Zusammenfassung Microcontroller Fundamentals

Joel von Rotz

2022-01-07

Inhaltsverzeichnis

Wichtige Seiten

Titel	Kapitel	Register
Port Control and Interrupts (`PORT`)	239	248
System Integration Module (`SIM`)	259	261 (`SCGCx`: 277)
Analog-to-Digital Converter (`ADC`)	761	768
FlexTimer Module (`FTM`)	891	903
Serial Peripherial Interface (`SPI`)	1123	1132
Inter-Integrated Circuit (`I2C`)	1181	1185
→ *I2C divider and hold values*	1200	-
Universal Asynchronous Receiver/Transmitter (`UART`)	1215	1224
General-Purpose Input/Output (`GPIO`)	1375	1378
Nested Vectored Interrupt Controller (`NVIC`)	60	-

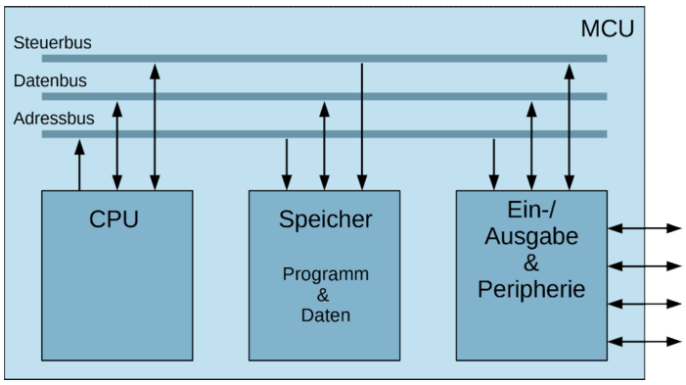
Beschreiben von Registern

```
#define PIN_MASK (0x1D)

uint32_t key = GPIO->PDIR;
keys = keys << 1;
GPIOC->PDDR = (GPIOC->PDDR & ~PIN_MASK) | keys;
```

1. Register auslesen
2. Mit `&` alle nicht zu verändernde Bits maskieren
3. Neuer Wert auf Register mit `|` (ODER) Verknüpfung beschreiben

> **Hinweis**
>
> Damit wird nur der zu veränderte Wert verändert und die anderen Werte bleiben unverändert.

Mikrocontroller



- ARM → *Advanced RISC Machines*
- CISC → *Complex Instruction Set Computer*
- RISC → *Reduced Instruction Set Computer*
- Memory mapped → Peripherien (Inputs & Outputs) geben sich als Memory-Stellen aus
- Mikrocontroller-Units (MCU, MC) sind Single-Chip Computer.

- Central Processing Unit (CPU) (Zentrale Verarbeitungseinheit)
- Speicher (RAM, ROM): für Programm & Daten
- Ein-/Ausgabe-Einheiten: für die Kommunikation mit der Umgebung
- Bus-System: Verbindet die Systemteile

Name	Bedeutung	Verbindungsart
MCU	Microcontroller Unit, besteht aus: Steuerbus, Adressbus, Datenbus	
CPU	Central Processing Unit	Unidirektional
RAM	Random Access Memory	Bidirektion
ROM	Read-Only Memory	Unidirektional
Flash-Speicher	Speicherbausstein	Bidirektional
Speicher	Für Programm und Daten	Bidirektional
Ein / Ausgabe (I/O)	Für Kommunikation mit Umgebung	Bidirektional
Adressbus	Ansprechen von Speicher und (I/O)	Unidirektional
Datenbus	Austausch von Daten	Bidirektional
Steuerbus	Steuerung des Bussystems, wie z.B Interrups, Read/Write, CLW	Bidirektional

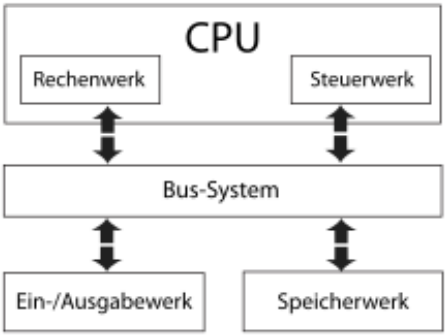
Vergleich MPU

MCU haben Speicher & Peripherien (A/D-Wandler, Schnittstellen, etc.) integriert. MPU besitzen nur den CPU und die restlichen Elemente müssen nachträglich verbunden werden.

MCU-Architektur

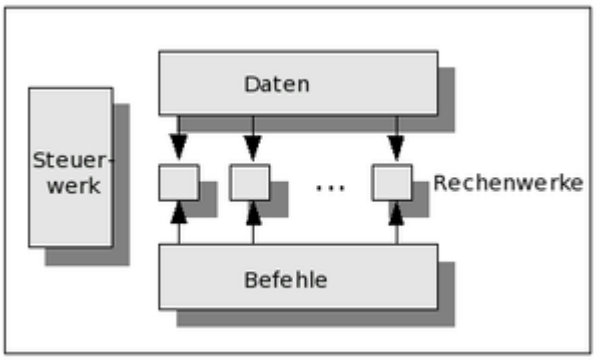
Von Neumann Architektur: hat nur ein gemeinsamer Bus für Programm und Daten. Programm und Daten im gleichen Memory, oft in low-cost MCUs.

→ Simpler



Harvard Architektur: hat zwei getrennte Bussysteme für Programm und Daten (⇒ hoher Durchsatz), oft in high-performance MCUs.

→ Schneller



Vereinfachte System Memory Map

0x0000_0000 – 0x0000_03FF	Exception Vectors	1kB Interrupt Vector Table
0x0000_0400 – 0x0007_FFFF	Read only region (ROM)	511KB Flash
0x0008_0000 – 0x1FFE_FFFF	Reserved	
0x1FFF_0000 – 0x2000_FFFF	SRAM region (RAM)	128KB RAM: 0x1FFF_0000 – 0x2000_FFFF
0x2001_0000 – 0x3FFF_FFFF	Reserved	
0x4000_0000 – 0x4007_FFFF	Peripherie	512KB peripheral bus Bridge (AIPS0)
0x4008_0000 – 0x400F_EFFF	Reserved	
0x400F_0000 – 0x400F_FFFF	GPIO	4KB GPIO region (davon ~280 Bytes benutzt)
0x4010_0000 – 0xFFFF_FFFF	Reserved	

