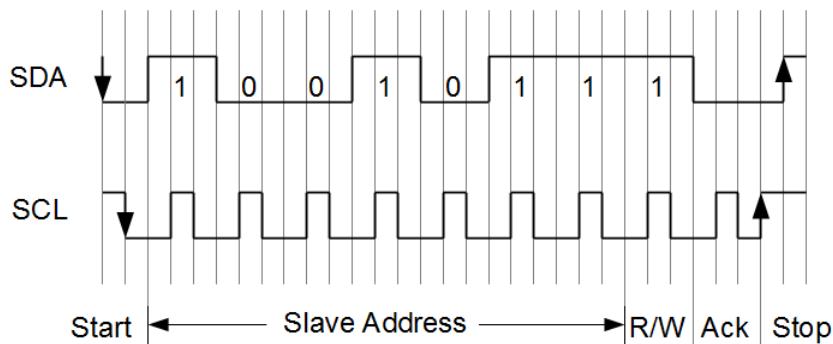


Abbildung 16.4.: Übertragung auf Bitebene

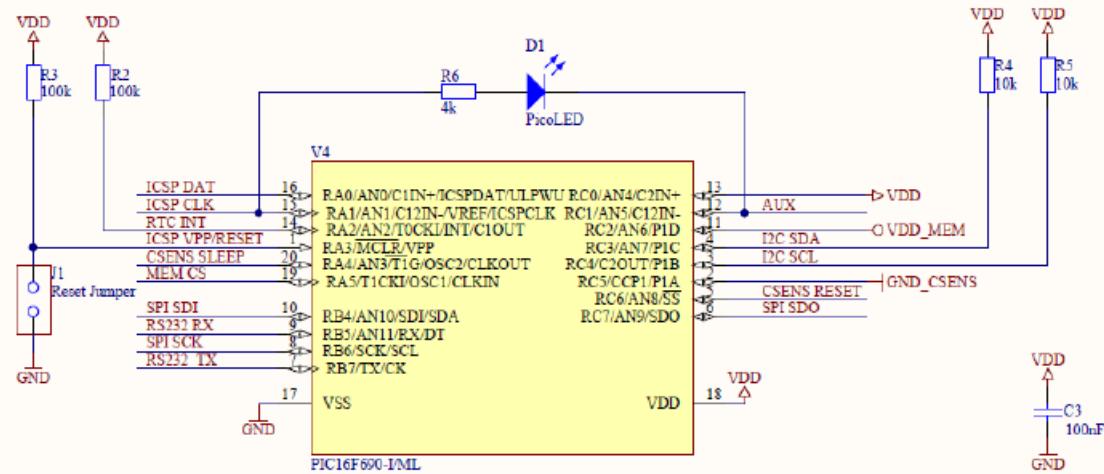
In Abbildung 16.4 sendet der Master als Erstes die Start-Condition und anschliessend wird die Slave-Adresse übertragen (1001011). Darauf folgt das R/W-Bit. Der Slave muss nun mit dem ACK-Bit seine Anwesenheit bestätigen. Wichtig ist, dass für die Dauer des ACK-Bits der Master sein SDA-Pin hochohmig schalten muss! Nach diesem ACK-Bit können Daten ausgetauscht werden, was in der Grafik nicht dargestellt wird. Am Schluss wird die Übertragung durch die Stop-Condition abgeschlossen. Da auf dem I2C-Bus mehrere Teilnehmer angeschlossen sind, aber nur ein Sender je Zeiteinheit erlaubt ist, muss unbedingt darauf geachtet werden, dass alle übrigen Teilnehmer im hochohmigen Zustand sind (Buskonflikte)! Einige Microcontroller können ihre Ausgänge Tristate schalten. Ist dies nicht möglich, so müssen die entsprechende Pins als Eingänge definiert werden.

16.4. Anwendungs-Beispiel

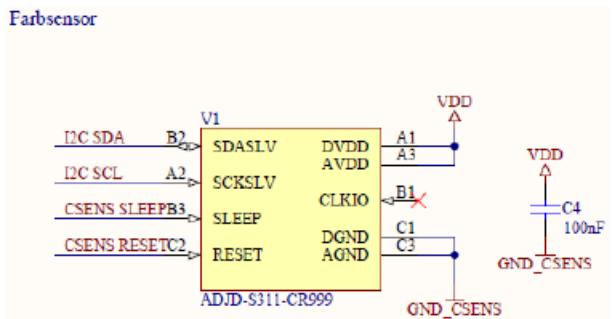
Als Beispiel für einen I2C-Bus ist in Abbildung 16.5 der Geodatenlogger Version 1 abgebildet, den die BFH für die Schweizerische Vogelwarte entwickelt hat. Der Geodatenlogger zeichnet den Verlauf des Tageslichtes über ein Jahr auf. Anhand der Tagesmittelpunkte und der Tageslängen können die Position (geografische Länge und Breite) für

jeden Tag berechnet werden. So kann nach der Rückkehr der Zugvögel bestimmt werden, über welche Route diese nach Süden gezogen sind und wo sie überwintert haben. In der Abbildung ist zu sehen, dass der Microcontroller über einen I2C-Bus mit dem Farbsensor und der RTC kommuniziert.

Mikrokontroller



Farbsensor



Echtzeituhr (RTC)

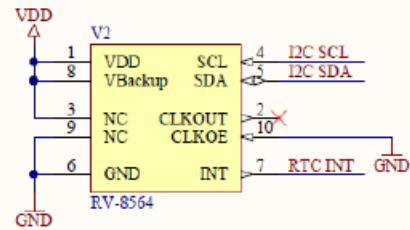


Abbildung 16.5.: Geodatenlogger, Kommunikation Microcontroller, Farbsensor und RTC über I2C

16.5. I2C-Peripherie auf dem STM32H7xx

Der STM32H7xx unterstützt vier I2C-Busse. Die dazu notwendige Peripherie ist auf dem Silizium des STM32H7xx integriert. Abbildung 16.6 zeigt ein Blockschaltbild dieser Peripherie:

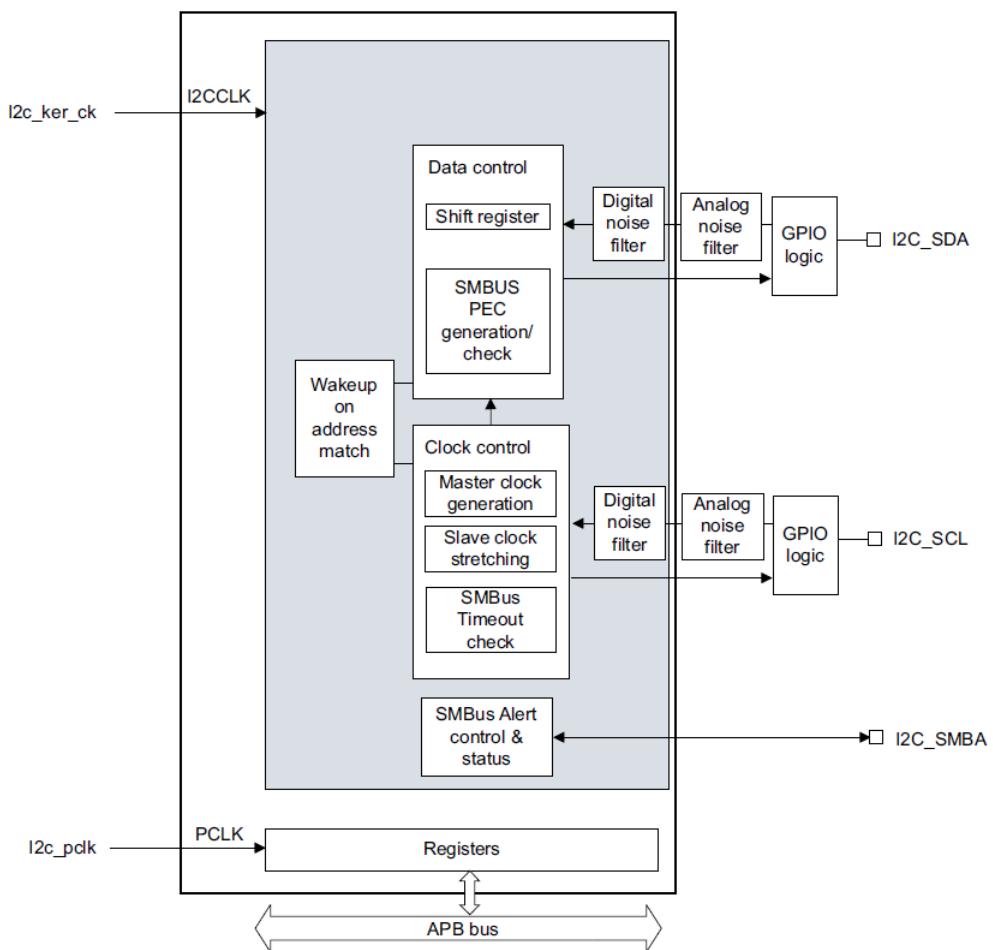


Abbildung 16.6.: I2C-Peripherie auf dem STM32H7xx, Quelle [11]

16.6. Programmierung der I2C-Peripherie auf dem STM32H7xx

Die Programmierung der I2C-Peripherie erfolgt wahlweise direkt auf die Register oder über die HAL-Library von ST [10].

Die Register der I2C-Peripherie sind in [11] ausführlich beschrieben. Die wichtigsten Register sind:

Register	CMSIS	Funktion
I2C control register 1	I2Cx→CR1	Aktiviert verschiedene Funktionen und Interrupts
I2C control register 2	I2Cx→CR2	Verschiedene Definitionen
I2C interrupt and status register	I2Cx→ISR	Interrupt Status
I2C interrupt clear register	I2Cx→ICR	Zurücksetzen der Interrupts
I2C receive data register	I2Cx→RXDR	Daten lesen
I2C transmit data register	I2Cx→TXDR	Daten schreiben

Tabelle 16.2.: Übersicht Register I2C-Controller

Für die Programmierung mit der HAL-Library von ST werden die folgenden Funktionen häufig verwendet:

Funktions-Prototyp	Funktionsbeschreibung
HAL_StatusTypeDef HAL_I2C_Init (I2C_HandleTypeDef * hi2c)	Initialisiert die I2C-Peripherie
HAL_StatusTypeDef HAL_I2C_Master_Transmit (I2C_HandleTypeDef * hi2c, uint16_t DevAddress, uint8_t * pData, uint16_t Size, uint32_t Timeout)	Daten senden (Master)
HAL_StatusTypeDef HAL_I2C_Master_Receive (I2C_HandleTypeDef * hi2c, uint16_t DevAddress, uint8_t * pData, uint16_t Size, uint32_t Timeout)	Daten empfangen (Master)

Tabelle 16.3.: Übersicht HAL-Funktionen I2C-Controller

Beispiel:

```
// define handle for i2c peripheral no 2
I2C_HandleTypeDef hi2c2;
...

// initialize i2c peripheral no 2
hi2c2.Instance = I2C2;
hi2c2.Init.Timing = 0x00B03FDB;
hi2c2.Init.OwnAddress1 = 0;
hi2c2.Init.AddressingMode = I2C_ADDRESSINGMODE_7BIT;
hi2c2.Init.DualAddressMode = I2C_DUALADDRESS_DISABLE;
hi2c2.Init.OwnAddress2 = 0;
hi2c2.Init.OwnAddress2Masks = I2C_OA2_NOMASK;
hi2c2.Init.GeneralCallMode = I2C_GENERALCALL_DISABLE;
hi2c2.Init.NoStretchMode = I2C_NOSTRETCH_DISABLE;
if (HAL_I2C_Init(&hi2c2) != HAL_OK)
{
    // Add error code implementation
}
...

// send 2 data bytes
uint8_t buf[2];
HAL_StatusTypeDef ret;
uint16_t address = 0x17;
buf[0] = 0;
buf[1] = 1;
ret = HAL_I2C_Master_Transmit(&hi2c2, address, buf, 2, 200);
if (ret == HAL_OK) {
    ...
}
```

Listing 16.1: Codebeispiel HAL-Library von ST für den I2C-Bus

17. SPI

17.1. Einleitung

SPI (Serial Peripheral Interface) ist ein synchrones, serielles High-Speed Bussystem für die Kommunikation zwischen Master und Slave. In Embedded Systems wird SPI oft für die Datenübertragung zwischen Microcontroller und Speicher (Flash, EEPROM) oder Peripherie-Bauelementen (I/O-Erweiterungen, Displays sowie A/D- und D/A-Wandlern) verwendet. SPI kann aber auch für die Kommunikation zwischen zwei Microcontrollern eingesetzt werden. Der Master (Microcontroller) ist für die Erzeugung des Takts und der Steuersignale verantwortlich, selektiert den Slave und startet die Datenübertragung.

17.2. Bussignale

SPI besteht aus zwei Datenleitungen (MOSI und MISO) sowie zwei Steuersignalen (SS und SCK). Die Slaves können entweder parallel verbunden werden (d.h. alle Ausgänge MISO der Slaves werden zusammengefasst und auf den MISO-Eingang des Masters geführt) oder seriell verbunden werden (der Ausgang eines Slaves MISO wird mit dem Eingang des nächsten Slaves MOSI verbunden, der Ausgang des letzten Slaves führt auf den MISO-Eingang des Masters).

Durch die Verwendung von zwei Datenleitungen (MOSI und MISO) arbeitet SPI vollduplex. In Abbildung 17.1 wird der Aufbau eines SPI-Busses mit einem Master und drei Slaves dargestellt. Die Slaves sind hier parallel verdrahtet.

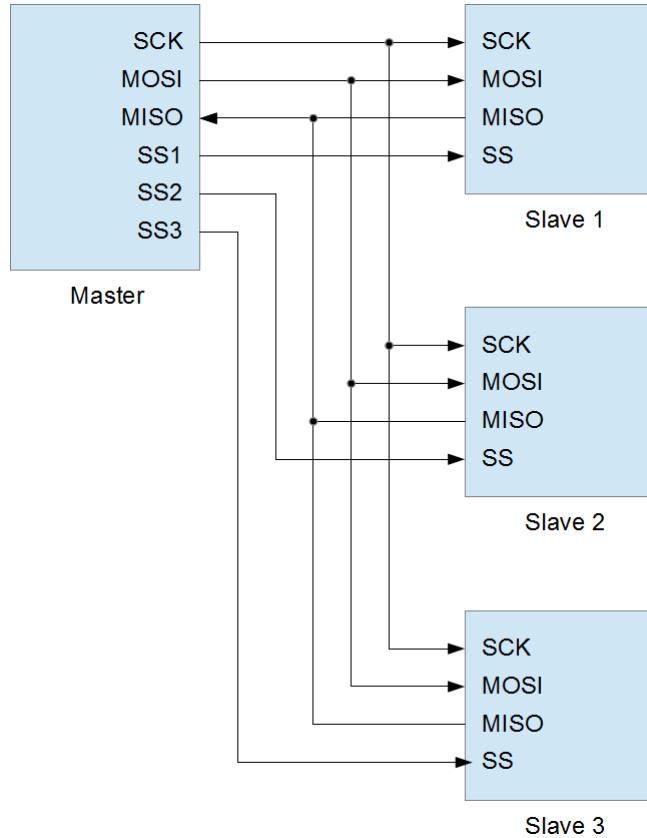


Abbildung 17.1.: SPI Bus mit einem Master und drei parallelen Slaves

SCK (Signal Clock):

Das Taktsignal SCK wird vom Master erzeugt. Die Slaves können auf dieses Signal synchronisieren.

SS (Slave Select):

Die Slaves werden vom Master mit Hilfe der SS Signalleitung ausgewählt. Der Slave wird angesprochen, indem seine Slave-Select Leitung vom Master auf Masse gezogen wird. Die maximale Anzahl von Slaves ist durch die Anzahl der Slave-Select Anschlüsse des Masters begrenzt. Bei einigen Herstellern wird dieses Signal auch Chip-Select genannt.

MOSI (Master Out Slave In):

Auf der Datenleitung MOSI werden die Daten vom Master zu einem Slave übertragen. Bei einigen Bausteinen wird diese Datenleitung auch SDO (Serial/SPI Data Out) genannt, siehe Beispiel Abbildung 17.3.

MISO (Master In Slave Out):

Auf der Datenleitung MISO werden die Daten von einem Slave zum Master übertragen. Bei einigen Bausteinen wird diese Datenleitung auch SDI (Serial/SPI Data In) genannt, siehe Beispiel Abbildung 17.3.

17.3. Interner Aufbau

Der Datentransfer findet über Schieberegister in Master und Slave statt. Während jedem Taktzyklus schreibt der Master ein Datenbit auf die MOSI-Leitung. Der Slave liest dieses Bit an seinem MOSI-Eingang ein, schreibt aber gleichzeitig ebenfalls ein Datenbit auf die MISO-Leitung. Somit entspricht der Aufbau des SPI-Busses einem grossen Schieberegister.

Der Microcontroller schreibt seine zu sendenden Daten in einen Transmit-Buffer, wo sie in einem Schieberegister serialisiert werden (ähnlich RS232). Die empfangenen Daten werden ebenfalls von einem Schieberegister aufbereitet und der Microcontroller kann die Daten parallel aus einem Receive-Buffer auslesen. Weil die SPI-Schnittstelle oft

sehr hoch getaktet wird (bis einige MBit/sec.) werden Transmit- und Receive-Buffer als FIFO für mehrere Bytes ausgelegt.