Heap: Globale Variablen ; Statische Variablen

Stack: Rücksprungadressen ; CPU-Status Backup ; Lokale Variablen

Bitgruppen & Speicherorganisation

$$64\text{K} = 2^{16} = 65536$$

$$1\text{K} = 2^{10} = 1024$$

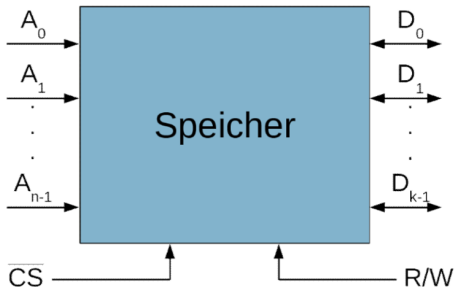
Nibble/Tetrade: 4 Bits / halbes Byte

Byte: 8 Bits / 1 Byte

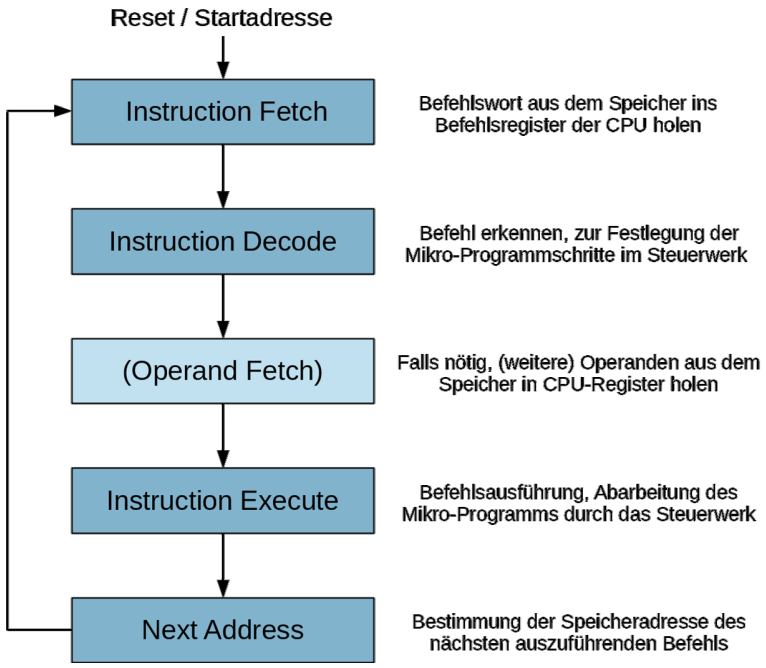
Word: *MCU Spezifisch* → 16 Bits / 2 Bytes

Speicher Berechnen:

- Anzahl Adressleitungen = n
- Anzahl Speicherstellen = 2^n
- Anzahl Bitspeicherplätze = $2^n \cdot k$

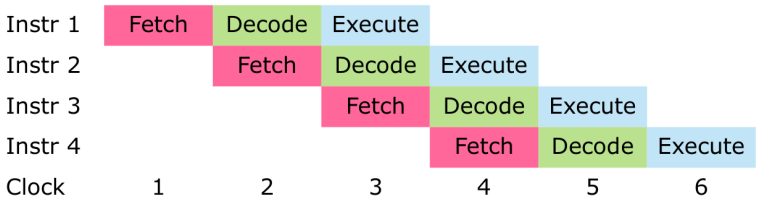


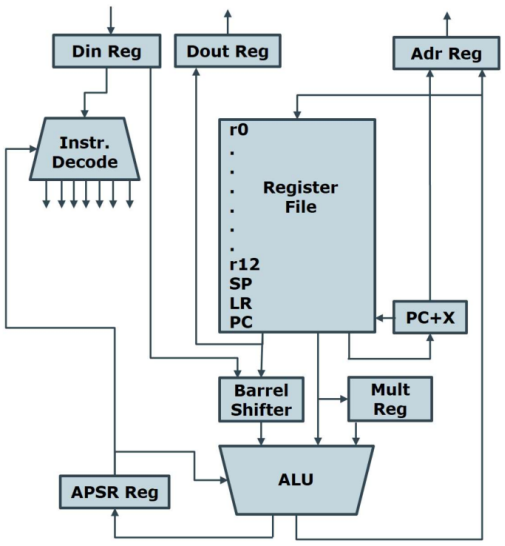
Befehlszyklus einer CPU



Pipelining

Die ARM Cortex M CPU haben eine 3 stufige Pipeline: In jedem Taktzyklus wird gleichzeitig ein Befehl geladen, der vorherige Befehl dekodiert und der vor-vorherige Befehl ausgeführt.





32 Bit, nach aussen von Neumann

Reg Register = Speicher (FF, RAM)
PC Programm Counter
ALU Arithmetic Logic Unit
APSR Condition Code Register
SP Stack Pointer:
Stapelspeicher für Kontext und Parameter
LR Link Register

Wichtige CPU Register

r0	General Purpose (Low)
r1	"
r2	"
r3	"
r4	"
r5	"
r6	"
r7	"
r8	General Purpose (High)
r9	"
r10	"
r11	"
r12	"
r13	Main Stack Pointer (MSP)
r14	Link Register (LR)
r15	Program Counter (PC)

APSR	Application Program Status Register NZCVQ+SIMD Flags

Alle Register sind 32 Bit

APSR:
N bit[31] Negative Flag
Z bit[30] Zero Flag
C bit[29] Carry Flag
V bit[28] Overflow Flag
Q bit[27] Saturation Flag

GE bit[19-16] Single Instruction
Multiple Data (SIMD) Flags

General Purpose Low:: gehen nur bis 7, da die meisten Befehle nur Register besprechen können, welche mit 3 Bits erreichbar sind Flags im APSR

General Purpose High:: Restliche Register für 32 Bit Befehle

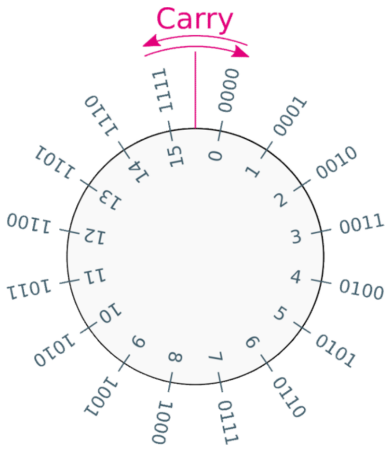
StackPointer (SP) → R13: Zeigt auf den aktuellen Speicherplatz des Stacks