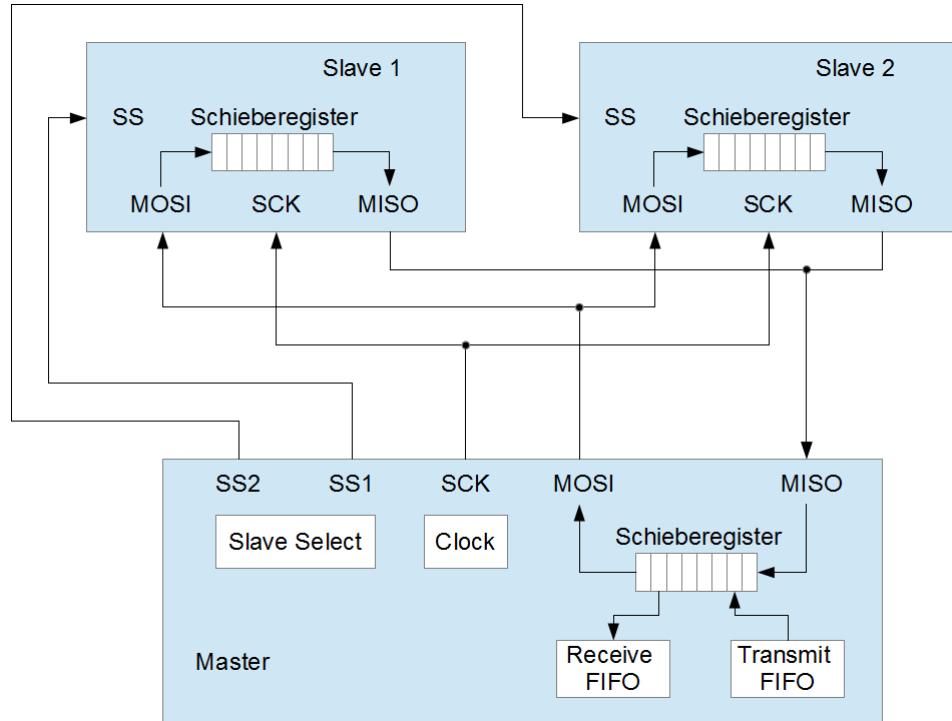
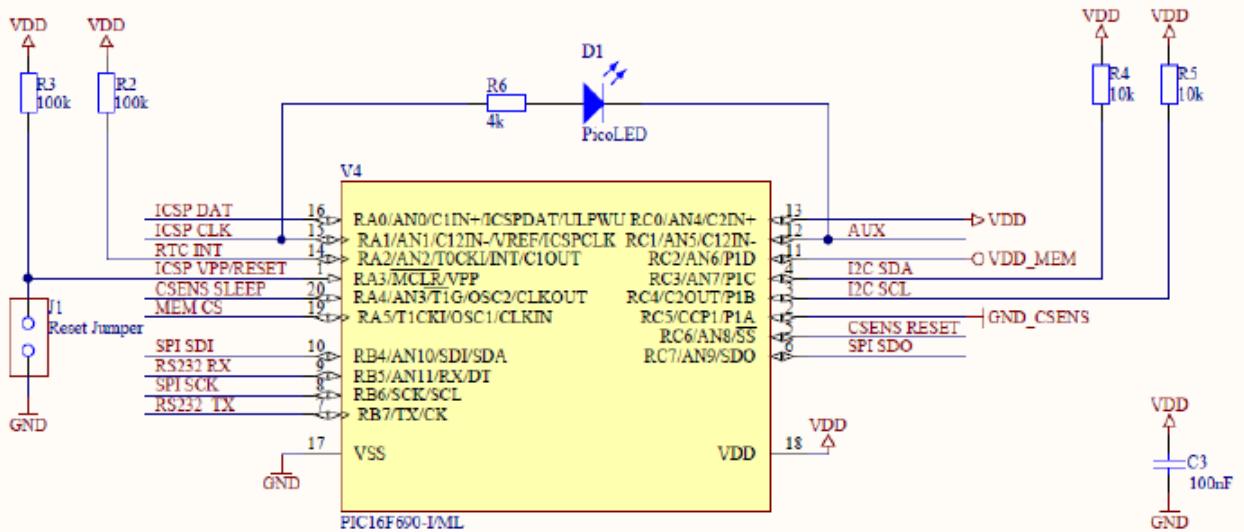


Abbildung 17.2.: SPI Schieberegister, ein Master und zwei Slaves

17.4. Beispiel

Als Beispiel für einen SPI-Bus ist in Abbildung 17.3 wieder ein Teil des Schaltplans des Geodatenloggers Version 1 abgebildet (siehe auch Kapitel I2C, Abbildung 16.5). Beim Geodatenlogger kommuniziert der Microcontroller mit dem Flash-Speicher über den SPI-Bus. Dies hat gegenüber einem Adress- / Datenbus den Vorteil, dass wesentlich weniger Leitungen auf dem Layout vorzusehen sind.

Mikrokontroller



Flash Speicher

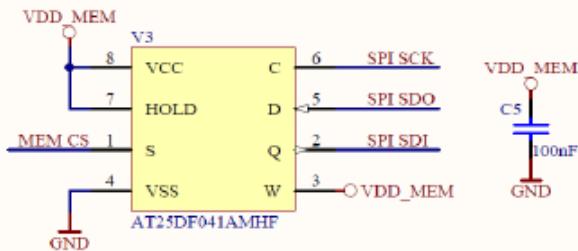


Abbildung 17.3.: Geodatenlogger, Kommunikation Microcontroller und Flash über SPI

17.5. Programmierung der SPI-Peripherie auf dem STM32H7xx

Die Programmierung der SPI-Peripherie erfolgt wahlweise direkt auf die Register oder über die HAL-Library von ST [10].

Die Register der SPI-Peripherie sind in [11] ausführlich beschrieben. Die wichtigsten Register sind:

Register	CMSIS	Funktion
SPI control register x	SPIx→CRx	Aktiviert verschiedene Funktionen
SPI configuration register x	SPIx→CFGx	Verschiedene Konfigurationen
SPI interrupt enable register	SPIx→IER	Interrupts aktivieren
SPI status register	SPIx→SR	Angaben zum Status
SPI interrupt flag clear register	SPIx→IFCR	Interrupts löschen
SPI receive data register	SPIx→RXDR	Daten lesen
SPI transmit data register	SPIx→TXDR	Daten schreiben

Tabelle 17.1.: Übersicht Register SPI-Controller

Für die Programmierung mit der HAL-Library von ST werden die folgenden Funktionen häufig verwendet:

Funktions-Prototyp	Funktionsbeschreibung
HAL_StatusTypeDef HAL_SPI_Init (SPI_HandleTypeDef * hspi)	Initialisiert die SPI-Peripherie
HAL_StatusTypeDef HAL_SPI_Transmit (SPI_HandleTypeDef * hspi, uint8_t * pData, uint16_t Size, uint32_t Timeout)	Daten senden
HAL_StatusTypeDef HAL_SPI_Receive (SPI_HandleTypeDef * hspi, uint8_t * pData, uint16_t Size, uint32_t Timeout)	Daten senden
HAL_StatusTypeDef HAL_SPI_TransmitReceive (SPI_HandleTypeDef * hspi, uint8_t * pTxData, uint8_t * pRxData, uint16_t Size, uint32_t Timeout)	Daten senden und empfangen

Tabelle 17.2.: Übersicht HAL-Funktionen SPI-Controller

Beispiel:

```
// define handle for spi peripheral no 6
SPI_HandleTypeDef hspi6;
...

// initialize spi peripheral no 6
hspi6.Instance = SPI6;
hspi6.Init.Mode = SPI_MODE_MASTER;
hspi6.Init.Direction = SPI_DIRECTION_2LINES;
hspi6.Init.DataSize = SPI_DATASIZE_8BIT;
hspi6.Init.CLKPolarity = SPI_POLARITY_LOW;
hspi6.Init.CLKPhase = SPI_PHASE_1EDGE;
hspi6.Init.NSS = SPI_NSS_SOFT;
hspi6.Init.BaudRatePrescaler = SPI_BAUDRATEPRESCALER_8;
hspi6.Init.FirstBit = SPI_FIRSTBIT_MSB;
hspi6.Init.TIMode = SPI_TIMODE_DISABLE;
hspi6.Init.CRCCalculation = SPI_CRCALCULATION_DISABLE;
hspi6.Init.CRCPolynomial = 0x0;
hspi6.Init.NSSPMode = SPI_NSS_PULSE_DISABLE;
hspi6.Init.NSSPolarity = SPI_NSS_POLARITY_LOW;
hspi6.Init.FifoThreshold = SPI_FIFO_THRESHOLD_01DATA;
hspi6.Init.TxCRCInitializationPattern = SPI_CRC_INITIALIZATION_ALL_ZERO_PATTERN;
hspi6.Init.RxCRCInitializationPattern = SPI_CRC_INITIALIZATION_ALL_ZERO_PATTERN;
hspi6.Init.MasterSSIDleness = SPI_MASTER_SS_IDLENESS_00CYCLE;
hspi6.Init.MasterInterDataIdleness = SPI_MASTER_INTERDATA_IDLENESS_00CYCLE;
hspi6.Init.MasterReceiverAutoSusp = SPI_MASTER_RX_AUTOSUSP_DISABLE;
hspi6.Init.MasterKeepIOState = SPI_MASTER_KEEP_IO_STATE_DISABLE;
hspi6.Init.IOSwap = SPI_IO_SWAP_DISABLE;
if (HAL_SPI_Init(&hspi6) != HAL_OK)
```

```
{  
    // Add error code implementation  
}  
...  
  
// access motion sensor on leguan board  
#define SPIMotionWhoAmI      (0x75)  
#define SPIMotionDeviceId    (0x42)  
  
uint8_t buf[2];           // transmit buffer to send to sensor  
buf[0] = SPIMotionWhoAmI | 0x80; // first transmit byte is WHO_AM_I, MSB (read) is set  
uint8_t rbuf[2];          // receive buffer  
rbuf[0] = 0x55;  
HAL_StatusTypeDef ret;     // ret value of SPI datatransfer  
  
HAL_GPIO_WritePin(GPIOB, GPIO_PIN_2, GPIO_PIN_RESET);      // enable chipselect for Motion  
    sensor  
ret = HAL_SPI_TransmitReceive(&hspi6, buf, rbuf, 2, 200); // transmit SPI data  
HAL_GPIO_WritePin(GPIOB, GPIO_PIN_2, GPIO_PIN_SET);        // disable chipselect for Motion  
    sensor
```

Listing 17.1: Codebeispiel HAL-Library von ST für den SPI-Bus

18. Timer, Counter, PWM, Watchdog

18.1. Einleitung

Auf sehr vielen Microcontrollern sind heute ein oder mehrere Timer integriert. Sie werden hauptsächlich als Zeitbasis entweder für die CPU (Betriebssystem, Uhr ...) oder für externe Komponenten (PWM-Signale, Baudrate für serielle Schnittstellen ...) verwendet. Mit zusätzlichen Funktionen wie Compare/Capture-Einheiten werden die Möglichkeiten noch erweitert. Dieses Kapitel vermittelt einen Überblick zu diesem Thema.

Die Einsatzmöglichkeiten von Timern sind:

- Zeitbasis (z.B. System-Clock für Betriebssystem)
- Clock-Generation (z.B. Baudraten für serielle Kommunikation)
- Zähler von externen HW-Ereignissen
- PWM-Generation
- Zeitmessungen

18.2. Timer/Counter

In der Regel werden 8-Bit oder 16-Bit-Timer/Counter eingesetzt. Im Timer-Betrieb wird durch einen Oszillator der Inhalt des Timer-Registers periodisch inkrementiert. Im Counter-Betrieb werden die Register infolge eines externen Signals inkrementiert.

Beim Überlauf des Registers (d.h. bei einem 16-Bit Timer der Wechsel von 0xFFFF auf 0) wird ein Overflow-Flag gesetzt, welches entweder durch Pollen in der SW abgefragt werden kann, oder welches einen Interrupt auslöst.

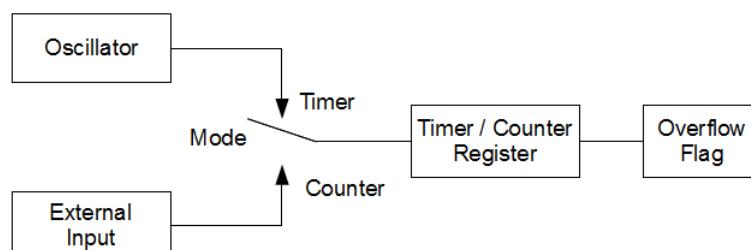


Abbildung 18.1.: Prinzipieller Aufbau eines Timer/Counter

Bemerkung: Bei einigen Controllern wird der Inhalt des Timer-Registers dekrementiert statt inkrementiert. Das Overflow-Flag wird beim Wechsel von 0 auf 0xFFFF gesetzt.

18.3. Timer mit Autoreload-Funktion

Um einen Clock mit definierter Periodendauer zu erzeugen werden Timer mit Autoreload-Funktion eingesetzt.

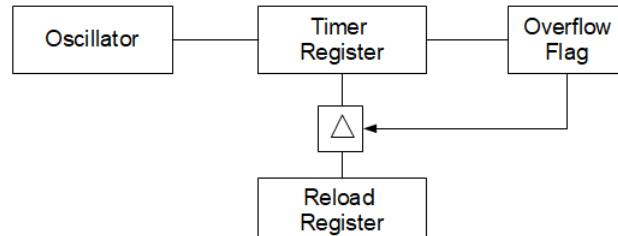


Abbildung 18.2.: Timer mit Autoreload-Mode

Im Autoreload-Mode wird bei einem Timer-Overflow nicht nur das Overflow-Flag gesetzt, sondern gleichzeitig auch der Timer mit dem im Reload-Register gespeicherten Wert neu geladen. Die Periodendauer für einen 16-bit Timer berechnet sich wie folgt:

$$t_{period} = \frac{(0xFFFF - \text{Reload_Value})}{f_{osc}}$$

18.4. Compare-Einheit, PWM

Bei einer Compare-Einheit vergleicht ein Hardware-Comparator den Wert im Timer-Register mit dem Wert in einem Compare-Register.

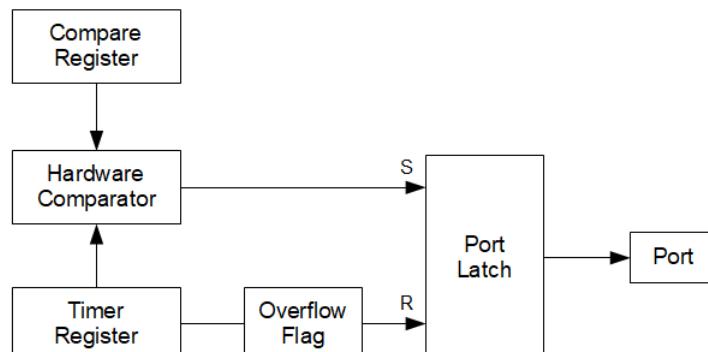


Abbildung 18.3.: Compare-Einheit

Mit Hilfe einer Compare-Einheit können PWM-Signale (Pulse Width Modulation) erzeugt werden, ohne Rechenleistung der CPU zu beanspruchen. Das Ausgangssignal des Comparators sowie das Timer-Overflowflag werden verwendet, um ein Ausgangs-Port (Pin des Microcontrollers) anzusteuern. Bei einem Timer-Overflow wird das Latch zurückgesetzt und der Port wird gelöscht. Wenn der Timer den Wert des Compare-Registers erreicht, wird das Latch durch den Comparator gesetzt und damit auch der Port. Dies wird in der nächsten Abbildung verdeutlicht:

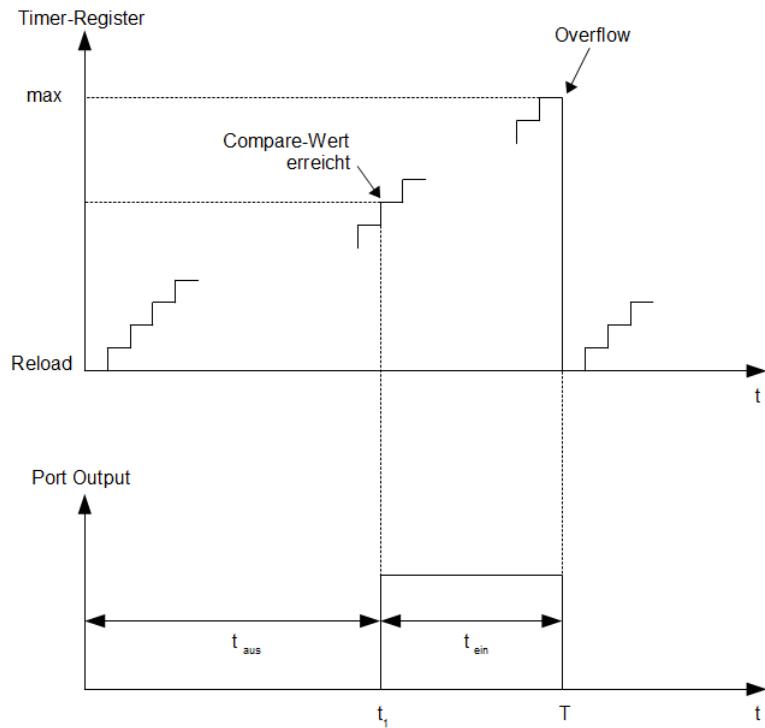


Abbildung 18.4.: Compare-Mode

Die Funktionalität in Abbildung 18.4 basiert auf einem Timer, der im Count-Up Mode betrieben wird. Der Duty-Cycle des PWM-Signals (d.h. die Zeit, die der Ausgang hoch ist (t_{ein}) im Verhältnis zur Periodendauer ($T = t_{\text{aus}} + t_{\text{ein}}$), z.B. 30% Duty-Cycle heisst, das Signal ist 30% der Zeit hoch) kann mit Hilfe des Compare-Wertes definiert werden. Die Periodendauer wird mit Hilfe des Reload-Wertes (und der Timer-Taktrate) definiert. Als Variante kann auch ein Timer im Count-Down Mode verwendet werden. Verglichen wird dann der Wert des Timers mit Null und beim Nulldurchgang wird der Timer mit einem Reload-Wert geladen.

18.5. Capture-Einheit

Mit Hilfe von Capture-Einheiten können Zeitmessungen durchgeführt werden (ähnlich einer Stoppuhr). Im Capture-Mode werden durch ein Trigger-Signal die Werte des Timer-Registers in ein Capture-Register kopiert. Das Trigger-Signal kann entweder SW-mässig oder durch ein externes HW-Signal ausgelöst werden.

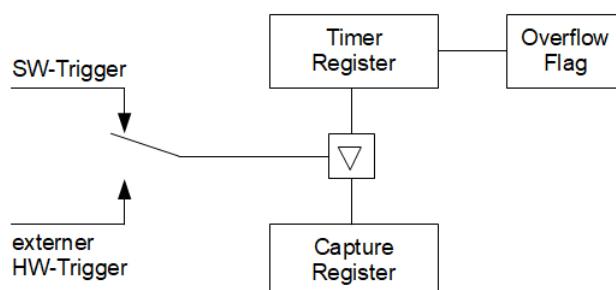


Abbildung 18.5.: Capture-Mode

18.6. Watchdog

18.6.1. Einführung

Die Aufgabe des Watchdogs besteht darin, einen Microcontroller aus einem undefinierten Zustand mittels Reset wieder in einen definierten Zustand zu bringen. Der undefinierte Zustand wird oft durch fehlerhaften Programmcode erreicht, z.B. durch eine Endlosschleife. Im Volksmund spricht man in diesem Falle von einem Programmabsturz.

Der Einsatz eines Watchdogs ist bei allen Systemen (also auch bei sehr einfachen Applikationen) sehr empfehlenswert. Deshalb besitzen viele Microcontroller einen integrierten Watchdog.

18.6.2. Controller-Externer Watchdog

Microcontroller-Externe Watchdogs können zusätzlich zur Watchdog-Funktionalität folgende Aufgaben übernehmen:

1. Überwachung der Speisung.
2. Überwachung des System-Clocks.

Bei der Speisung sind kurze Spannungsunterbrüche gefährlich, da sie das System in einen undefinierten Zustand führen können. Externe Watchdogs überprüfen deshalb auch die Speisung und lösen bei Problemen einen Reset aus. Diese Funktion kann aber auch von Reset-Bausteinen übernommen werden.

Externe Watchdogs haben in der Regel eine eigene Zeitbasis. Dadurch können sie ebenfalls den Systemtakt überwachen.

Eine typische externe Watchdogschaltung ist in Abbildung 18.6 dargestellt:

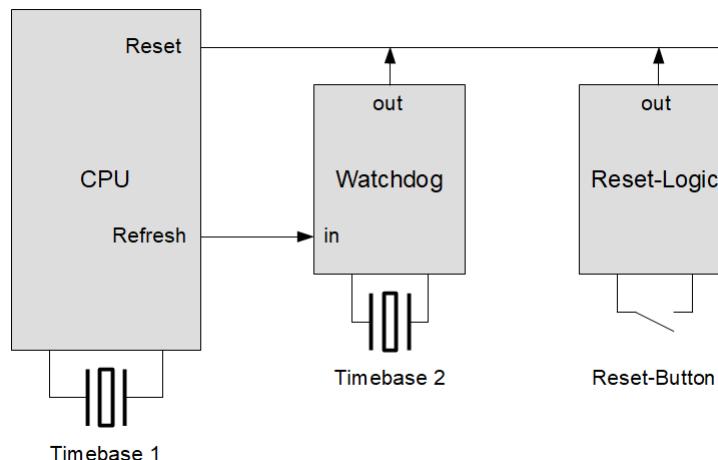


Abbildung 18.6.: Externer Watchdog

18.6.3. Controller-Interner Watchdog

Ein im Microcontroller integrierter Watchdog führt im Fehlerfall ebenfalls einen Reset durch. Der so ausgelöste Reset ist von einem extern angelegten Reset oft nicht zu unterscheiden.

Je nach Controller existiert für den internen Watchdog eine eigene Zeitbasis, die oft auf einem RC-Glied basiert.

18.6.4. Aufbau eines Watchdog

Der Watchdog wird mit einem Timer aufgebaut, welcher periodisch durch die SW zurückgesetzt wird. Wenn dieser Refresh-Zyklus ausbleibt, läuft der Timer ab und löst einen Reset aus.

Abbildung 18.7 zeigt den grundsätzlichen Aufbau eines Watchdogs:

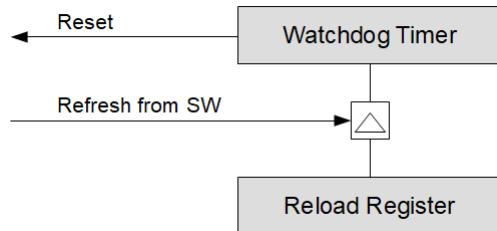


Abbildung 18.7.: Aufbau eines Watchdogs

18.6.5. Starten des Watchdog

Je nach Watchdog kann dieser auf folgende zwei Arten gestartet werden:

1. Hardwaremäßig nach dem Reset. Dies hat den Vorteil, dass das Aufstarten des Microcontrollers schon zu Beginn überwacht wird.
2. Softwaremäßig durch setzen der entsprechenden Kontrollbits in einem Watchdog-Register. Dies sollte möglichst früh in der SW erfolgen, d.h. nicht erst nach der Initialisierung des Systems.

18.6.6. Refresh des Watchdog

Der Refresh erfolgt ebenfalls per SW durch setzen der entsprechenden Flags oder Register. Ganz wichtig ist in diesem Zusammenhang, dass der Zeitpunkt des Refreshs nicht wahllos im Code verstreut liegt, sondern möglichst nur einmal je Programmzyklus durchgeführt wird, also z.B. am Ende der Hauptschleife in main().

Um die Sicherheit des Systems zu erhöhen kann der Refresh abhängig von Schlüsseln erfolgen. Schlüssel können in Kombination mit einem Watchdog verwendet werden um:

- Den zeitlichen Ablauf eines Programms zu kontrollieren.
- Zu prüfen, ob einzelne wichtige Funktionen auch tatsächlich ausgeführt wurden.

Dabei geht man wie folgt vor: Verwendet werden ein oder mehrere Schlüssel. Schlüssel sind Variablen, die den Programmfluss kontrollieren. In jeder Funktion, die für das Programm relevant ist, wird ein Schlüssel mit einem definierten Wert gesetzt. Sollten Sie viele Stellen in Ihrem Programm haben, welche einen Schlüssel setzen, so können Sie anstelle vieler Schlüssel auch nur einen Schlüssel verwenden und dessen Wert jeweils mit Hilfe eines CRC-Polynoms anpassen. Der Wert Ihres Schlüssels am Schluss eines Programmzyklus gibt Ihnen so die Gewissheit, dass alle wichtigen Funktion in der richtigen Reihenfolge ausgeführt wurden. Am Ende eines Programmzyklus prüfen Sie den Schlüssel. Ist er korrekt, setzen Sie den Watchdog zurück und löschen den Schlüssel. Sollte im Programmablauf etwas falsch abgelaufen sein, erhalten Sie einen falschen Schlüssel. Entsprechend wird der Watchdog nicht zurückgesetzt und das System wird durch den Reset neu gestartet.

18.7. Timer auf dem STM32H7xx

Der STM32H7xx stellt diverse Timer zur Verfügung:

- 1 High-Resolution Timer (HRTIM)
- 2 Advanced Control Timer (TIM1 und TIM8)
- 10 General Purpose Timer (TIM2 bis TIM5 und T12 bis T17)
- 2 Basic Timer (TIM6 und TIM7)
- 1 Low-Power Timer (LPTIM)
- 2 Watchdog
- 1 RTC (Real Time Clock)

Das Blockdiagramm der General Purpose Timer TIM2 bis TIM5 ist in Abbildung 18.8 dargestellt.

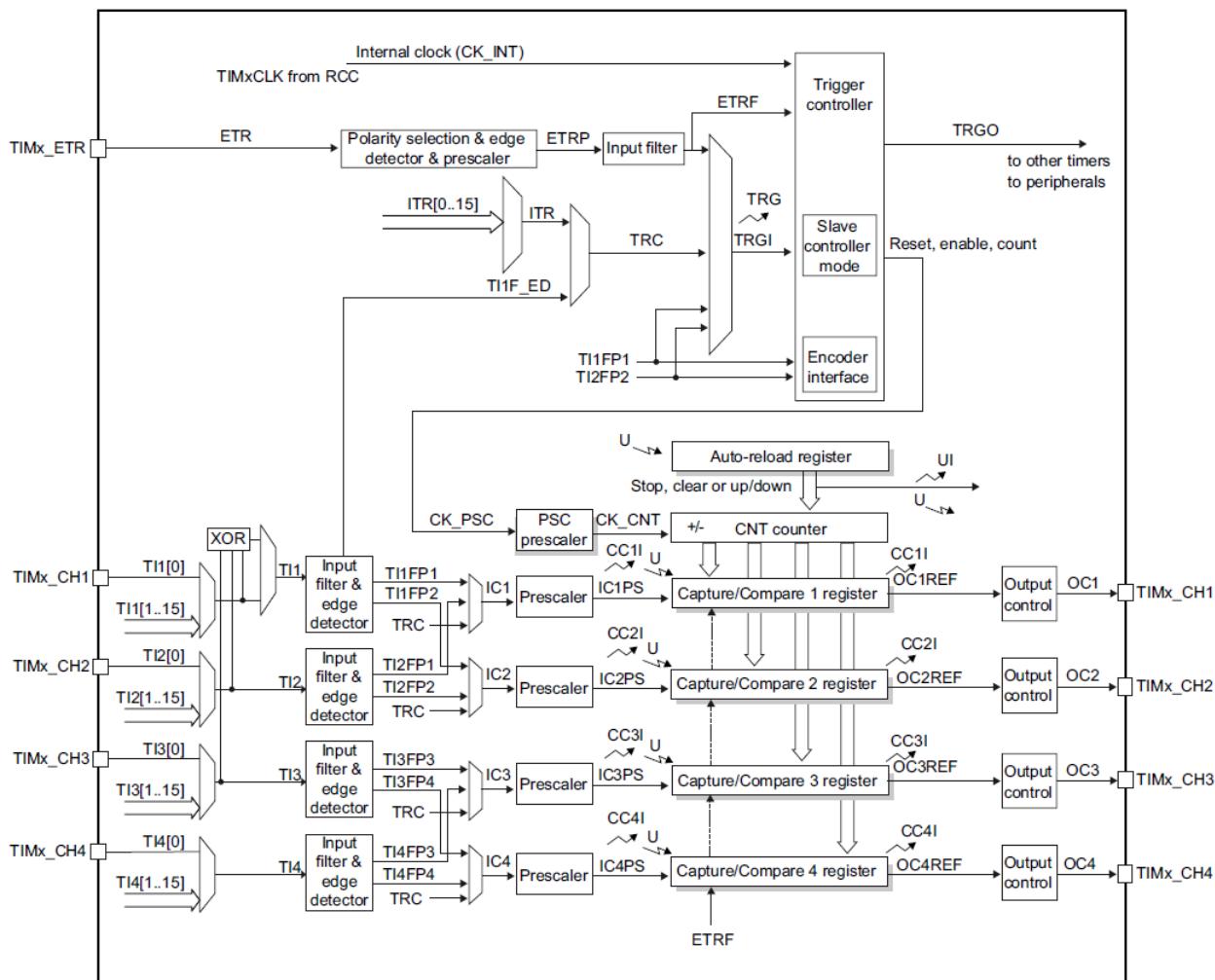


Abbildung 18.8.: Blockschaltbild der General Purpose Timer des STM32H7xx, Quelle: [11]

Beachten Sie im Blochschaltbild insbesondere das Timer-Register (CNT counter), das Autoreload-Register (oberhalb des Timer-Registers) sowie die Compare-Capture-Einheiten (beim STM32H7xx sind dies 4 Einheiten/Channel pro Timer).

18.8. Programmierung der Timer auf dem STM32H7xx

Die Programmierung der Timer erfolgt wahlweise direkt auf die Register oder über die HAL-Library von ST [10].

Die Register der Timer sind in [11] ausführlich beschrieben. Die wichtigsten Register sind:

Register	CMSIS	Funktion
TIM control register x	TIMx→CR1	Aktiviert verschiedene Funktionen
TIM status register	TIMx→SR	Timer Status
TIM capture/compare mode register	TIMx→CCMRx	Definition compare / capture mode
TIM counter register	TIMx→CNT	Counter-Wert
TIM auto reload register	TIMx→ARR	Autoreload-Wert
TIM DMA/Interrupt enable register	TIMx→DIER	DMA und Interrupts enablen

Tabelle 18.1.: Übersicht Register Timer

Für die Programmierung mit der HAL-Library von ST werden die folgenden Funktionen häufig verwendet:

Funktions-Prototyp	Funktion
HAL_StatusTypeDef HAL_TIM_Base_Init (TIM_HandleTypeDef * htim)	Initialisiert den Timer
HAL_StatusTypeDef HAL_TIM_Base_Start (TIM_HandleTypeDef * htim)	Startet den Timer
HAL_StatusTypeDef HAL_TIM_Base_Start_IT (TIM_HandleTypeDef * htim)	Startet den Timer mit Interrupt
HAL_StatusTypeDef HAL_TIM_Base_Stop (TIM_HandleTypeDef * htim)	Stoppt den Timer

Tabelle 18.2.: Übersicht HAL-Funktionen Timer

Beispiel:

```
// define handle for timer 2
TIM_HandleTypeDef htim2;
...

// initialize timer 2
htim2.Instance = TIM2;
htim2.Init.Prescaler = 0;
htim2.Init.CounterMode = TIM_COUNTERMODE_UP;
htim2.Init.Period = 0xFFFFFFFF;
htim2.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
htim2.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_ENABLE;
if (HAL_TIM_Base_Init(&htim2) != HAL_OK)
{
    // Add error code implementation
}
...
// start timer 2, enabling interrupts
HAL_TIM_Base_Start_IT(&htim2);

// timer 2 irq handler
void TIM2_IRQHandler(void)
{
    HAL_TIM_IRQHandler(&htim2);
    ...
}
```

Listing 18.1: Codebeispiel HAL-Library von ST

19. A/D-Wandler

19.1. Einleitung

ADC (Analog-to-Digital Converter, A/D-Wandler) messen eine analoge Spannung am Eingang, vergleichen diese mit einer Referenzspannung und erzeugen als Resultat einen digitalen Wert am Ausgang, welcher der analogen Eingangsspannung entspricht. Die CPU kann diesen Wert entweder über den Adress-/Databus oder über eine serielle Schnittstelle (z.B. SPI, I2S) einlesen. Um die CPU zu entlasten werden die Werte des ADC oft auch mit Hilfe der DMA ausgelesen.

A/D-Wandler haben oft mehrere analoge Eingänge, von welchen jeweils ein Eingang mit Hilfe eines Multiplexers selektiert wird.

Die wichtigsten Charakteristiken eines A/D-Wandlers sind:

- Bitbreite (typische Werte sind 8, 10, 12 oder 16 Bit)
- Wandlungszeit

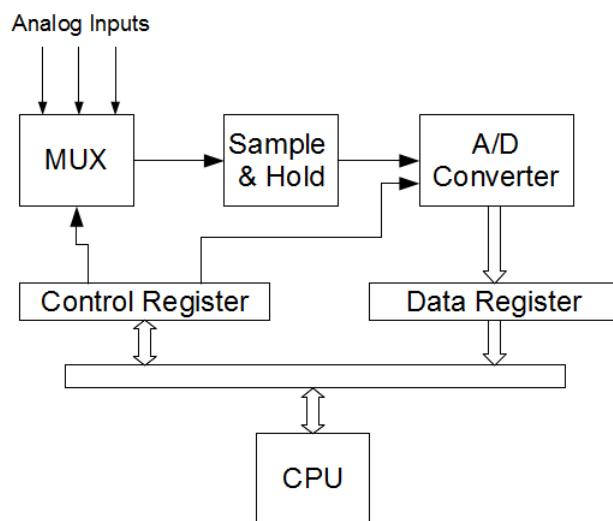


Abbildung 19.1.: Prinzipieller Aufbau eines A/D-Wandlers

19.2. A/D-Wandler des STM32H7xx

Der STM32H7xx besitzt drei A/D-Wandler mit einer Auflösung von je 16 Bit. Jeder A/D-Wandler hat einen Multiplexer für 20 Eingangskanäle. Das Blockschaltbild eines A/D-Wandlers des STM32H7xx ist in Abbildung 19.2 dargestellt.

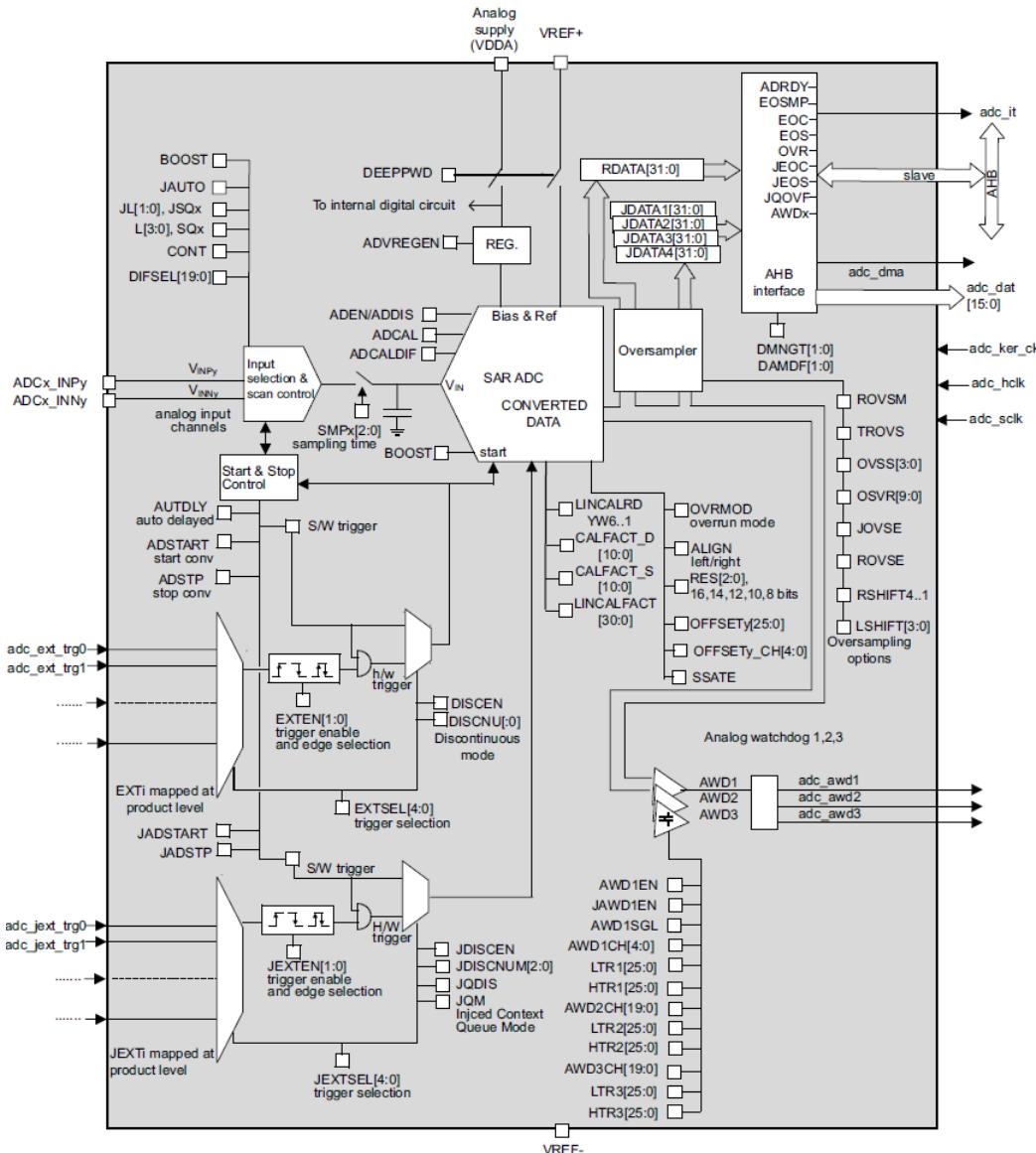


Abbildung 19.2.: Blockschaltbild A/D-Wandler des STM32H7xx, Quelle: [11]

19.3. Programmierung der ADC auf dem STM32H7xx

Die Programmierung der ADC erfolgt wahlweise direkt auf die Register oder über die HAL-Library von ST [10].