Link Register (LR) → R14: Abspeicherung für Programmsprünge

ProgrammCounter(PC) → R15: Zeigt auf den aktuellen oder den nächsten Speicher für den nächsten Befehl

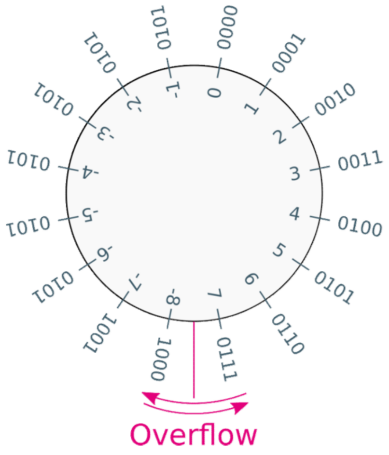
Carry

Vorzeichenlose Zahlen (**unsigned**)



Overflow

Zahlen mit Vorzeichen (**signed**)

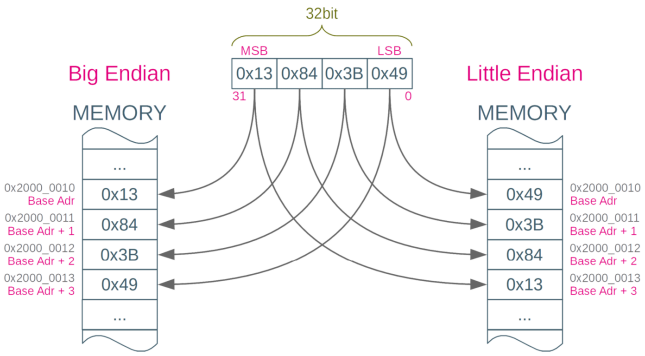


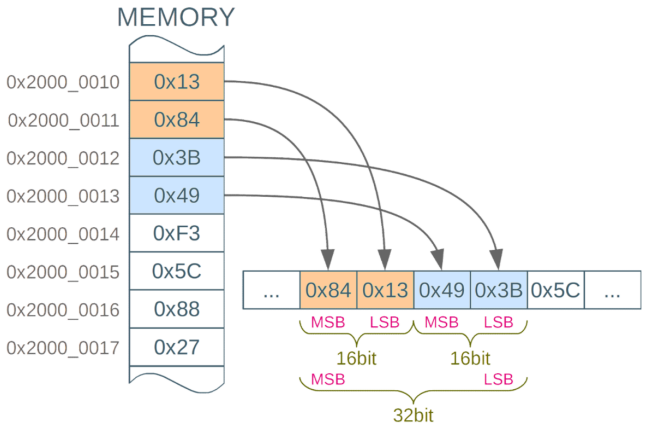
Zweier Komplement

	-27	0	0	0	1	1	0	1	1
①	27	0	0	0	1	1	0	1	1
②	NOT	1	1	1	0	0	1	0	0
③	+1	1	1	1	0	0	1	0	1

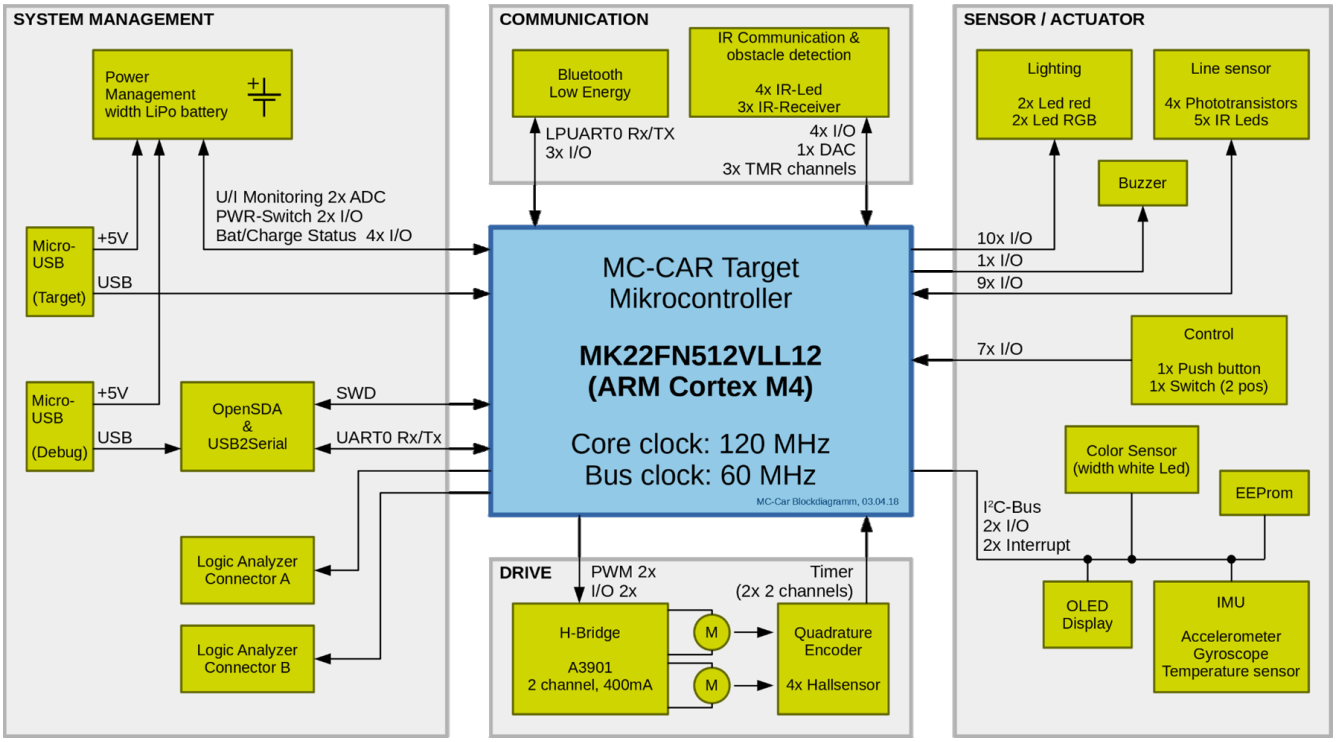
Endianess

Es gibt drei Arten. Im ARM werden die Befehle mit **Halfword Big Endian** abgespeichert!

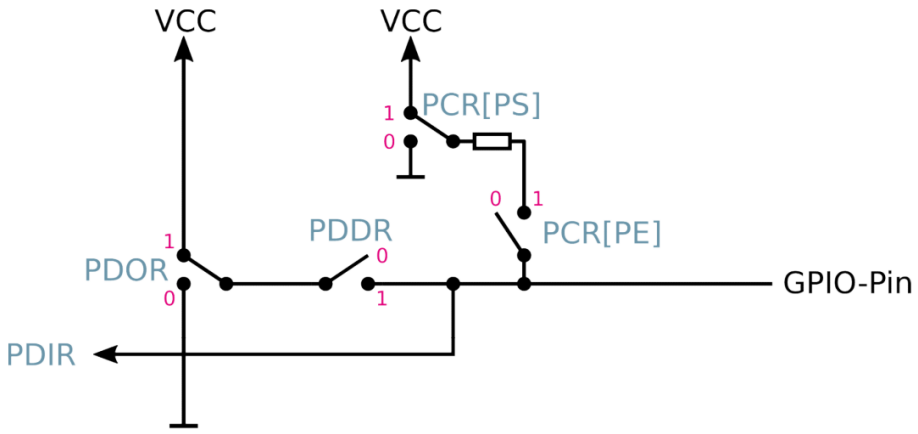




MC-CAR



GPIO



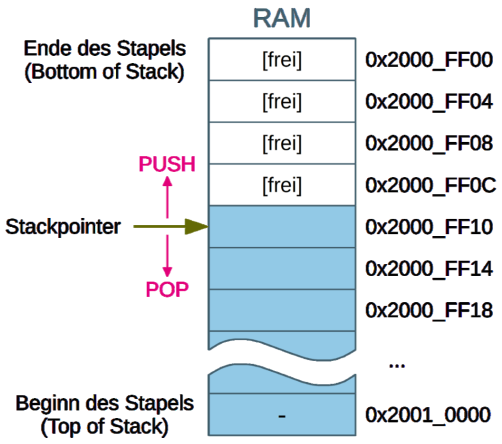
! Port Konfiguration

Pins müssen für die einzelnen Module via `PORTx` konfiguriert werden (insbesondere `PORTx->PCR->MUX` und die nötigen Einstellungen wie Pull-Up oder Open Drain).

Stack

Stack-Pointer Bewegungen

- Der Stackpointer zeigt stets auf die Adresse des letzten **benutzten Speicherplatzes** auf dem Stack
- **PUSH** (Daten auf den Stack legen) **dekrementiert** den Stackpointer
- **POP** (Daten vom Stack holen) **inkrementiert** den Stackpointer
- Deshalb wächst der Stack in Richtung kleinerer Adressen

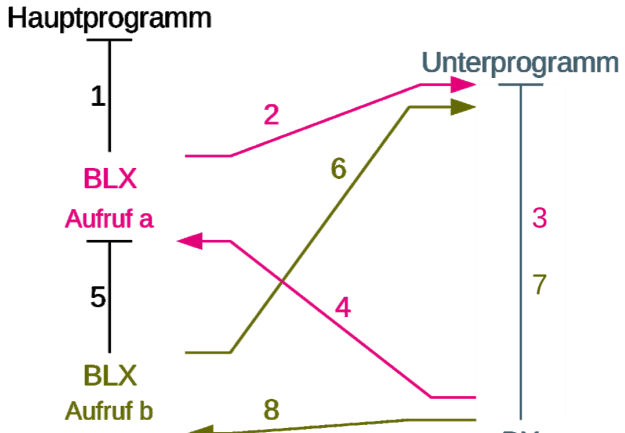


Stack-Initialisierung

```
MEMORY
{
  /* Define each memory region */
  PROGRAM_FLASH (rx) : ORIGIN = 0x0, LENGTH = 0x80000 /* 512K bytes (alias Flash) */
  SRAM_UPPER (rwx) : ORIGIN = 0x20000000, LENGTH = 0x10000 /* 64K bytes (alias RAM) */
  SRAM_LOWER (rwx) : ORIGIN = 0x1fff0000, LENGTH = 0x10000 /* 64K bytes (alias RAM2) */
}

SECTIONS
{
  /* Define output sections */
  ...
  _StackSize = 0x1000;
      /* Locate actual Stack in memory map -> from 0x20000000
         -> from 0x20000000 to 0x1FFFF000 (Stack is upside down) */
  .stack ORIGIN(SRAM_UPPER) + LENGTH(SRAM_UPPER) - _StackSize - 0: ALIGN(4)
  {
    _vStackBase = .;
    . = ALIGN(4);
    _vStackTop = . + _StackSize;
  } > SRAM_UPPER
  ...
}
```

Unterprogramme

Prinzip



- **BLX** (, BL_{32}) enthält `LR=PC`, `PC=Rx`
- **BX** enthält PC=LR oder POP {. . . , pc} wenn vorher push {. . . , lr}
- **C-Parameter**-Übergabe: in Registern oder auf dem Stack
- C: **lokale Variablen** auf dem Stack

Im Unterprogramm werden nicht gebrauchte (lokale) Variablen, welche aber nachher wieder benötigt werden, auf den Stack gelegt (zum Beispiel bei Scope-Wechsel über eine weitere Funktion). Der Kompiler entscheided dabei selbst, was genau gebraucht wird und was nicht.