Die Register der ADC sind in [11] ausführlich beschrieben. Die wichtigsten Register sind:

Register	CMSIS	Funktion
ADC interrupt and status register	ADCx→ISR	Interrupt und Status
ADC interrupt enable register	ADCx→IER	Interrupts aktivieren
ADC control register	ADCx→CR	Verschiedene Definitionen
ADC configuration register	ADCx→CFGRx	Verschiedene Konfigurationen
ADC regular data register	ADCx→DR	Daten lesen

Tabelle 19.1.: Übersicht Register ADC

Für die Programmierung mit der HAL-Library von ST werden die folgenden Funktionen häufig verwendet:

Funktions-Prototyp	Funktionsbeschreibung
HAL_StatusTypeDef HAL_ADC_Init (ADC_HandleTypeDef * hadc)	Initialisiert den ADC
HAL_StatusTypeDef HAL_ADC_ConfigChannel (ADC_HandleTypeDef * hadc, ADC_ChannelConfTypeDef * sConfig)	Konfiguriert den ADC
HAL_StatusTypeDef HAL_ADC_Start (ADC_HandleTypeDef * hadc)	ADC starten
HAL_StatusTypeDef HAL_ADC_Start_IT (ADC_HandleTypeDef * hadc)	ADC starten, Interrupts aktiv
HAL_StatusTypeDef HAL_ADC_Start_DMA (ADC_HandleTypeDef * hadc, uint32_t * pData, uint32_t Length)	ADC starten, DMA aktiv

Tabelle 19.2.: Übersicht HAL-Funktionen ADC

Beispiel:

```
// define handle for adc1
ADC_HandleTypeDef hadc1;
...

// initialize adc1
hadc1.Instance = ADC1;
hadc1.Init.ClockPrescaler = ADC_CLOCK_ASYNC_DIV1;
hadc1.Init.Resolution = ADC_RESOLUTION_16B;
hadc1.Init.ScanConvMode = ADC_SCAN_DISABLE;
hadc1.Init.EOCSelection = ADC_EOC_SINGLE_CONV;
hadc1.Init.LowPowerAutoWait = DISABLE;
hadc1.Init.ContinuousConvMode = DISABLE;
hadc1.Init.NbrOfConversion = 1;
hadc1.Init.DiscontinuousConvMode = DISABLE;
hadc1.Init.ExternalTrigConv = ADC_SOFTWARE_START;
hadc1.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_NONE;
hadc1.Init.ConversionDataManagement = ADC_CONVERSIONDATA_DR;
hadc1.Init.Overrun = ADC_OVR_DATA_PRESERVED;
hadc1.Init.LeftBitShift = ADC_LEFTBITSHIFT_NONE;
hadc1.Init.OversamplingMode = DISABLE;
if (HAL_ADC_Init(&hadc1) != HAL_OK)
{
    // Add error code implementation
}
...
// configure adc1, channel 2
ADC_ChannelConfTypeDef sConfig = {0};
sConfig.Channel = ADC_CHANNEL_2;
sConfig.Rank = ADC_REGULAR_RANK_1;
sConfig.SamplingTime = ADC_SAMPLETIME_1CYCLE_5;
sConfig.SingleDiff = ADC_SINGLE_ENDED;
sConfig.OffsetNumber = ADC_OFFSET_NONE;
sConfig.Offset = 0;
sConfig.OffsetSignedSaturation = DISABLE;
if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
{
    // Add error code implementation
}

// start adc conversion
if (HAL_ADC_Start(&hadc1) != HAL_OK)
{
    // Add error code implementation
}
```

Listing 19.1: Codebeispiel HAL-Library von ST für den ADC

20. D/A-Wandler

20.1. Einleitung

DAC (Digital-to-Analog Converter, D/A-Wandler) legen an ihrem Ausgang eine analoge Spannung an, welche dem digitalen Eingangswert entspricht.

Die wichtigsten Charakteristiken eines D/A-Wandlers:

- Bitbreite (typische Werte sind 8, 10, 12 oder 16 Bit)
- Wandlungszeit

D/A-Wandler können auch mit einem PWM-Ausgang und einem zusätzlichen Tiefpass-Filter realisiert werden.

20.2. D/A-Wandler des STM32H7xx

Der STM32H7xx besitzt zwei D/A-Wandler mit einer Auflösung von je 12 Bit, zwei Channels pro DAC und 1MHz Wandlungszeit. Das Blockschaltbild eines D/A-Wandlers des STM32H7xx ist in Abbildung 20.1 dargestellt.

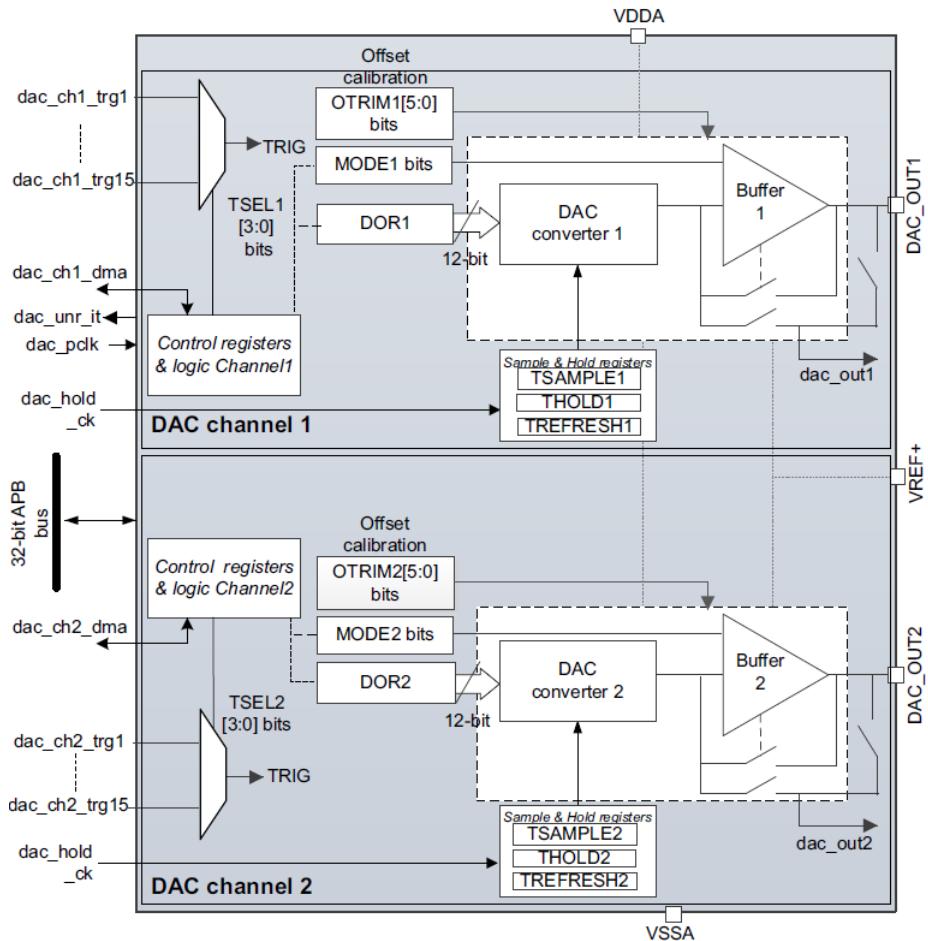


Abbildung 20.1.: Blockschaltbild D/A-Wandler des STM32H7xx, Quelle: [11]

20.3. Programmierung der DAC auf dem STM32H7xx

Die Programmierung der DAC erfolgt wahlweise direkt auf die Register oder über die HAL-Library von ST [10].

Die Register der DAC sind in [11] ausführlich beschrieben. Die wichtigsten Register sind:

Register	CMSIS	Funktion
DAC control register	DACx→CR	Verschiedene Definitionen
DAC status register	DACx→SR	Status des DAC
DAC data register	DACx→DR	Daten schreiben, 12-bit, right-aligned

Tabelle 20.1.: Übersicht Register DAC

Für die Programmierung mit der HAL-Library von ST werden die folgenden Funktionen häufig verwendet:

Funktions-Prototyp	Funktionsbeschreibung
HAL_StatusTypeDef HAL_DAC_Init (DAC_HandleTypeDef * hdac)	Initialisiert den DAC
HAL_StatusTypeDef HAL_DAC_ConfigChannel (DAC_HandleTypeDef * hdac, DAC_ChannelConfTypeDef * sConfig, uint32_t Channel)	Konfiguriert den DAC
HAL_StatusTypeDef HAL_DAC_SetValue (DAC_HandleTypeDef * hdac, uint32_t Channel, uint32_t Alignment, uint32_t Data)	DAC Wert schreiben
HAL_StatusTypeDef HAL_DAC_Start (DAC_HandleTypeDef * hdac, DAC starten uint32_t Channel)	

Tabelle 20.2.: Übersicht HAL-Funktionen DAC

Beispiel:

```
// define handle for dac1
DAC_HandleTypeDef hdac1;
...

// initialize dac1
hdac1.Instance = DAC1;
if (HAL_DAC_Init(&hdac1) != HAL_OK)
{
    // Add error code implementation
}

...
// configure dac1, channel out1
DAC_ChannelConfTypeDef sConfig = {0};
sConfig.DAC_SampleAndHold = DAC_SAMPLEANDHOLD_DISABLE;
sConfig.DAC_Trigger = DAC_TRIGGER_NONE;
sConfig.DAC_OutputBuffer = DAC_OUTPUTBUFFER_ENABLE;
sConfig.DAC_ConnectOnChipPeripheral = DAC_CHIPCONNECT_DISABLE;
sConfig.DAC_UserTrimming = DAC_TRIMMING_FACTORY;
if (HAL_DAC_ConfigChannel(&hdac1, &sConfig, DAC_CHANNEL_1) != HAL_OK)
{
    // Add error code implementation
}

// start dac conversion
if (HAL_DAC_Start(&hdac1, DAC_CHANNEL_1) != HAL_OK)
{
    // Add error code implementation
}
```

Listing 20.1: Codebeispiel HAL-Library von ST für den DAC

Teil IV.

Anhang

A. Informationseinheiten

CPUs verwenden folgende Einheiten zur Darstellung der Information:

Bit (1 Bit): Wertebereich 0..1

Ein Bit ist die kleinste Informationseinheit. Es kann die Zustände 0 und 1 annehmen. Physikalisch können diese Zustände als Spannung (bei Halbleitern), als Licht (Lichtwellenleiter), oder als elektromagnetisches Feld (Speicher) dargestellt werden.



Abbildung A.1.: Bit

Nibble (4 Bit): Wertebereich 0..15

Das Nibble fasst 4 Bits zusammen. Es wird als Informationseinheit insbesondere für die Darstellung von hexadezimalen Zahlen verwendet. Das Nibble hatte früher mit den ersten 4-Bit Mikroprozessoren eine wichtige Bedeutung.



Abbildung A.2.: Nibble

Byte (8 Bit): Wertebereich 0..255

Das Byte ist die wohl wichtigste Informationseinheit. Ein Byte besteht aus 8 Bits. Speichergrößen werden heute in Byte angegeben (kByte, MByte, GByte). Auch die Übertragungsgeschwindigkeit von Kommunikationssystemen wird häufig in Byte pro Sekunde angegeben.



Abbildung A.3.: Byte

Halfword (16 Bit): Wertebereich 0..65535

Ein Halfword besteht aus zwei Bytes, dem „lower Byte“ (Bit 0 bis 7) und dem „upper Byte“ (Bit 8 bis 15). Halfwords werden bei ARM-Cores häufig als informationseinheit verwendet.

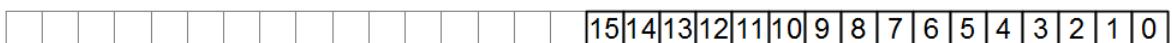


Abbildung A.4.: Halfword

Word (32 Bit): Wertebereich 0.. $2^{32}-1$

Ein Word besteht aus zwei Halfwords, dem „lower Halfword“ (Bit 0 bis 15) und dem „upper Halfword“ (Bit 16 bis 31). Words sind bei vielen CPUs eine wichtige Informationseinheit, weil die Register und die Datenverarbeitung in der ALU häufig in 32 Bit, also als Word, ausgeführt werden.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	---	---	---	---	---	---	---	---	---	---

Abbildung A.5.: Word

Double Word (64 Bit): Wertebereich $0..2^{64}-1$

Ein Double Word besteht aus zwei Words, dem „lower Word“ (Bit 0 bis 31) und dem „upper Word“ (Bit 32 bis 63). Double Words sind bei 64-Bit CPUs die Standard-Informationseinheit. Bei 32-Bit CPUs werden Sie für die Abbildung grosser Zahlenbereiche verwendet, wobei dann zwei Register für die Speicherung des Double Words verwendet werden.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32

Abbildung A.6.: Double Word

LSB und MSB

Das tiefstwertige Bit einer Informationseinheit wird auch LSB (Least Significant Bit) genannt. Das höchstwertige Bit entsprechend MSB (Most Significant Bit).

B. Zahlenformate

B.1. Zweierkomplement

Das Zweierkomplement bietet uns die Möglichkeit, negative Zahlen im Binärsystem darzustellen. Dabei werden keine zusätzlichen Symbole wie + und - benötigt. Positive Zahlen werden im Zweierkomplement mit einer führenden 0 versehen und ansonsten nicht verändert. Negative Zahlen werden mit einer führenden 1 dargestellt und wie folgt codiert: Sämtliche Ziffern der entsprechenden positiven Zahl werden negiert. Zum Ergebnis wird 1 addiert.

Beispiele:

Dezimal	Zweierkomplement
+4	00000100
-4	11111011 + 1 = 11111100
-1	11111111
127	01111111
-128	10000000

Tabelle B.1.: Zweierkomplement

Durch die im Zweierkomplement verwendete Codierung wird erreicht, dass nur eine einzige Darstellung der Null existiert. Addition und Subtraktion benötigen keine Fallunterscheidung, es gibt nur eine Null, wenn man annimmt, dass Überläufe abgeschnitten werden.

Beispiel:

$$-4 + 3 = -1 \text{ führt zu } 11111100 + 00000011 = 11111111;$$

B.2. Floating Point

B.2.1. Single Precision

Floating Point Single Precision wird als 32-Bit Wert wie folgt abgelegt:

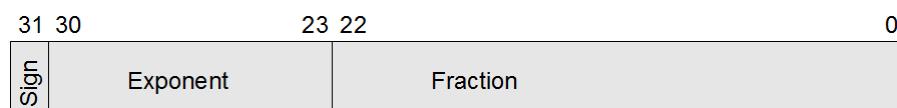


Abbildung B.1.: Floating Point Single Precision Datenformat

Es sind folgende Kombinationen von Exponent, Fraction und Sign möglich:

Exponent	Fraction	Sign	Beschreibung
0	0	0	Positive Null (+0)
0	0	1	Negative Null (-0)
0	not 0	-	denormalisierte Zahl, sehr klein
1 bis 254	-	-	normalisierte Zahl
255	0	0	Positiv Unendlich ($+\infty$)
255	0	1	Negativ Unendlich ($-\infty$)
255	not 0	-	NaN (Not a Number), invalid value

Tabelle B.2.: Floating Point Werte, Single Precision

Umrechnung für normalisierte Single Precision Werte:

$$Value = (-1)^{Sign} * 2^{(Exponent - 127)} * (1 + (\frac{1}{2} * Fraction[22]) + (\frac{1}{4} * Fraction[21]) + \dots + (\frac{1}{2^{23}} * Fraction[0]))$$

Umrechnung für denormalisierte Single Precision Werte:

$$Value = (-1)^{Sign} * 2^{(-126)} * ((\frac{1}{2} * Fraction[22]) + (\frac{1}{4} * Fraction[21]) + \dots + (\frac{1}{2^{23}} * Fraction[0]))$$

B.2.2. Double Precision

Floating Point Double Precision wird als 64-Bit Wert wie folgt abgelegt:

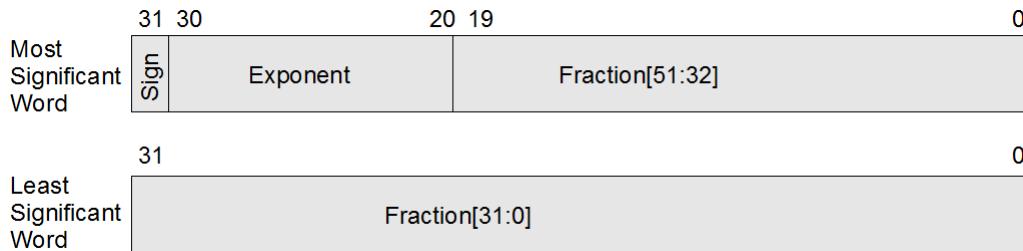


Abbildung B.2.: Floating Point Double Precision Datenformat

Es sind folgende Kombinationen von Exponent, Fraction und Sign möglich:

Exponent	Fraction	Sign	Beschreibung
0	0	0	Positive Null (+0)
0	0	1	Negative Null (-0)
0	not 0	-	denormalisierte Zahl, sehr klein
1 bis 0x7FE	-	-	normalisierte Zahl
0x7FF	0	0	Positiv Unendlich ($+\infty$)
0x7FF	0	1	Negativ Unendlich ($-\infty$)
0x7FF	not 0	-	NaN (Not a Number), invalid value

Tabelle B.3.: Floating Point Werte, Double Precision

Umrechnung für normalisierte Double Precision Werte:

$$Value = (-1)^{Sign} * 2^{(Exponent - 1023)} * (1 + (\frac{1}{2} * Fraction[51]) + (\frac{1}{4} * Fraction[50]) + \dots + (\frac{1}{2^{52}} * Fraction[0]))$$

Umrechnung für denormalisierte Double Precision Werte:

$$Value = (-1)^{Sign} * 2^{(-1022)} * ((\frac{1}{2} * Fraction[51]) + (\frac{1}{4} * Fraction[50]) + \dots + (\frac{1}{2^{52}} * Fraction[0]))$$

C. AAPCS

C.1. Einleitung

Die AAPCS (ARM Architecture Procedure Call Standard) ist eine von ARM definierte Konvention, welche definiert, wie CPU-Register und der Stack verwendet werden sowie welche Datentypen unterstützt werden. Diese Konvention ist notwendig, damit kompilierte oder assemblierte Module später zusammengelinkt werden können.

Die AAPCS muss berücksichtigt werden:

1. Wenn Sie Code schreiben, der sowohl den ARM-Instruktionssatz als auch den Thumb-Instruktionssatz verwendet.
2. Wenn Sie Projekte sowohl mit C als auch mit C++ Code schreiben.
3. Wenn Sie Projekte schreiben, welche auch Assembler-Subroutinen enthalten.

Die AAPCS definiert noch weitere Eigenschaften wie Endianess, Composite Types usw. In diesem Anhang werden nur die wichtigsten Konventionen zusammengefasst. Details finden Sie in [4].

C.2. Aufruf von Subroutinen

Die AAPCS definiert, wie einer Subroutine respektive Funktion Parameter übergeben werden, sodass Subroutinen in unterschiedlichen Sprachen geschrieben werden können und das Zusammenfügen aller Subroutinen in einem Projekt keine Probleme bietet.

Die AAPCS definiert insbesondere:

- Übergabewerte einer Subroutine
- Rückgabewerte einer Subroutine
- Anforderungen an die aufrufende Subroutine
- Anforderungen an die aufgerufene Subroutine

C.2.1. Core Registers

Die Register einer ARM-CPU werden gemäss AAPCS wie folgt verwendet:

Register	AAPCS Synonym	Beschreibung
r0	a1	Argument 1 / Integer-Rückgabewert 32-Bit / Scratch-Register
r1	a2	Argument 2 / Integer-Rückgabewert 64-Bit / Scratch-Register
r2	a3	Argument 3 / Scratch-Register
r3	a4	Argument 4 / Scratch-Register
r4	v1	Variable Register 1
r5	v2	Variable Register 2
r6	v3	Variable Register 3
r7	v4	Variable Register 4
r8	v5	Variable Register 5
r9	v6 / SB / TR	Plattformspezifisches Register / Variable Register 6
r10	v7	Variable Register 7
r11	v8	Variable Register 8
r12	ip	Intra-Procedure Call / Scratch-Register
r13	sp	Stack-Pointer
r14	lr	Link-Register
r15	pc	Program-Counter

Tabelle C.1.: Verwendung der ARM CPU-Register gemäss AAPCS

Die Register r0 bis r3 werden für die Parameterübergabe an Subroutinen sowie für die Rückgabewerte verwendet. Diese Register können durch die aufgerufene Subroutine auch als Zwischenspeicher verwendet und dadurch modifiziert werden (Scratch-Register). Das Register r12 kann vom Linker als Scratch-Register für den Aufruf von Subroutinen verwendet werden. Die aufrufende Subroutine muss also davon ausgehen, dass nach dem Rücksprung aus der aufgerufenen Subroutine die Register r0 bis r3 sowie r12 verändert wurden.

r4 bis r8, sowie r10 und r11 können in einer Subroutine verwendet werden, um lokale Variablen zu speichern. Es gilt aber das Verursacherprinzip, d.h. dass die Register auf dem Stack gerettet und am Schluss der Subroutine wieder hergestellt werden müssen.

r9 wird plattformspezifisch verwendet. D.h. jede Plattform kann diesem Register eine spezielle Verwendung zuweisen. Gibt es keine spezielle Verwendung, so kann r9 als Variable Register eingesetzt werden.

Die Register r13 bis r15 haben spezielle Eigenschaften: r13 wird als Stack-Pointer verwendet (SP), r14 als Link-Register (LR) zum Speichern der Rücksprungadresse aus Subroutinen und r15 als Program-Counter (PC).

C.2.2. Co-Prozessor Registers

Durch den Einsatz von Co-Prozessoren werden zusätzliche Register eingeführt. Typisches Beispiel bei ARM sind die Register der FPU. Diese Register werden nicht für den Aufruf von Subroutinen verwendet (Parameter-Passing, siehe Kapitel C.2.1), können jedoch innerhalb von Subroutinen modifiziert werden.

Die FPU des ARM-Cortex-M7 hat zweiunddreissig 32-bit single-precision Register, s0 bis s31. Dies können auch als sechzehn double-precision Register verwendet werden ([2]). Für diese Register ist folgendes definiert:

- Die Register s0 bis s15 können innerhalb einer Subroutine modifiziert werden, ohne sie zu retten.
- Die Register s16 bis s31 müssen bei Subroutinen-Aufrufen erhalten bleiben. Werden sie in einer Subroutine verwendet, müssen sie zuerst auf dem Stack gerettet und am Schluss wieder hergestellt werden.

C.3. Verwendung des Stacks

Der Stack ist ein Bereich im Datenspeicher, welcher wie folgt verwendet wird:

1. Übergabe von zusätzlichen Parametern an Subroutinen, welche nicht über Register übergeben werden.
2. Speicherbereich von lokalen Variablen in Subroutinen.

Die AAPCS definiert, dass der Stack full-descending ist. Dies bedeutet, dass der Stack hin zu kleinen Adressen wächst. Weiter wird der Stack-Pointer (sp, r13) verwendet, um auf den zuletzt geschriebenen Eintrag auf dem Stack zu zeigen.

An den Stack werden folgende Bedingungen gestellt:

- Stack-limit < SP <= stack-base. Der SP liegt immer innerhalb des Stackbereichs, also zwischen der Basisadresse und der Endadresse.
- SP mod 4 = 0. Der SP ist immer word-aligned.
- Der Zugriff auf den Stack (lesen oder schreiben) darf nur im Bereich zwischen SP und stack-base erfolgen. Adressen tiefer als der SP sind zwar im Stack-Bereich, deren Inhalt ist aber nicht definiert.

C.4. Verwendete Datentypen

Tabelle C.2 zeigt die von der AAPCS definierten fundamentalen Datentypen:

Data Type	Machine Type	Byte size	Byte alignment
Integer	unsigned byte	1	1
Integer	signed byte	1	1
Integer	unsigned halfword	2	2
Integer	signed halfword	2	2
Integer	unsigned word	4	4
Integer	signed word	4	4
Integer	unsigned double-word	8	8
Integer	signed double-word	8	8
Floating Point	half precision	2	2
Floating Point	single precision (IEEE754)	4	4
Floating Point	double precision (IEEE754)	8	8
Pointer	data pointer	4	4
Pointer	code pointer	4	4

Tabelle C.2.: Fundamentale Datentypen gemäss AAPCS

D. ASCII Tabelle

Zeichen	Dez	Hex									
NUL	0	0x00	SOH	1	0x01	STX	2	0x02	ETX	3	0x03
EOT	4	0x04	ENQ	5	0x05	ACK	6	0x06	BEL	7	0x07
BS	8	0x08	TAB	9	0x09	LF	10	0x0A	VT	11	0x0B
FF	12	0x0C	CR	13	0x0D	SO	14	0x0E	SI	15	0x0F
DLE	16	0x10	DC1	17	0x11	DC2	18	0x12	DC3	19	0x13
DC4	20	0x14	NAK	21	0x15	SYN	22	0x16	ETB	23	0x17
CAN	24	0x18	EM	25	0x19	SUB	26	0x1A	ESC	27	0x1B
FS	28	0x1C	GS	29	0x1D	RS	30	0x1E	US	31	0x1F
Leerz.	32	0x20	!	33	0x21	"	34	0x22	#	35	0x23
\$	36	0x24	%	37	0x25	&	38	0x26	'	39	0x27
(40	0x28)	41	0x29	*	42	0x2A	+	43	0x2B
,	44	0x2C	-	45	0x2D	.	46	0x2E	/	47	0x2F
0	48	0x30	1	49	0x31	2	50	0x32	3	51	0x33
4	52	0x34	5	53	0x35	6	54	0x36	7	55	0x37
8	56	0x38	9	57	0x39	:	58	0x3A	;	59	0x3B
<	60	0x3C	=	61	0x3D	>	62	0x3E	?	63	0x3F
@	64	0x40	A	65	0x41	B	66	0x42	C	67	0x43
D	68	0x44	E	69	0x45	F	70	0x46	G	71	0x47
H	72	0x48	I	73	0x49	J	74	0x4A	K	75	0x4B
L	76	0x4C	M	77	0x4D	N	78	0x4E	O	79	0x4F
P	80	0x50	Q	81	0x51	R	82	0x52	S	83	0x53
T	84	0x54	U	85	0x55	V	86	0x56	W	87	0x57
X	88	0x58	Y	89	0x59	Z	90	0x5A	[91	0x5B
\	92	0x5C]	93	0x5D	^	94	0x5E	_	95	0x5F
`	96	0x60	a	97	0x61	b	98	0x62	c	99	0x63
d	100	0x64	e	101	0x65	f	102	0x66	g	103	0x67
h	104	0x68	i	105	0x69	j	106	0x6a	k	107	0x6b
l	108	0x6c	m	109	0x6d	n	110	0x6e	o	111	0x6f
p	112	0x70	q	113	0x71	r	114	0x72	s	115	0x73
t	116	0x74	u	117	0x75	v	118	0x76	w	119	0x77
x	120	0x78	y	121	0x79	z	122	0x7a	{	123	0x7b
	124	0x7C	}	125	0x7D	~	126	0x7E	DEL	127	0x7F

Tabelle D.1.: ASCII-Tabelle

E. Befehlsübersicht Cortex-Mx

Dieses Kapitel wurde aus dem Dokument [9] übernommen.

The processor implements a version of the thumb instruction set. Table E.1 lists the supported instructions.

In Table E.1:

- Angle brackets, <>, enclose alternative forms of the operand
- Braces, {}, enclose optional operands
- The operands column is not exhaustive
- Op2 is a flexible second operand that can be either a register or a constant
- Most instructions can use an optional condition code suffix

Mnemonic	Operands	Brief description	Flags
ADC, ADCS	{Rd,} Rn, Op2	Add with carry	N,Z,C,V
ADD, ADDS	{Rd,} Rn, Op2	Add	N,Z,C,V
ADD, ADDW	{Rd,} Rn, #imm12	Add	N,Z,C,V
ADR	Rd, label	Load PC-relative address	-
AND, ANDS	{Rd,} Rn, Op2	Logical AND	N,Z,C
ASR, ASRS	Rd, Rm, <Rs #n>	Arithmetic shift right	N,Z,C
B	label	Branch	-
BFC	Rd, #lsb, #width	Bit field clear	-
BFI	Rd, Rn, #lsb, #width	Bit field insert	-
BIC, BICS	{Rd,} Rn, Op2	Bit clear	N,Z,C
BKPT	#imm	Breakpoint	-
BL	label	Branch with link	-
BLX	Rm	Branch indirect with link	-
BX	Rm	Branch indirect	-
CBNZ	Rn, label	Compare and branch if non zero	-
CBZ	Rn, label	Compare and branch if zero	-
CLREX	-	Clear exclusive	-
CLZ	Rd, Rm	Count leading zeros	-
CMN	Rn, Op2	Compare negative	N,Z,C,V
CMP	Rn, Op2	Compare	N,Z,C,V
CPSID	iflags	Change processor state, disable interrupts	-
CPSIE	iflags	Change processor state, enable interrupts	-
DMB	-	Data memory barrier	-
DSB	-	Data synchronization barrier	-
EOR, EORS	{Rd,} Rn, Op2	Exclusive OR	N,Z,C
ISB	-	Instruction synchronization barrier	-
IT	-	If-then condition block	-
LDM	Rn{!}, reglist	Load multiple registers, increment after	-
LDMDB, LDMEA	Rn{!}, reglist	Load multiple registers, decrement before	-
LDMFD, LDMIA	Rn{!}, reglist	Load multiple registers, increment after	-
LDR	Rt, [Rn, #offset]	Load register with word	-
LDRB, LDRBT	Rt, [Rn, #offset]	Load register with byte	-
LDRD	Rt, Rt2, [Rn, #offset]	Load register with two bytes	-
LDREX	Rt, [Rn, #offset]	Load register exclusive	-
LDREXB	Rt, [Rn]	Load register exclusive with byte	-

Tabelle E.1.: Assembler Befehlsübersicht

Mnemonic	Operands	Brief description	Flags
LDREXH	Rt, [Rn]	Load register exclusive with halfword	-
LDRH, LDRHT	Rt, [Rn, #offset]	Load register with halfword	-
LDRSB, LDRSBT	Rt, [Rn, #offset]	Load register with signed byte	-
LDRSH, LDRSHT	Rt, [Rn, #offset]	Load register with signed halfword	-
LDRT	Rt, [Rn, #offset]	Load register with word	-
LSL, LSLS	Rd, Rm, <Rs #n>	Logical shift left	N,Z,C
LSR, LSRS	Rd, Rm, <Rs #n>	Logical shift right	N,Z,C
MLA	Rd, Rn, Rm, Ra	Multiply with accumulate, 32-bit result	-
MLS	Rd, Rn, Rm, Ra	Multiply and subtract, 32-bit result	-
MOV, MOVS	Rd, Op2	Move	N,Z,C
MOVT	Rd, #imm16	Move top	-
MOVW, MOV	Rd, #imm16	Move 16-bit constant	N,Z,C
MRS	Rd, spec_reg	Move from special register to general register	-
MSR	spec_reg, Rm	Move from general register to special register	N,Z,C,V
MUL, MULS	{Rd,} Rn, Rm	Multiply, 32-bit result	N,Z
MVN, MVNS	Rd, Op2	Move NOT	N,Z,C
NOP	-	No operation	-
ORN, ORNS	{Rd,} Rn, Op2	Logical OR NOT	N,Z,C
ORR, ORRS	{Rd,} Rn, Op2	Logical OR	N,Z,C
PKHTB, PKHBT	{Rd,} Rn, Rm, Op2	Pack Halfword	-
POP	reglist	Pop registers from stack	-
PUSH	reglist	Push registers onto stack	-
QADD	{Rd,} Rn, Rm	Saturating double and add	-
QADD16	{Rd,} Rn, Rm	Saturating add 16	-
QADD8	{Rd,} Rn, Rm	Saturating add 8	-
QASX	{Rd,} Rn, Rm	Saturating add and subtract with exchange	-
QDADD	{Rd,} Rn, Rm	Saturating add	-
QDSUB	{Rd,} Rn, Rm	Saturating double and subtract	-
QSAX	{Rd,} Rn, Rm	Saturating subtract and add with exchange	-
QSUB	{Rd,} Rn, Rm	Saturating subtract	-
QSUB16	{Rd,} Rn, Rm	Saturating subtract 16	-
QSUB8	{Rd,} Rn, Rm	Saturating subtract 8	-
RBIT	Rd, Rn	Reverse bits	-
REV	Rd, Rn	Reverse byte order in a word	-
REV16	Rd, Rn	Reverse byte order in each halfword	-
REVSH	Rd, Rn	Reverse byte order in bottom halfword and sign extend	-
ROR, RORS	Rd, Rm, <Rs #n>	Rotate right	N,Z,C
RRX, RRXS	Rd, Rm	Rotate right with extend	N,Z,C
RSB, RSBS	{Rd,} Rn, Op2	Reverse subtract	N,Z,C,V
SADD16	{Rd,} Rn, Rm	Signed add 16	-
SADD8	{Rd,} Rn, Rm	Signed add 8	-
SASX	{Rd,} Rn, Rm	Signed add and subtract with exchange	-
SBC, SBCS	{Rd,} Rn, Op2	Subtract with carry	N,Z,C,V
SBFX	Rd, Rn, #lsb, #width	Signed bit field extract	-
SDIV	{Rd,} Rn, Rm	Signed divide	-
SEV	-	Send event	-
SHADD16	{Rd,} Rn, Rm	Signed halving add 16	-
SHADD8	{Rd,} Rn, Rm	Signed halving add 8	-
SHASX	{Rd,} Rn, Rm	Signed halving add and subtract with exchange	-
SHSAX	{Rd,} Rn, Rm	Signed halving subtract and add with exchange	-
SHSUB16	{Rd,} Rn, Rm	Signed halving subtract 16	-
SHSUB8	{Rd,} Rn, Rm	Signed halving subtract 8	-

Tabelle E.1.: Assembler Befehlsübersicht

Mnemonic	Operands	Brief description	Flags
SMLABB, SMLABT, SMLATB, SMLATT	Rd, Rn, Rm, Ra	Signed multiply accumulate long (halfwords)	Q
SMLAD, SMLADX	Rd, Rn, Rm, Ra	Signed multiply accumulate dual	Q
SMLAL	RdLo, RdHi, Rn, Rm	Signed multiply with accumulate (32 x 32 + 64), 64-bit result	-
SMLALBB, SMLALBT, SMLALTB, SMLALTT	RdLo, RdHi, Rn, Rm	Signed multiply accumulate long, halfwords	-
SMLALD, SMLALDX	RdLo, RdHi, Rn, Rm	Signed multiply accumulate long dual	-
SMLAWB, SMLAWT	Rd, Rn, Rm, Ra	Signed multiply accumulate, word by halfword	Q
SMLSD	Rd, Rn, Rm, Ra	Signed multiply subtract dual	Q
SMLS LD	RdLo, RdHi, Rn, Rm	Signed multiply subtract long dual	-
SMMLA	Rd, Rn, Rm, Ra	Signed most significant word multiply accumulate	-
SMMLS, SMMLR	Rd, Rn, Rm, Ra	Signed most significant word multiply subtract	-
SMMUL, SMMULR	{Rd,} Rn, Rm	Signed most significant word multiply	-
SMUAD	{Rd,} Rn, Rm	Signed dual multiply add	Q
SMULBB, SMULBT, SMULTB, SMULTT	{Rd,} Rn, Rm	Signed multiply (halfwords)	-
SMULL	RdLo, RdHi, Rn, Rm	Signed multiply (32 x 32), 64-bit result	-
SSAT	Rd, #n, Rm {,shift #s}	Signed saturate	Q
SSAT16	Rd, #n, Rm	Signed saturate 16	Q
SSAX	{Rd,} Rn, Rm	Signed subtract and add with exchange	GE
SSUB16	{Rd,} Rn, Rm	Signed subtract 16	-
SSUB8	{Rd,} Rn, Rm	Signed subtract 8	-
STM	Rn{!}, reglist	Store multiple registers, increment after	-
STMDB, STMEA	Rn{!}, reglist	Store multiple registers, decrement before	-
STMFD, STMIA	Rn{!}, reglist	Store multiple registers, increment after	-
STR	Rt, [Rn, #offset]	Store register word	-
STRB, STRBT	Rt, [Rn, #offset]	Store register byte	-
STRD	Rt, Rt2, [Rn, #offset]	Store register two words	-
STREX	Rd, Rt, [Rn, #offset]	Store register exclusive	-
STREXB	Rd, Rt, [Rn]	Store register exclusive byte	-
STREXH	Rd, Rt, [Rn]	Store register exclusive halfword	-
STRH, STRHT	Rt, [Rn, #offset]	Store register halfword	-
STRT	Rt, [Rn, #offset]	Store register word	-
SUB, SUBS	{Rd,} Rn, Op2	Subtract	N,Z,C,V
SUB, SUBW	{Rd,} Rn, #imm12	Subtract	N,Z,C,V
SVC	#imm	Supervisor call	-
SXTAB	{Rd,} Rn, Rm,{,ROR#}	Extend 8 bits to 32 and add	-
SXTAB16	{Rd,} Rn, Rm,{,ROR#}	Dual extend 8 bits to 16 and add	-
SXTAH	{Rd,} Rn, Rm,{,ROR#}	Extend 16 bits to 32 and add	-
SXTB16	{Rd,} Rm {,ROR #n}	Signed extend byte 16	-
SXTB	{Rd,} Rm {,ROR #n}	Sign extend a byte	-
SXTH	{Rd,} Rm {,ROR #n}	Sign extend a halfword	-
TBB	[Rn, Rm]	Table branch byte	-
TBH	[Rn, Rm, LSL # 1]	Table branch halfword	-
TEQ	Rn, Op2	Test equivalence	N,Z,C
TST	Rn, Op2	Test	N,Z,C
UADD16	{Rd,} Rn, Rm	Unsigned add 16	GE
UADD8	{Rd,} Rn, Rm	Unsigned add 8	GE
USAX	{Rd,} Rn, Rm	Unsigned subtract and add with exchange	GE
UHADD16	{Rd,} Rn, Rm	Unsigned halving add 16	-
UHADD8	{Rd,} Rn, Rm	Unsigned halving add 8	-