Vor- & Nachteile

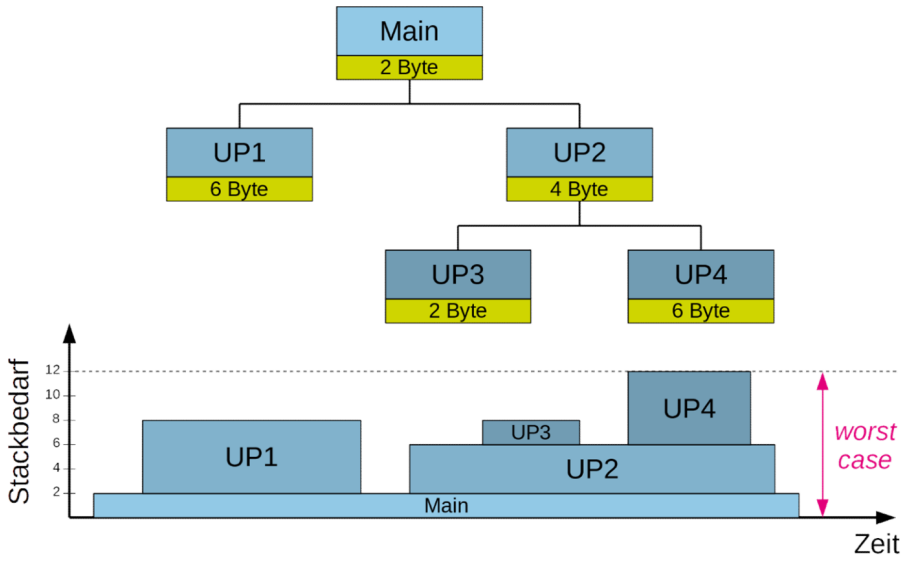
Vorteile

- Wiederkehrende Befehlsfolgen sind nur einmal im Speicher abgelegt ⇒ **geringerer Speicherbedarf**
- Wiederkehrende Befehlsfolgen werden nur einmal programmiert und getestet ⇒ **kleinerer Entwicklungsaufwand**
- Programme können modular aufgebaut werden ⇒ **kleineres Fehlerrisiko**
- Programme können von mehreren Personen parallel entwickelt werden ⇒ **höhere Produktivität im Team**
- Teilprogramme können unabhängig voneinander übersetzt werden ⇒ **kürzere Compile-Zeit, Bibliotheken mit Standardfunktionen**

Nachteile

- Der Aufruf des Unterprogramms, Parameterübergabe und Rücksprung brauchen Zeit ⇒ **langsamere Programmausführung**

Stackgrössen bei verschachtelten Unterprogrammen



Interrupts

> **!** Flag zurücksetzen nicht vergessen!
>
> Immer direkt nachdem die Interrupt-Funktion aufgerufen wurde, das Flag zurücksetzen, damit der gleiche Interrupt erneut erkannt werden kann.

```
void FTM0_IRQHandler(void) {
  FTM0->SC &= ~FTM_SC_TOF_MASK;
  ...
}
```

Polling & Interrupt

Bei **Polling** wird das Interrupt-Flag konstant abgefragt und beim aktiven Flag wird es gelöscht. Beim **Interrupt** springt der Programmzähler an die entsprechende IRQ-Adresse und man muss **manuell** das Flag löschen.

Interrupt unterbricht und **Polling** hat eine **Zeitverzögerung** (da z.B. das Programm die Endlosschleife durcharbeiten muss).

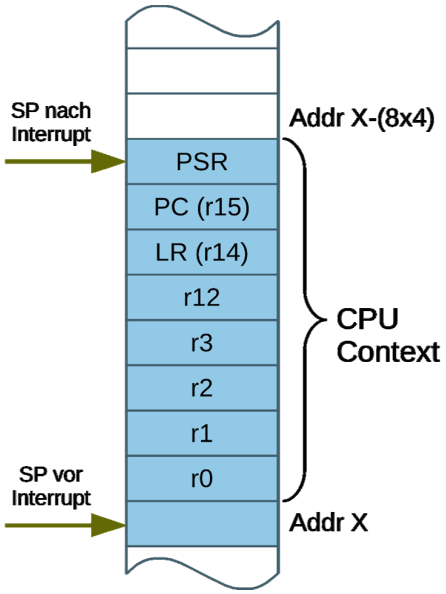
```
int main (void) {

}

void FTM3_IRQHandler(void) {
  FTM3->SC &= ~FTM_SC_TOF_MASK; // clear Interrupt flag
  ...
}
```

Statusrettung

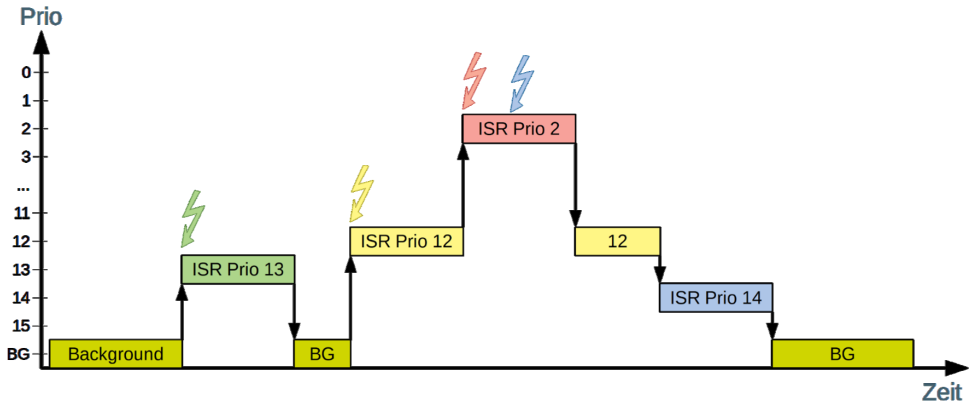
- Bei Eintritt in eine ISR wird der **CPU-Status automatisch auf den Stack** gerettet:
- ISR und Unterprogramm-Aufruf unterscheiden sich wie folgt:

	ISR	Unterprogramm
Aufruf	spontan	durch BL/BLX
Statusrettung	automatisch	durch Programm
Rücksprung	automatisch	durch BX

- Am Ende der ISR werden die Informationen auf dem Stack automatisch zurück in die CPU geholt (PULL)



Interrupt Aktivieren/Deaktivieren & Prioritäten setzen

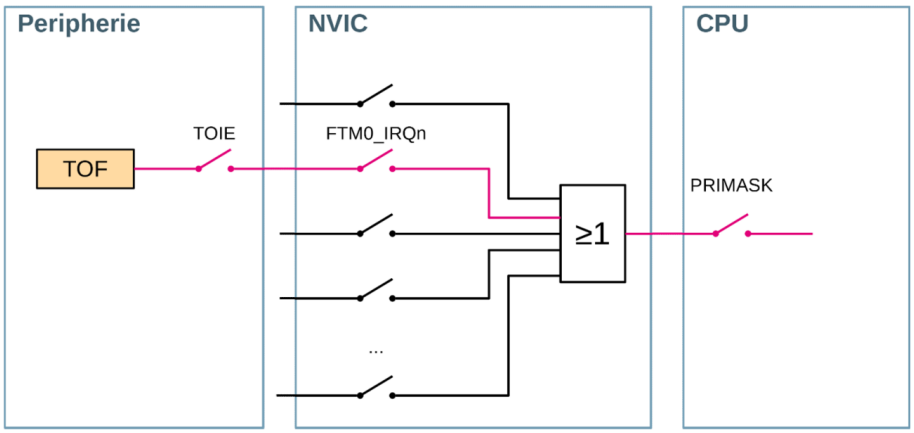


Die Interrupt-Prioritäten werden mit folgenden

```
NVIC_SetPriority(FTM3_IRQn, 10); // set interrupt priority: 0..15, 0=max Prio
NVIC_EnableIRQ(FTM3_IRQn); // enable interrupt
NVIC_DisableIRQ(FTM3_IRQn); // disable interrupt
```

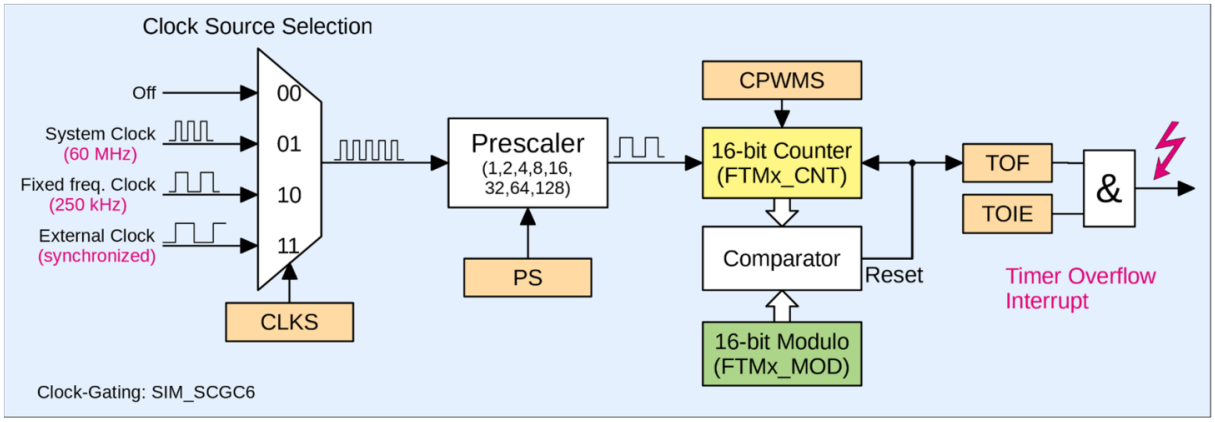
Interrupt-Freigabelogik

Damit ein Interrupt ausgeführt werden kann, muss dieser im **NVIC aktiviert** werden. Wird dies nicht gemacht, springt auch der Interrupt nicht zum entsprechenden IRQ-Handler.



Timer

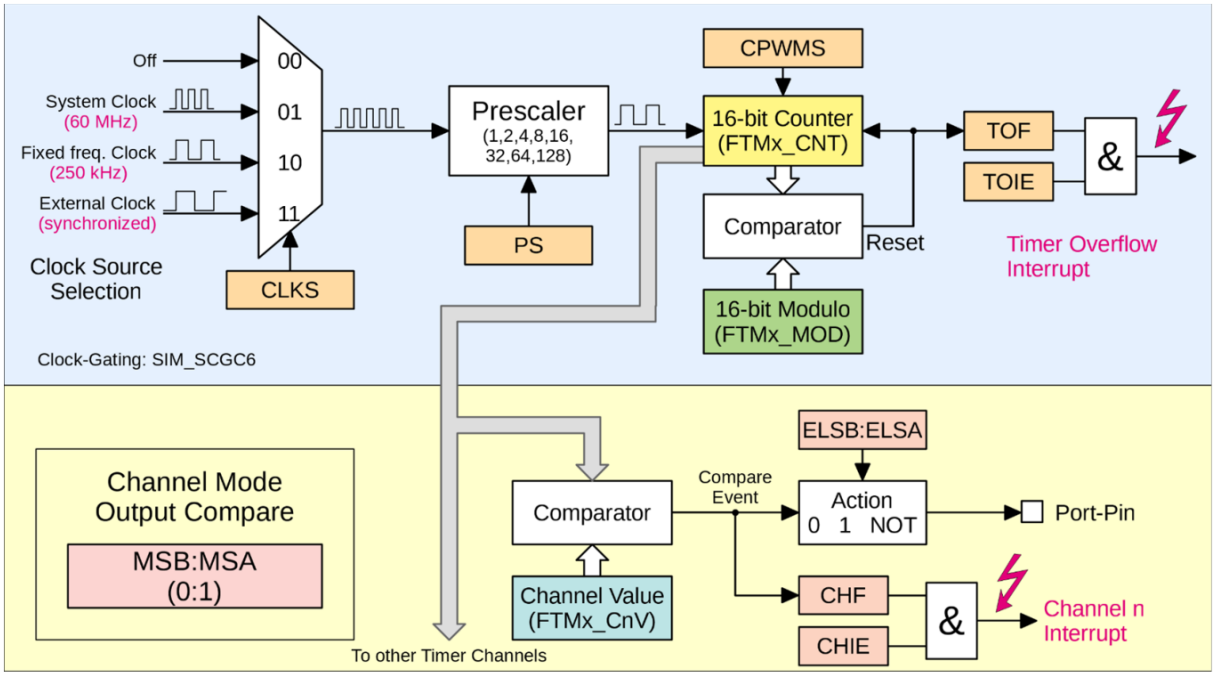
Modulo Counter



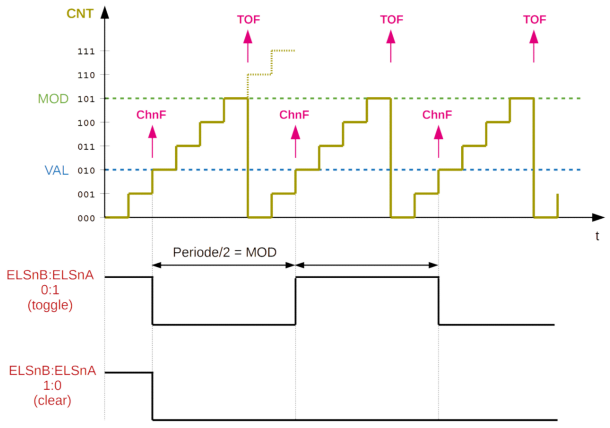
Adresse	Register	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
400x_x000	FTMx_SC									TOF	TOIE	CPWMS	CLKS		PS		
400x_x004	FTMx_CNT	Bit 15..0															
400x_x008	FTMx_MOD	Bit 15..0															