Tabelle E.1.: Assembler Befehlsübersicht

Mnemonic	Operands	Brief description	Flags
UHASX	{Rd,} Rn, Rm	Unsigned halving add and subtract with exchange	-
UHSAX	{Rd,} Rn, Rm	Unsigned halving subtract and add with exchange	-
UHSUB16	{Rd,} Rn, Rm	Unsigned halving subtract 16	-
UHSUB8	{Rd,} Rn, Rm	Unsigned halving subtract 8	-
UBFX	Rd, Rn, #lsb, #width	Unsigned bit field extract	-
UDIV	{Rd,} Rn, Rm	Unsigned divide	-
UMAAL	RdLo, RdHi, Rn, Rm	Unsigned multiply accumulate accumulate long ($32 \times 32 + 32 + 32$), 64-bit result	-
UMLAL	RdLo, RdHi, Rn, Rm	Unsigned multiply with accumulate ($32 \times 32 + 64$), 64-bit result	-
UMULL	RdLo, RdHi, Rn, Rm	Unsigned multiply (32×32), 64-bit result	-
UQADD16	{Rd,} Rn, Rm	Unsigned saturating add 16	-
UQADD8	{Rd,} Rn, Rm	Unsigned saturating add 8	-
UQASX	{Rd,} Rn, Rm	Unsigned saturating add and subtract with exchange	-
UQSAX	{Rd,} Rn, Rm	Unsigned saturating subtract and add with exchange	-
UQSUB16	{Rd,} Rn, Rm	Unsigned saturating subtract 16	-
UQSUB8	{Rd,} Rn, Rm	Unsigned saturating subtract 8	-
USAD8	{Rd,} Rn, Rm	Unsigned sum of absolute differences	-
USADA8	{Rd,} Rn, Rm, Ra	Unsigned sum of absolute differences and accumulate	-
USAT	Rd, #n, Rm {,shift #s}	Unsigned saturate	Q
USAT16	Rd, #n, Rm	Unsigned saturate 16	Q
UASX	{Rd,} Rn, Rm	Unsigned add and subtract with exchange	GE
USUB16	{Rd,} Rn, Rm	Unsigned subtract 16	GE
USUB8	{Rd,} Rn, Rm	Unsigned subtract 8	GE
UXTAB	{Rd,} Rn, Rm,{,ROR#}	Rotate, extend 8 bits to 32 and add	-
UXTAB16	{Rd,} Rn, Rm,{,ROR#}	Rotate, dual extend 8 bits to 16 and add	-
UXTAH	{Rd,} Rn, Rm,{,ROR#}	Rotate, unsigned extend and add halfword	-
UXTB	{Rd,} Rm {,ROR #n}	Zero extend a byte	-
UXTB16	{Rd,} Rm {,ROR #n}	Unsigned extend byte 16	-
UXTH	{Rd,} Rm {,ROR #n}	Zero extend a halfword	-
VABS.F32	Sd, Sm	Floating-point absolute	-
VADD.F32	{Sd,} Sn, Sm	Floating-point add	-
VCMP.F32	Sd, <Sm #0.0>	Compare two floating-point registers, or one floating-point register and zero	FPSCR
VCMPE.F32	Sd, <Sm #0.0>	Compare two floating-point registers, or one floating-point register and zero with Invalid Operation check	FPSCR
VCVT.S32.F32	Sd, Sm	Convert between floating-point and integer	-
VCVT.S16.F32	Sd, Sd, #fbits	Convert between floating-point and fixed point	-
VCVTR.S32.F32	Sd, Sm	Convert between floating-point and integer with rounding	-
VCVT<B H>.F32.F16	Sd, Sm	Converts half-precision value to single-precision	-
VCVTT<B T>.F32.F16	Sd, Sm	Converts single-precision register to half-precision	-
VDIV.F32	{Sd,} Sn, Sm	Floating-point divide	-
VFMA.F32	{Sd,} Sn, Sm	Floating-point fused multiply accumulate	-
VFNMA.F32	{Sd,} Sn, Sm	Floating-point fused negate multiply accumulate	-

Tabelle E.1.: Assembler Befehlsübersicht

Mnemonic	Operands	Brief description	Flags
VFMS.F32	{Sd,} Sn, Sm	Floating-point fused multiply subtract	-
VFNMS.F32	{Sd,} Sn, Sm	Floating-point fused negate multiply subtract	-
VLDM.F<32 64>	Rn{!}, list	Load multiple extension registers	-
VLDR.F<32 64>	<Dd Sd>, [Rn]	Load an extension register from memory	-
VLMA.F32	{Sd,} Sn, Sm	Floating-point multiply accumulate	-
VLMS.F32	{Sd,} Sn, Sm	Floating-point multiply subtract	-
VMOV.F32	Sd, #imm	Floating-point move immediate	-
VMOV	Sd, Sm	Floating-point move register	-
VMOV	Sn, Rt	Copy ARM core register to single precision	-
VMOV	Sm, Sm1, Rt, Rt2	Copy 2 ARM core registers to 2 single precision	-
VMOV	Dd[x], Rt	Copy ARM core register to scalar	-
VMOV	Rt, Dn[x]	Copy scalar to ARM core register	-
VMRS	Rt, FPSCR	Move FPSCR to ARM core register or APSR	N,Z,C,V
VMSR	FPSCR, Rt	Move to FPSCR from ARM Core register	FPSCR
VMUL.F32	{Sd,} Sn, Sm	Floating-point multiply	-
VNEG.F32	Sd, Sm	Floating-point negate	-
VNMLA.F32	Sd, Sn, Sm	Floating-point multiply and add	-
VNMLS.F32	Sd, Sn, Sm	Floating-point multiply and subtract	-
VNMUL	{Sd,} Sn, Sm	Floating-point multiply	-
VPOP	list	Pop extension registers	-
VPUSH	list	Push extension registers	-
VSQRT.F32	Sd, Sm	Calculates floating-point square root	-
VSTM	Rn{!}, list	Floating-point register store multiple	-
WFE	-	Wait for event	-
WFI	-	Wait for interrupt	-

Tabelle E.1.: Assembler Befehlsübersicht

F. Kontroll- und Datenstrukturen in Assembler

F.1. Einführung Kontrollstrukturen

Der Ablauf eines Programmes wird mit Hilfe von Kontrollstrukturen gesteuert. Dabei stehen folgende Strukturen zur Verfügung:

- Sequenzen
- Entscheidungen
- Schleifen

Eine Sequenz ist eine Folge von Instruktionen, die ohne weitere Kriterien abgearbeitet wird. Bei Entscheidungen und Schleifen hingegen können Bedingungen den Programmablauf steuern.

Hochsprachen stellen die Kontrollstrukturen bereits in ihrem Sprachumfang zur Verfügung. So z.B. die Sprache C wie folgt:

Entscheidungen / bedingte Anweisungen:

- if-else (Einfachentscheidung)
- switch (Mehrfachentscheidung)

Schleifen:

- while (Schleife mit Vorabprüfung)
- do-while (Schleife mit Endprüfung)
- for (Zählschleife)

In der Assemblerprogrammierung können Kontrollstrukturen ebenfalls angewendet werden. Dadurch wird der Code strukturierter und besser wartbar. Es stehen zwar keine direkten Befehle für Kontrollstrukturen wie in einer Hochsprache zur Verfügung, diese können jedoch durch eine Folge von Instruktionen, sozusagen eine Schablone, nachgebildet werden. Die nachfolgenden Unterkapitel zeigen, wie dies gemacht wird.

F.2. Einfachentscheidung

Bei Einfachentscheidungen wird eine Bedingung (condition) geprüft, die entweder „erfüllt“ (true) oder „nicht erfüllt“ (false) sein kann.

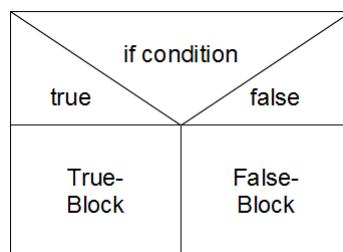


Abbildung F.1.: Einfachentscheidung

Ist die Bedingung erfüllt, so wird der „True-Block“ ausgeführt, sonst der „False-Block“. Der „False-Block“ ist optional, d.h. eine Einfachentscheidung kann auch lediglich den „True-Block“ enthalten. In den Programmiersprachen C und Assembler kann die Einfachentscheidung wie folgt programmiert werden:

C	Assembler
<pre>if (condition) { /* true-Block */ } else { /* false-Block */ }</pre>	<pre>if: CMP <condition> BNE false true: ... B endif @ true-block false: ... B endif @ false-block endif:</pre>

Abbildung F.2.: Einfachentscheidung in C und Assembler

Eine Einfachentscheidung besteht in Assembler immer aus einer einleitenden Instruktion, welche die Statusbits verändert (hier „CMP“) und einer anschliessenden bedingten Verzweigung (hier „BNE“). Als einleitende Instruktion können alle Befehle verwendet werden, welche die Statusbits verändern (TST, TEQ, CMP, MOVS...). Als bedingte Verzweigung können alle „Bcc“ Instruktionen verwendet werden.

Die ARM Cortex-M3, Cortex-M4 und Cortex-M7 Prozessoren unterstützen eine weitere, sehr effiziente Art, um Einfachentscheidungen zu programmieren: Die bedingte Ausführung von Instruktionen (Conditional Execution). Dazu wird die IT (If-Then) Instruktion verwendet. Anschliessend an die IT-Instruktion folgt ein Block mit bis zu vier bedingt ausgeführten Instruktionen. Zur Notation dieser Instruktion werden optional maximal drei Suffixes, entweder „T“ (Then) oder „E“ (Else) angehängt, beispielsweise also ITE (If-Then-Else) oder ITEE (If-Then-Else-Else).

Beispiel

Ermittlung des Maximums von zwei Zahlen in C:

```
if(a > b)
{
    c = a;
}
else
{
    c = b;
}
```

Listing F.1: Bestimmung des Maximalwerts in C

Der Code in Assembler zur Ermittlung des Maximums kann mit Hilfe der IT-Instruktion (in diesem Falle ITE) wie folgt programmiert werden (die Variablen a, b und c sind in den Registern r0, r1 und r2 abgelegt):

```
CMP    r0, r1      @ test if (a > b)
ITE    GT           @ If Then Else block
MOVGT r2, r0      @ if (a > b) then c = a
MOVLE r2, r1      @ else c = b
```

Listing F.2: Bestimmung des Minimalwerts in Assembler

Dieser Code wird durch die CPU sehr effizient ausgeführt, weil keine Sprünge notwendig sind.

F.3. Mehrfachentscheidung

Die Mehrfachentscheidung trifft eine Auswahl unter mehreren Alternativen. Dazu wird ein Ausdruck (expression) mit konstanten, ganzzahligen Werten verglichen. Bei Übereinstimmung wird entsprechend verzweigt. Stimmt der Ausdruck mit keiner der angebotenen Alternativen überein, so wird ein Default-Block ausgeführt.

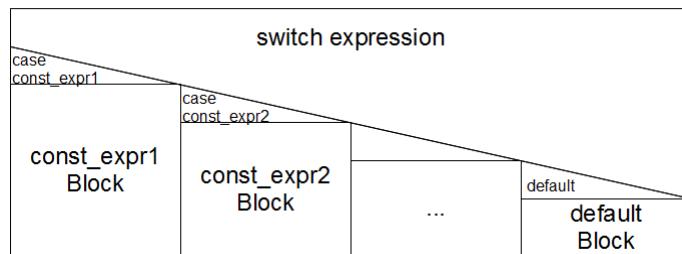


Abbildung F.3.: Mehrfachentscheidung

In den Programmiersprachen C und Assembler kann die Mehrfachentscheidung wie folgt programmiert werden:

C	Assembler
<pre> switch (expression) { case (const-expr1): ... break; case (const-expr2): ... break; ... default: ... break; } </pre>	<pre> switch: MOV r0, #expression CMP r0, #const_expr1 BEQ case1 ... CMP r0, #const_expr2 BEQ case2 ... B default case1: ... B endswitch @ const-expr1 block case2: ... B endswitch @ const-expr2 block default: B endswitch @ default block endswitch: </pre>

Abbildung F.4.: Mehrfachentscheidung in C und Assembler

Bei der Mehrfachentscheidung wird zuerst der zu prüfende Ausdruck (expression) in ein Register kopiert, um anschliessend mit den möglichen Alternativen (const-expr) verglichen zu werden. Bei Übereinstimmung erfolgt ein bedingter Sprung zum entsprechenden Label.

Wenn in der switch-Anweisung viele Werte geprüft werden müssen, kann dies zu langsamem Code führen. Oft werden dann Sprungtabellen (jump table) verwendet, welche wie folgt programmiert werden können:

```

jt_switch: LDR      r1, =jumptable      @ load r1 with base address of jump table
            MOV      r0, #1           @ load r0 with expression to test
            CMP      r0, #tablemax     @ check if value is in jump table
            IT       L0             @ start if-then block
            LDRLO   pc, [r1, r0, LSL #2] @ ok -> load pc with label from jump table
            B       default         @ (expression > tablemax) -> default
jt_case1:  B       endswitch          @ const-expr1 block
jt_case2:  B       endswitch          @ const-expr2 block
jt_default: B       endswitch          @ default block
jt_endswitch:

```

```
jumptable:
    .word    jt_default           @ case 0 not defined -> default
    .word    jt_case1
    .word    jt_case2

.set tablemax, 0x03                  @ jump table has 3 entries
```

Listing F.3: Verwendung von Sprungtabellen in Assembler

Einschränkungen durch die Verwendung von Sprungtabellen:

- Die Tabelle darf nicht zu gross werden. Sie kann nicht alle denkbaren Werte für const-expression (32-Bit Integer) enthalten. Beispielsweise ist eine Sprungtabelle ungeeignet, wenn die Ausdrücke 1 und 10000 geprüft werden müssen. Dies würde eine Tabelle mit 10000 Einträgen benötigen.
- Die Tabellengrösse ist zur Laufzeit zu prüfen (CMP r0, #tablemax)
- Für nicht definierte Werte muss in der Tabelle das default-Label eingetragen werden (im obigen Beispiel existiert „case 0“ nicht, in der Tabelle muss deshalb bei Index 0 das default-Label eingetragen werden).

F.4. Schleife mit Vorabprüfung

Die Schleife mit Vorabprüfung (while-Schleife) steuert Sequenzen, bei denen die Anzahl der Durchläufe nicht konstant oder zur Kompilationszeit nicht bekannt ist.

Zu Beginn wird eine Bedingung geprüft. Ist diese „erfüllt“, so wird der Schleifenkörper ausgeführt und anschliessend die Bedingung erneut geprüft. Ist sie „nicht erfüllt“, wird abgebrochen. Falls die Bedingung schon beim ersten Test „nicht erfüllt“ ist, wird der Schleifenkörper gar nie ausgeführt.

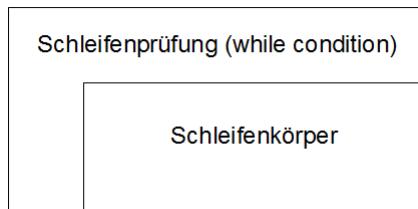


Abbildung F.5.: Schleife mit Vorabprüfung

In den Programmiersprachen C und Assembler kann die Schleife mit Vorabprüfung wie folgt programmiert werden:

C	Assembler
while (condition) { /* Schleifenkörper */ }	while: CMP <condition> BNE endwhile ... B while endwhile: ...

Abbildung F.6.: Schleife mit Vorabprüfung in C und Assembler

Wie schon bei der Einfachentscheidung werden die Statusbits zuerst durch eine einleitende Instruktion (hier CMP) verändert. Anschliessend erfolgt eine bedingte Verzweigung ans Schleifenende, falls die Bedingung „nicht erfüllt“ ist. Dadurch wird auch der Fall erfasst, dass der Schleifenkörper gar nicht ausgeführt wird. Nach Ausführung des Schleifenkörpers erfolgt ein unbedingter Sprung an den Schleifenanfang.

Der Code in Abbildung F.6 ist nicht sehr effizient, weil er zwei Sprungbefehle (B und BNE) enthält, die je Durchlauf ausgeführt werden. Durch Umorganisation des Codes kann dieser optimiert werden, je Durchlauf wird nur noch ein bedingter Sprungbefehl (BEQ) ausgeführt:

Beispiel:

Optimierter Code für die Schleife mit Vorabprüfung:

```
while:   B    test
loop:    ...
test:    CMP   <condition>
        BEQ   loop
endwhile:
```

Listing F.4: Schleife mit Vorabprüfung, optimiert

F.5. Schleife mit Endprüfung

Im Gegensatz zur Schleife mit Vorabprüfung wird die Schleifenprüfung hier erst nach dem Schleifenkörper durchgeführt. Ist die Bedingung „erfüllt“, wird der Schleifenkörper ein weiteres Mal ausgeführt. Ist die Bedingung „nicht erfüllt“, wird abgebrochen. Der Schleifenkörper wird mindestens einmal ausgeführt.