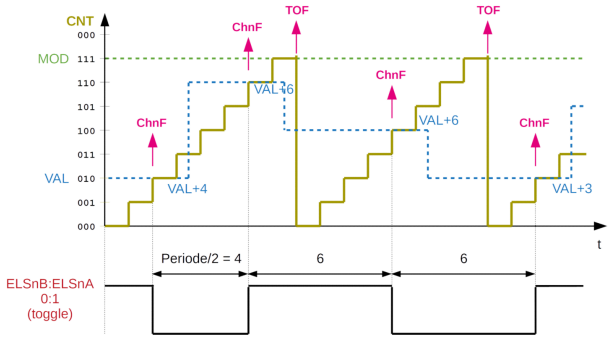
> **!** `MOD` vor `CLKS` Konfiguration
>
> Das FTM-Register `FTMx->MOD` muss vor dem Setzen der Clock-Source konfiguriert werden.

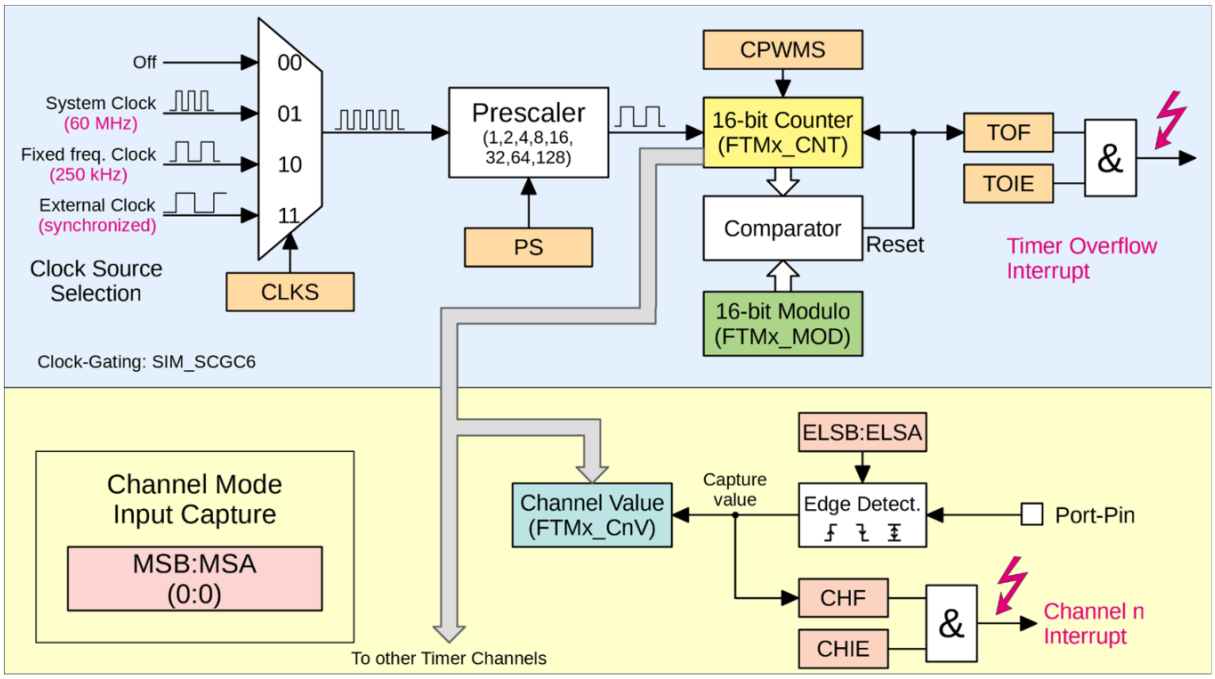
$$T_{TOF} = \frac{(\text{MOD}+1)\cdot \text{PS}}{f_{CLK}} \qquad \text{MOD} = \frac{T_{TOF}\cdot f_{CLK}}{\text{PS}} - 1$$



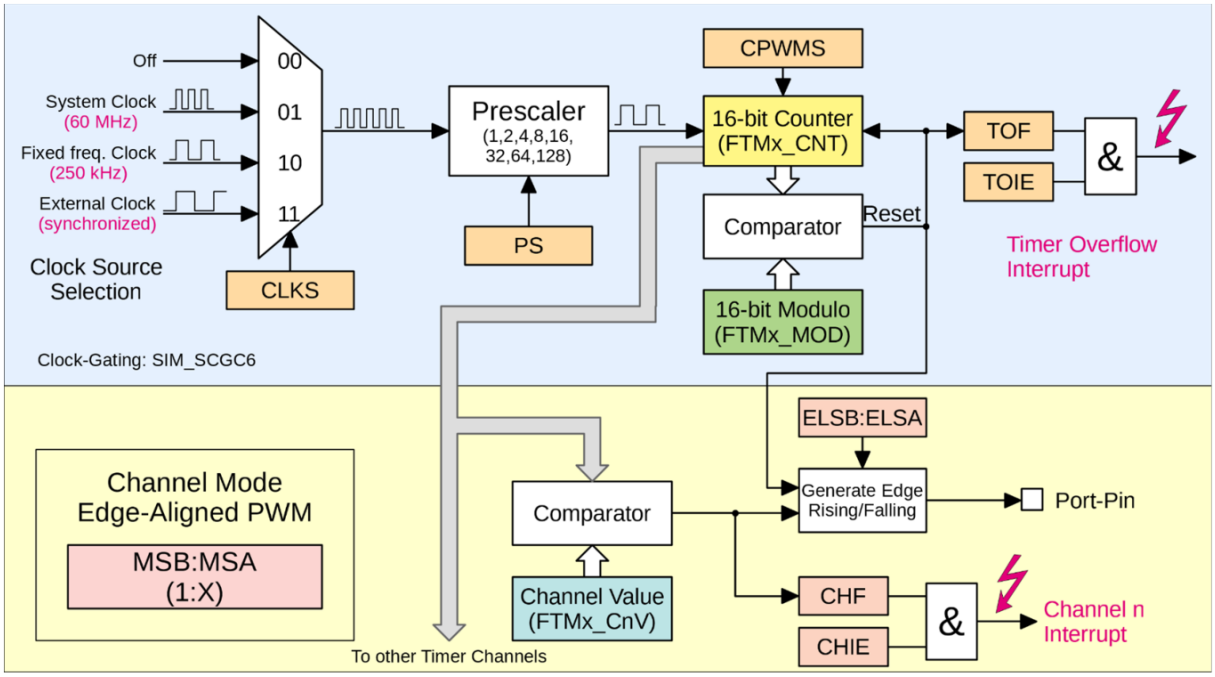
Adresse	Register	15	...	8	7	6	5	4	3	2	1	0
400x_xxxx	FTMx_CnSC				CHF	CHIE	MSB:MSA		ELSB:ELSA		ICRST	DMA
400x_xxxx	FTMx_CnV	Bit 15..0										