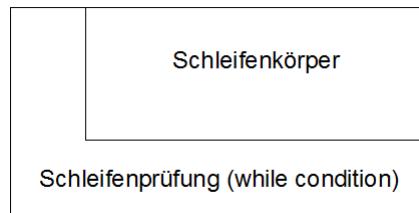


Abbildung F.7.: Schleife mit Endprüfung

In den Programmiersprachen C und Assembler kann die Schleife mit Endprüfung wie folgt programmiert werden:

C	Assembler
do { /* Schleifenkörper */ } while (condition)	do: while: ... CMP <condition> BEQ do enddo: ...

Abbildung F.8.: Schleife mit Endprüfung in C und Assembler

Auch hier werden in der Schleifenprüfung durch eine Instruktion (in diesem Beispiel „CMP“) die Statusbits verändert. Anschliessend erfolgt eine bedingte Verzweigung an den Schleifenanfang, falls die Bedingung weiter „erfüllt“ ist. Die do-while Schleife entspricht somit der zweiten, optimierten while-Schleife aus Kapitel F.4, jedoch ohne die erste Branch-Instruktion, da der Schleifenkörper immer vor der Schleifenprüfung ausgeführt wird.

F.6. Zählschleifen

Die Zählschleife (for-Schleife) ist ein Spezialfall der Schleife mit Vorabprüfung. In C werden in der Zählschleife die kontrollierenden Anweisungen zusammengefasst und zu Beginn der Schleife sichtbar gemacht: Normalerweise wird ein Zähler verwendet, welcher zuerst initialisiert wird (Anfangswert). Als Abbruchbedingung wird der Wert des Zählers mit dem Endwert verglichen. Solange die Bedingung „erfüllt“ ist, wird der Schleifenkörper ausgeführt und der Zähler anschliessend um die Schrittweite verändert. Die Schrittweite kann sowohl positiv als auch negativ sein.



Abbildung F.9.: Zählschleife

In den Programmiersprachen C und Assembler kann die Zählschleife wie folgt programmiert werden:

C	Assembler
<pre>for (r0=Anfangswert; r0 < Endwert; r0+=Schrittweite) { /* Schleifenkörper */ }</pre>	<pre>for: MOV r0, #Anfangswert ... loop: CMP r0, #Endwert BGE endfor ... ADD r0, r0, #Schrittweite B loop endfor:</pre>

Abbildung F.10.: Zählschleife in C und Assembler

Ein Register rX (in diesem Beispiel r0) wird als Zähler verwendet. Zuerst wird der Zähler mit dem Anfangswert initialisiert. Beim Label „loop“ wird mit „CMP“ die Abbruchbedingung geprüft, d.h. es wird verglichen, ob der Zähler den Endwert schon erreicht hat. Ist dies der Fall, so wird mit „BGE“ zum Label „endfor“ verzweigt. Ansonsten wird der Schleifenkörper ausgeführt und anschliessend der Zähler um die Schrittweite verändert.

Je nach Abbruchkriterium kann natürlich auch eine andere Instruktion als „BGE“ verwendet werden. Für negative Schrittweiten kann man anstelle von „ADD“ auch „SUB“ oder eine negative Zahl einsetzen.

F.7. Einführung Datenstrukturen

Datenstrukturen fassen mehrere einfache oder wiederum strukturierte Datentypen in einer Struktur zusammen. Im folgenden Listing ist das Beispiel einer Datenstruktur für eine Adresse gegeben:

```
struct Adresse
{
    char Name[MAX_CHAR];
    char Vorname[MAX_CHAR];
    char Strasse[MAX_CHAR];
    int Nummer;
    int PLZ;
    char Ort[MAX_CHAR];
};
```

Listing F.5: Datenstruktur für eine Adresse in der Programmiersprache C

In Hochsprachen werden Datenstrukturen durch den Sprachumfang unterstützt. Datenstrukturen fassen logisch zusammen gehörende Daten unter einem Namen zusammen. Die zusammen gehörenden Daten können so unter einem gemeinsamen Namen referenziert und über Selektoren einzeln ausgewählt werden. Daraus ergeben sich folgende Vorteile für den Code:

- Besser lesbar und wartbar
- Besser strukturiert

Auch in Assembler kann man sich die Vorteile von Datenstrukturen zu Nutzen machen. Dies ist Thema dieses Kapitels.

Datenstrukturen sind immer problembezogen. Sie werden so definiert, dass sie die Wirklichkeit (beispielsweise eine Adresse) auf einen Datensatz abbilden. Zudem besteht immer eine enge Kopplung zwischen Programmcode und Datenstrukturen. Wird eine Datenstruktur geändert, hat dies auch Änderungen im Programmcode zur Folge.

F.8. Eindimensionale Arrays

In einem Array werden mehrere Elemente des gleichen Datentyps gespeichert. Die einzelnen Elemente werden fortlaufend ohne Lücken im Speicher abgelegt. Mit Hilfe eines Index werden die einzelnen Elemente angesprochen. Das erste Element hat den Index 0 (array[0]).

Beispiel

Anlegen eines Arrays mit vier Elementen vom Typ Word:

```
.data
array: .space 4*4, 0x12          @ array of 4 words, each initialized with 0x12121212
```

Listing F.6: Array mit vier Word-Elementen

Die Assembler-Direktive „.space“ reserviert die gegebene Anzahl Bytes, für 4 Words sind das 4*4 Bytes. Die einzelnen Bytes können initialisiert werden, hier mit 0x12, was schlussendlich jedes Word mit 0x12121212 initialisiert. Falls das Array nicht initialisiert werden soll, kann es auch in der „.bss“ Sektion angelegt werden.

Beispiel

Element Index 2 des Word-Arrays löschen:

```
LDR r0, =array           @ load r0 with base address of array
LDR r1, =2                @ load r1 with index 2
MOV r2, #0                @ value to write is 0
STR r2, [r0, r1, LSL #2]  @ array[2] = 0
B main
```

Listing F.7: Word-Array Index 2 löschen

Zuerst wird die Basisadresse des Arrays nach r0 kopiert. Dadurch zeigt r0 auf das erste Element im Array (Index 0). Anschliessend wird der Index des anzusprechenden Elementes in r1 kopiert. Der Zugriff auf das Array erfolgt mit der Instruktion STR und einer indirekten Adressierung. Die Adresse berechnet sich aus r0 + r1*4. Der Skalierungsfaktor (die Multiplikation mit 4, im Code durch LSL #2) ist wesentlich, da die Adressen beim ARM Cortex-M3, Cortex-M4 und Cortex-M7 Prozessor byte-organisiert sind.

Wichtig: Für die indirekte Adressierung wird ein Skalierungsfaktor verwendet. Dieser hat folgende Werte:

- 1 für Bytes
- 2 für Halfword
- 4 für Word

Beispiel

Anlegen eines Byte-Arrays und löschen von Index 2:

```
.data
array: .space 4, 0x34          @ array of 4 bytes, each initialized with 0x34

.text
main:
    LDR    r0, =array      @ load r0 with base address of array
    LDR    r1, =2           @ load r1 with index 2
    MOV    r2, #0           @ value to write is 0
    STRB   r2, [r0, r1]     @ array[2] = 0 (Byte)
```

Listing F.8: Beispiel mit Byte-Array: Definition und löschen Index 2

F.9. Zweidimensionale Arrays

Betrachten wir ein Array mit 3 Zeilen und 4 Spalten vom Typ Word (array[3][4]):

```
array: .space 3*4*4          @ reserve 3 rows * 4 columns * 4 Bytes
```

Listing F.9: Zweidimensionales Array initialisieren

Das Array kann wie folgt dargestellt werden:

[0,0]	[0,1]	[0,2]	[0,3]
[1,0]	[1,1]	[1,2]	[1,3]
[2,0]	[2,1]	[2,2]	[2,3]

Abbildung F.11.: Zweidimensionale Array

Dieses Array kann nun entweder zeilenorientiert oder spaltenorientiert im Speicher abgelegt werden. Ob Daten zeilenorientiert oder spaltenorientiert abgelegt werden ergibt sich in der Regel aus der Programmiersprache. Bei C/C++ oder Python werden die Daten zeilenorientiert (row major), bei FORTRAN oder MATLAB hingegen spaltenorientiert (columns major) gespeichert. Nachfolgend wird das zeilenorientierte Vorgehen besprochen. Beim zeilenorientierten Vorgehen wird das Array mit 3 Zeilen und 4 Spalten wie folgt im Speicher abgelegt:

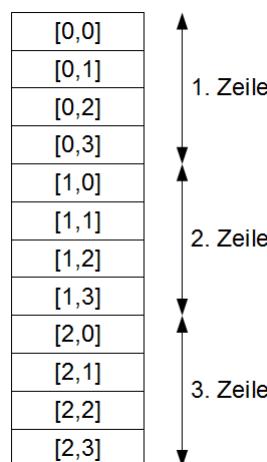


Abbildung F.12.: Zeilenorientiertes Speichern des zweidimensionalen Arrays

Zum Ansprechen des Elementes mit Zeile „i“ und Spalte „j“ können wir für obiges Beispiel folgende, vereinfachte Formel verwenden:

$$\text{Adresse} = 4 * i + j$$

Diese Formel ist nur gültig für ein 4-spaltiges Array mit Byte-Elementen und der Basisadresse 0. Eine allgemeinere Formel, welche auch die Basisadresse des Arrays sowie die Elementgrösse miteinbezieht, lautet wie folgt:

$$\text{Elementadresse} = \text{Basisadresse} + S * (\text{SZ} * i + j)$$

dabei bedeuten:

Basisadresse: Startadresse des Arrays

S: Skalierungsfaktor, Byte = 1, Halfword = 2, Word = 4

SZ: Spaltenzahl

i: Zeilenindex

j: Spaltenindex

Die allgemeine Umsetzung dieser Formel ist aufwändig. Insbesondere die Multiplikation SZ*i erfordert je nach Mikroprozessor viel Rechenzeit. Die Multiplikation mit dem Skalierungsfaktor S kann hingegen durch eine Schieboperation implementiert werden (Faktor 2 oder 4).

F.10. Strings

Ein String ist eine Folge von ASCII-Zeichen, welche mit dem ASCII-Zeichen NUL ('\\0' oder 0x00) terminiert wird. Ein String kann wie folgt angelegt werden:

```
text: .asciz "abcABC0123"      @ null-terminated string
```

Listing F.10: String

Beachten Sie, dass das Terminierungszeichen NUL ('\\0') und die Ziffer '0' (0x30) unterschiedliche ASCII-Werte haben!

F.11. Strukturen

Mit Hilfe von Strukturen können unterschiedliche Datentypen zusammengefasst werden. Als Beispiel soll eine Struktur zum Speichern eines Datums betrachtet werden:

```
date: .byte 14      @ day as byte (1-31)
      .byte 03      @ month as byte (1-12)
      .hword 1879    @ year as halfword (e.g. 0000 - 3000)
```

Listing F.11: Datenstruktur date

Die Struktur „date“ besitzt zwei Byte-Elemente sowie ein Halfword-Element. Um auf die einzelnen Strukturelemente zuzugreifen können Offsets definiert werden, welche die Position der einzelnen Elemente angeben. Für die Struktur „date“ ergibt dies:

```
.set    OFFSET_MONTH, 1
.set    OFFSET_YEAR, 2
```

Listing F.12: Definition der Offsets

Für das Element „Tag“ wird kein Offset benötigt, da seine Adresse identisch ist mit der Basisadresse von „date“.

Soll beispielsweise das Jahr ins Register r0 kopiert werden, so kann dies wie folgt programmiert werden:

```
LDR    r1, =date          @ load r1 with base address
LDRH   r0, [r1, #OFFSET_YEAR]  @ read year (halfword)
```

Listing F.13: Zugriff auf das Jahr der Datenstruktur „date“

F.12. Stack

Eine Beschreibung des Aufbaus und möglicher Operationen eines Stacks finden Sie im Kapitel 9.1 „Subroutinen“. Der dort beschriebene Stack wird von der CPU für die Parameterübergabe, das Retten von Registern und für lokale Variablen verwendet. Der Anwender kann zusätzliche Stacks zum Speichern anwendungsspezifischer, beliebiger Daten definieren und anlegen.

Annahmen für die Implementation eines anwendungsspezifischen Stacks:

- Die Werte, welche auf den Stack geschrieben (push) oder vom Stack gelesen (pop) werden, stehen im Register r0. Dadurch können die Stack-Operationen auch von C-Funktionen aufgerufen werden, da nach AAPCS die Parameter in r0 übergeben oder zurückgegeben werden.
- Der Beispielstack ist für Word-Variablen ausgelegt.
- Der Stack wächst in negativer Richtung (abnehmende Adressen).
- Der Stack-Pointer zeigt auf das zuletzt geschriebene Element (push: decrement before, pop: increment after)
- Als Stack-Pointer wird eine globale Variable verwendet.

Beispiel

Anlegen von Stack und Stack-Pointer:

```
.set    STACK_SIZE, 1000      @ define stack size
.stack
stack: .space  STACK_SIZE, 0      @ reserve address space for stack
stack_pointer:.word stack + STACK_SIZE @ global variable, holds actual stack pointer
```

Listing F.14: Initialisierung des Stacks

Die Variable stack_pointer muss auf die Startadresse des Stacks zeigen. Da der Stack in negative Richtung wächst, ist dies die höchste Adresse des Stacks. Somit muss stack_pointer auf stack + STACK_SIZE initialisiert werden.

Damit ergibt sich nach der Initialisierung folgender Zustand des Stacks und der Pointer:

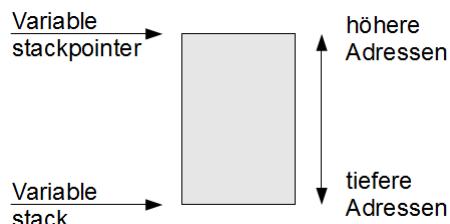


Abbildung F.13.: Stack nach der Initialisierung

Die push-Funktion kann wie folgt implementiert werden:

```
@ parameter r0 holds value to write to stack
@ r1 holds stack pointer address
@ r2 holds stack pointer value
@ r3 holds stack address
@ stack full error is not implemented
```

```
push:
    PUSH  {lr}          @ save return address on system-stack
    LDR   r1, =stack_pointer @ load r1 with stack pointer address
    LDR   r2, [r1]         @ load r2 with stack pointer value
    LDR   r3, =stack       @ load r3 with stack address
    CMP   r2, r3         @ check if stack is already full
    BLE  stack_full
```

```

        SUB    r2, r2, #4          @ decrement stack pointer (decrement before)
        STR    r2, [r1]            @ save new stack pointer value
        STR    r0, [r2]            @ push value to stack
        B      endpush             @ -> end
stack_full:   ...           @ error code
endpush:      POP   {pc}       @ return

```

Listing F.15: push

Die pop-Funktion kann wie folgt implementiert werden:

- @ parameter r0 is return value, element read from stack
- @ r1 holds stack pointer address
- @ r2 holds stack pointer value
- @ r3 holds stack address
- @ stack empty error is not implemented

```

pop:
        PUSH   {lr}                @ save return address on system-stack
        LDR    r1, =stack_pointer  @ load r1 with stack pointer address
        LDR    r2, [r1]              @ load r2 with stack pointer value
        LDR    r3, =stack            @ load r3 with stack address
        ADD    r3, #STACK_SIZE     @ r3 holds stack top address
        CMP    r2, r3                @ check if stack is empty
        BGE   stack_empty           @ -> end
        LDR    r0, [r2]              @ pop value from stack, r0 is return value
        ADD    r2, r2, #4            @ increment stack pointer (increment after)
        STR    r2, [r1]              @ save new stack pointer value
        B      endpop               @ -> end
stack_empty: ...
endpop:      POP   {pc}       @ return

```

Listing F.16: pop

Im Gegensatz zum Stack aus Kapitel 9 „Subroutinen“, der keine Prüfung auf Überlauf oder Unterlauf hat, kann bei einem „handgestrickten“ Stack eine Prüfung eingebaut und entsprechend durch eine Fehlerbehandlung reagiert werden.

F.13. Queue / Ringbuffer

Eine Queue kann im Gegensatz zu einem Stack an beiden Enden bearbeitet werden. Es gilt „First In, First Out“. Eine Queue benötigt zwei Pointer, welche auf die Positionen zeigen, wo aktuell gelesen und geschrieben wird:

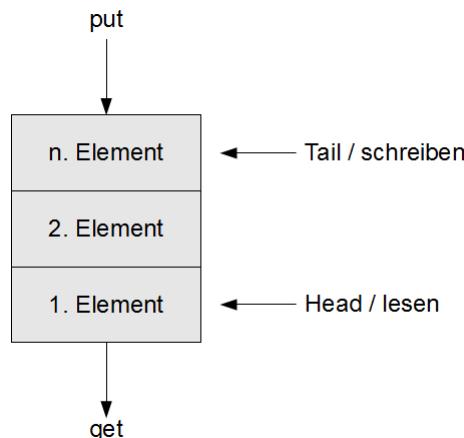


Abbildung F.14.: Queue

Bei der Implementierung einer Queue sind folgende Punkte zu beachten:

- Head und Tail dürfen sich nicht kreuzen
- Eine Queue kann sich wie ein Wurm durch den ganzen Speicher fressen

In der Praxis werden anstelle von Queues oft Ringbuffer eingesetzt. Dadurch entfällt die Wurmproblematik. Für die Implementation von Ringbuffern gibt es zwei Varianten:

1. Head und Tail dürfen auf das gleiche Speicherelement zeigen. Es ist eine zusätzliche Variable „size“ zu definieren, welche die Anzahl gespeicherter Werte angibt. Size = 0 entspricht einem leeren Ringbuffer.
2. Head und Tail dürfen nie auf dasselbe Speicherelement zeigen. Dadurch wird ein Speicherelement geopfert, es braucht jedoch keine size-Variable ($\text{size} = \text{Tail} - \text{Head} - 1$). Initialisierung z.B. mit Head = 1, Tail = 2, gelesen wird bei Position Head + 1.

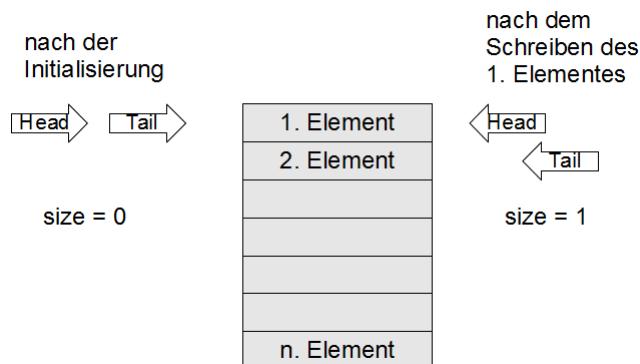


Abbildung F.15.: Head und Tail im Ringbuffer, gemäss Variante 1

In beiden Fällen ist im Code der Überlauf der beiden Pointer Head und Tail zu prüfen (im nachfolgenden Beispiel der Überlauf von 8 nach 1).

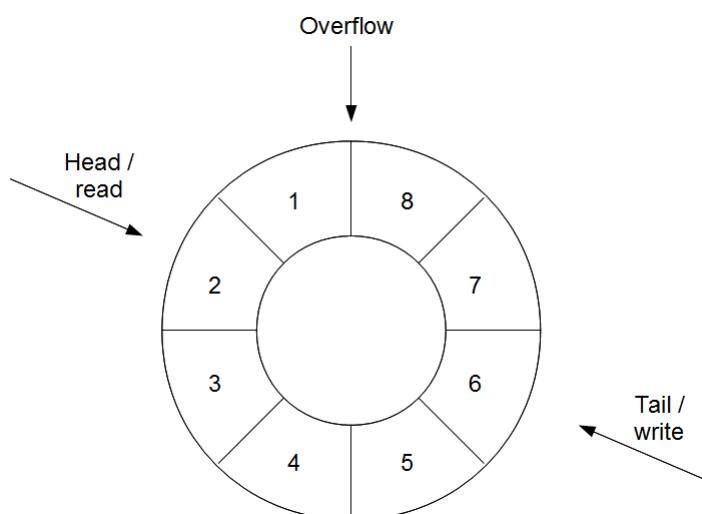


Abbildung F.16.: Beispiel Ringbuffer mit 8 Elementen

F.14. Kommentare

F.14.1. Übersicht

Gute Kommentare sind für das Verständnis von Assembler-Code besonders wichtig. Ein paar Tipps:

- Geben Sie in den Subroutinen-Beschreibungen nicht nur an, was die Subroutine macht, sondern auch, welche Register für das Speichern lokaler Variablen verwendet werden.
- Wenn eine Subroutine einen etwas komplexeren Algorithmus implementiert, so geben Sie diesen ebenfalls in der Subroutinen-Beschreibung an. Geben Sie hier für die Beschreibung bereits die Register an, welche anschliessend im Code verwendet werden. Verwenden Sie anschliessend die einzelnen Zeilen der Algorithmus-Beschreibung als zusätzlichen Kommentar im Assembler-Code.
- Setzen Sie für jede Assembler-Zeile einen sinnvollen Kommentar, mit Bezug auf den Algorithmus.

F.14.2. Beispiel Bubble-Sort

Als Beispiel, wie Assembler-Code kommentiert werden kann, finden Sie nachfolgenden Code. Dabei geht es darum, ein Array mit Hilfe des Bubble-Sort Algorithmus zu sortieren. Die Subroutine sort_array kann wie folgt implementiert werden:

```
*****  
* Array of numbers  
*****  
.data  
.align  
.set size, 4           @ number of array elements  
array:                 @ array to sort  
.word 0x37, 0x7382, 0x12, 0xA457  
  
*****  
* Code  
*****  
.text                  @ section text (executable code)  
  
/**  
 * \brief Sorts an array of unsigned numbers, using bubble sort  
 *        there are no optimizations!  
 * \param r0: array base address  
 * \param r1: array size  
 * \return none  
 * \remark The subroutine is using the following algorithm:  
 *          for (r2 = (array_size - 1); r2 >= 0; r2--)  
 *          {  
 *              for (r3 = 1; r3 <= r2; r3++)  
 *              {  
 *                  if (numbers[r3-1] > numbers[r3])  
 *                  {  
 *                      temp = numbers[r3-1];  
 *                      numbers[r3-1] = numbers[r3];  
 *                      numbers[r3] = temp;  
 *                  }  
 *              }  
 *          }  
 * \remark registers r4 to r6 are used to swap numbers:  
 *         r4: numbers[r3]  
 *         r5: numbers[r3-1]  
 *         r6: address offset in array  
 */  
sort_array:  
    PUSH   {r4-r6,lr}  
    # loop 1: for (r2 = (array_size - 1); r2 >= 0; r2--)  
    MOV    r2, r1           @ r2 = array_size - 1  
    SUB    r2, r2, #1  
sort_loop1:
```

```

    CMP    r2, #0          @ check loop 1, r2 >= 0 ?
    BLT    sort_end        @ no, loop 1 end --> branch sort_end

    # loop 2: for (r3 = 1; r3 <= r2; r3++)
    MOV    r3, #1          @ r3 = 1
sort_loop2:
    CMP    r3, r2          @ r3 <= r2?
    BGT    sort_loop1_end  @ no, loop2 end --> branch sort_loop1_end
swap:
    MOV    r6, r3, LSL #2  @ address offset for index r3: r6 = r3*4
    LDR    r4, [r0,r6]      @ r4 = numbers[r3]
    SUB   r6, r6, #4       @ address offset for index (r3 - 1)
    LDR    r5, [r0,r6]      @ r5 = numbers[r3-1]
    CMP    r5, r4          @ if (numbers[r3-1] > numbers[r3])
    BLS    sort_loop2_end  @ no --> do not swap
    STR    r4, [r0,r6]      @ numbers[r3-1] = lower value
    ADD    r6, r6,#4       @ prepare next address offset
    STR    r5, [r0,r6]      @ numbers[r3] = higher value

sort_loop2_end:
    ADD    r3, r3, #1       @ loop 2 end, prepare next loop, r3++
    B     sort_loop2        @ next loop 2

sort_loop1_end:
    SUB    r2, r2, #1       @ loop 1 end, prepare next loop, r2--
    B     sort_loop1        @ next loop 1
sort_end:
    POP   {r4-r6,pc}        @ return

```

Listing F.17: Subroutine mit Bubble-Sort Algorithmus

Die Subroutine `sort_array` kann aus dem Hauptprogramm wie folgt aufgerufen werden (es ist die AAPCS zu beachten):

```

/***
 * \brief main program
 *        calls subroutine sort_array, passing array and size of array
 * \param none
 */
main:
    LDR    r0, =array        @ r0 points to array base address
    MOV    r1, #size          @ r1 holds number of values in array
    BL    sort_array
    B     main

```

Listing F.18: Aufruf der Subroutine `sort_array` aus dem Hauptprogramm

Glossar

AAPCS ARM Architecture Procedure Call Standard.

ADC Analog-to-Digital Converter.

ALU Arithmetic Logical Unit.

BSP Board Support Package.

CAS Column Address Strobe.

CCITT Comité Consultatif International Téléphonique et Télégraphique.

CISC Complex Instruction Set Computer.

CMSIS Cortex Microcontroller Software Interface Standard.

CPU Central Processing Unit.

CS Chip Select.

CTS Clear to Send.

DAC Digital-to-Analog Converter.

DMA Direct Memory Access.

DRAM Dynamic Random Access Memory.

DSP Digital Signal Processing.

EEPROM Electrically Erasable Programmable ROM.

EIA Electronic Industries Alliance.

ELF Executable and Linking Format.

EPROM Erasable Programmable ROM.

EXTI External Interrupt/Event Controller.

FPU Floating Point Unit.

FRAM Ferroelectric Random Access Memory.

GPIO General-Purpose Input Output.

I2C Inter-Integrated-Circuit.

IDE Integrated Development Environment.

ISR Interrupt Service Routine.

LIFO Last In First Out.

LSB Least Significant Bit.

MAC Multiply ACCumulate.

MIMD Multiple Instruction Multiple Data.

MISD Multiple Instruction Single Data.

MISO Master In Slave Out.

MMU Memory Management Unit.

MOSI Master Out Slave In.

MPU Memory Protection Unit.

MSB Most Significant Bit.

NaN Not a Number.

NMI Non Maskable Interrupt.

NVIC Nested Vectored Interrupt Controller.

OTP One Time Programmable ROM.

PSR Program Status Register.

PWM Pulse Width Modulation.

RAM Random Access Memory.

RAS Row Address Strobe.

RISC Reduced Instruction Set Computer.

ROM Read Only Memory.

RS232 Recommended Standard 232.

RTS Request to Send.

SCK Signal Clock.

SCL Serial CLock.

SDA Serial DAta.

SDRAM Synchronous Dynamic Random Access Memory.

SIMD Single Instruction Multiple Data.

SISD Single Instruction Single Data.

SPI Serial Peripheral Interface.

SRAM Static Random Access Memory.

SS Slave Select.

TCM Tightly Coupled Memory.

TLB Translation Lookaside Buffer.

UAL Unified Assembler Language.

UART Universal Asynchronous Receiver Transmitter.

VTOR Vector Table Offset Register.

Literaturverzeichnis

- [1] C. W. A. Sloss, D. Symes, *ARM System Developer's Guide*. Morgan Kaufman, 2004, ISBN: 978-1-55860-874-0.
- [2] ARM, *ARM Cortex-M7 Processor Technical Reference Manual*, 2014. [Online]. Available: <https://developer.arm.com/documentation/ddi0489/b/>
- [3] ——, *ARM Cortex-M7 Device Generic User Guide*, 2018. [Online]. Available: <https://developer.arm.com/documentation/dui0646/c>
- [4] ——, *Procedure Call Standard for ARM Architecture*, Jun 2020. [Online]. Available: <https://developer.arm.com/documentation/ihi0042/latest>
- [5] ——, *ARMv7-M Architecture Reference Manual*, 2021. [Online]. Available: <https://developer.arm.com/documentation/ddi0403/latest>
- [6] ARM, *CMSIS - Cortex Microcontroller Software Interface Standard*, <http://www.arm.com/products/processors/cortex-m/cortex-microcontroller-software-interface-standard.php>, 2021, [Verschiedene Informationen zu CMSIS].
- [7] ChaN, "Fatfs module application note," 2000. [Online]. Available: <http://elm-chan.org/fsw/ff/doc/appnote.html>
- [8] Philips, "The i2c-bus specification, version 2.1," 2000. [Online]. Available: <http://www.nxp.com/documents/other/39340011.pdf>
- [9] STMicroelectronics, *STM32F7 Series and STM32H7 Series Cortex-M7 processor programming manual*, June 2019. [Online]. Available: https://www.st.com/resource/en/programming_manual/dm00237416-stm32f7-series-and-stm32h7-series-cortexm7-processor-programming-manual-stmicroelectronics.pdf
- [10] ——, *Description of STM32H7 HAL and LL drivers*, July 2020. [Online]. Available: https://www.st.com/resource/en/user_manual/dm00392525-description-of-stm32h7-hal-and-lowlayer-drivers-stmicroelectronics.pdf
- [11] ——, *STM32H7xx Reference Manual*, Feb 2020. [Online]. Available: <https://www.st.com/en/microcontrollers-microprocessors/stm32h743-753.html#documentation>
- [12] ——, *STM32H743 Datasheet*, April 2021. [Online]. Available: <https://www.st.com/resource/en/datasheet/stm32h743bi.pdf>
- [13] J. Yiu, *The definitive guide to ARM Cortex-M3 and Cortex-M4 processors*. Newnes, 2014, ISBN-13: 978-0-12-408082-9.

Stichwortverzeichnis

- .2byte, 102
- .4byte, 102
- .align, 100
- .arm, 98
- .ascii, 101
- .asciz, 101
- .asm, 71
- .balign, 101
- .bss, 105
- .byte, 101
- .data, 104
- .elf, 72
- .end, 107
- .endm, 110
- .equ, 104
- .exitm, 110
- .extern, 99
- .global, 99
- .hword, 102
- .if, 108
- .ifdef, 109
- .ifndef, 109
- .include, 103
- .irp, 112
- .lis, 71
- .macro, 110
- .o, 71
- .org, 104
- .p2align, 101
- .rept, 111
- .s, 71
- .section, 105
- .set, 104
- .space, 103
- .text, 104
- .thumb, 98
- .word, 102
- =, 104

- 8051, 16

- A/D-Wandler, 207
- AAPCS, 223
- ADC, 207
- Adressbus, 19
- Adressdecoder, 21
- ALU, 9
- Applikation, 133

- Arbeitsspeicher, 27
- ARM Cortex-Ax, 16
- ARM Cortex-Mx, 16
- ascending, 118
- Assembler, 71
- Assembler-Datei, 64
- Assembler-Direktiven, 97
- Ausgangstreiber, 158

- BASEPRI, 163
- Betriebssystem, 135
- Bit, 217
- Board Support Package, 152
- Bootloader, 130
- BSP, 152
- Bus-Grant, 41
- Bus-Request, 41
- Businterface, 9
- Byte, 217

- Cache, 29
- Cache Controller, 30, 31
- Cache Data Streaming, 31
- Cache Eviction, 32
- Cache Hit, 31
- Cache Line, 30
- Cache Line Fill, 31
- Cache Memory, 30, 31
- Cache Miss, 30, 31
- Cache Policies, 34
- Cache Set Associative, 33
- Cache Tags, 31
- Cache Way, 32
- Caching, 30
- CAS, 25
- Chip Select, 21
- Circular Addressing, 12
- CISC, 15
- clobber list, 140
- CMSIS, 148
- Co-Prozessoren, 13
- Coherent Cache, 34
- Coldfire, 16
- column-address, 25
- Computer-Kategorien, 3
- Conditional Execution, 236
- Controlbus, 19
- Counter, 199

- CPU, 8
Cross-Assembler, 71
CTS, 179
- D/A-Wandler, 211
DAC, 211
Datenbus, 7, 19
Datenspeicher, 7
decode, 9
descending, 118
deterministisch, 162
Direct Mapped Cache, 32
Directory Store, 31
Dirty Bit, 31
Distributed Memory, 6
DMA, 40
do-while, 239
Double Word, 218
DRAM, 23
DSP, 11
- Editor, 71
EEPROM, 23
Embedded System, 162
Entscheidungen, 235
EPROM, 22
even-Parity, 181
EXC_RETURN, 172
Exception, 161
Exception Handler, 162
Exception Latency, 174
Exception Number, 52
execute, 9
EXTI, 169
- FatFS, 152
fetch, 8
flüchtige Speicher, 23
Flash, 23
Floating Point Double Precision, 10
Floating Point Single Precision, 10
FLOPS, 16
Flynn, 3
for-Schleife, 240
FPU, 9
FRAM, 23
Frame-Pointer, 126
Frames, 38
full-descending, 225
- GE, 52
GPIO, 155
- Halfword, 217
Handshake-Signale, 181
Hardware-Ressourcen, 135
Harvard Architektur, 7
Header, 65
- Heap, 127
I2C, 187
I2C-Acknowledge, 189
ICI/IT, 52
IDE, 70, 73, 150
Idiome, 13
IEEE-754, 10
if Schleife, 235
Inline-Assembler, 139
input operand list, 140
Instruction Unit, 8
Instruktionsbus, 7
Interrupt, 161
Interrupt Latency, 174
Intrinsics, 13
ISR, 134, 162
IT-Instruktion, 236
- Kommentar, 66
Kontrollstrukturen, 235
- Labelfeld, 65
LIFO, 117
Link-Register, 116, 125, 224
Linker, 72
List-File, 71
Locator, 72
logische Adressen, 37
lokale Variablen, 125
LSB, 218
- MAC, 11
Main Memory, 27
Map-File, 72
Maschinencode, 72
MAX3245, 179
Memory Access Attributes, 35
Memory Region, 35
Memory-Map, 21
Memory-Matrix, 24
Microcontroller, 14
Mikroprozessor, 14
MIMD, 5
MIPS, 16
MISD, 4
MISO, 194
MMU, 34, 36
MOSI, 194
MSB, 218
MSP430, 16
- NaN, 220
Nibble, 217
nichtflüchtige Speicher, 22
NMI, 161
NVIC, 166
- Object-File, 71

odd-Parity, 181
 Open Collector, 159
 Open Drain, 159
 Operanden, 66
 Operatoren, 68
 OTP, 22
 output operand list, 140

 Page Table, 38
 Pages, 38
 Parity-Bit, 180
 Peripherie, 145
 physikalische Adressen, 37
 PIC, 16
 Pinbelegung, 26
 Pop, 117
 PRIMASK, 163
 Program-Counter, 8, 125, 224
 Programmspeicher, 7
 Prozessor, 14
 PSR, 51
 Push, 117
 Push-Pull, 158
 PWM, 200

 Rücksprungadresse, 116
 RAM, 23
 RAS, 25
 Rechenwerk, 9
 Refresh-Zyklus, 203
 Register, 9, 27
 relocation, 37
 Ringbuffer, 12
 RISC, 15
 ROM, 22
 row-address, 25
 RS232, 177
 RTS, 179

 Schaltwerk, 8
 Schleifen, 235
 SCK, 194
 SCL, 187
 Scratch-Register, 125, 224
 SDA, 187
 SDI, 194
 SDO, 194
 SDRAM, 23
 Sequenzen, 235
 Set Index Field, 31
 Shared Memory, 5
 SIMD, 4, 11
 SISD, 3
 Speichergrösse, 26
 Speicherorganisation, 27
 SPI, 193
 Split Cache, 30
 SRAM, 23

 SS, 194
 Stack, 117
 Stack-Pointer, 117, 125, 224
 stacking, 171
 Start-Condition, 188
 Startup-Code, 129
 Starutp-Code, 130
 Steuercodes, 181
 Steuerwerk, 8
 Stop-Condition, 188
 swapping, 36
 switch, 237
 Symbole, 67

 TCM, 27
 Timer, 199
 TLB, 38
 TLB hit, 39
 TLB miss, 39
 Trashing, 32
 Tristate, 159

 UAL, 75, 99
 UART, 177
 Unified Cache, 30
 unstacking, 172

 Valid Bit, 31
 virtuelle Adressen, 37
 von-Neumann Architektur, 7
 VTOR, 168

 Wahrheitstabellen, 27
 wait cycles, 29, 34
 Watchdog, 202
 while, 238
 Word, 217
 Write Buffer, 34
 Writeback Policy, 34
 Writethrough Policy, 34

 Xon/Xoff, 181

 Zweierkomplement, 219