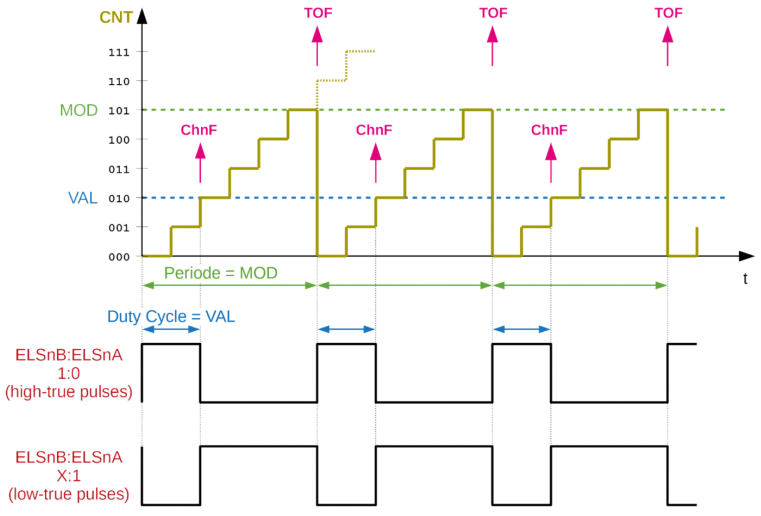




Adresse	Register	15	...	8	7	6	5	4	3	2	1	0
400x_xxxx	FTMx_CnSC				CHF	CHIE	MSB:MSA		ELSB:ELSA		ICRST	DMA
400x_xxxx	FTMx_CnV	Bit 15..0										

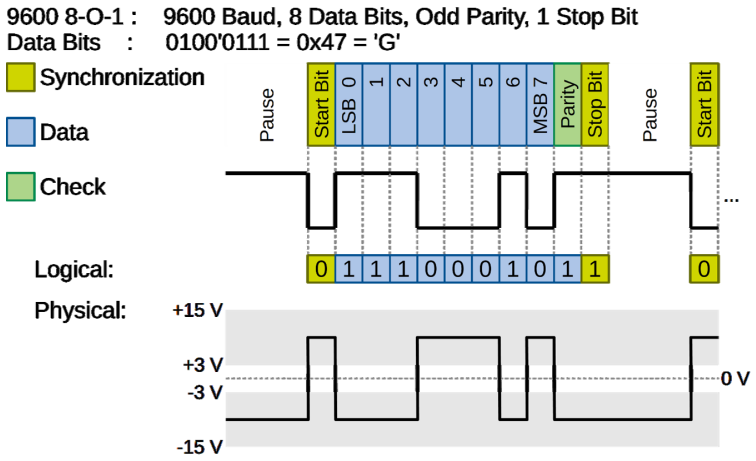


Adresse	Register	15	...	8	7	6	5	4	3	2	1	0
400x_xxxx	FTMx_CnSC				CHF	CHIE	MSB:MSA		ELSB:ELSA		ICRST	DMA
400x_xxxx	FTMx_CnV	Bit 15..0										

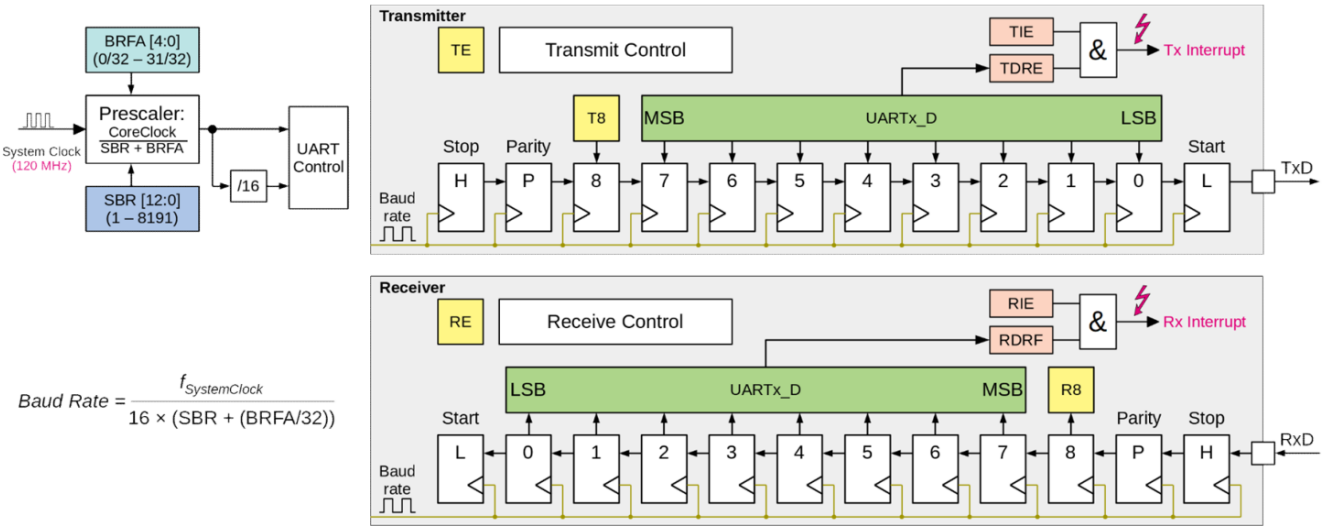


UART / RS232

> ⚠ Target UART-Schnittstelle
>
> **Pull-Up** Widerstand und der **Open-Drain** in der *Port Pin Configuration* aktivieren.



Übersicht



$$Baud\ Rate = \frac{f_{SystemClock}}{16 \times (SBR + (BRFA/32))}$$

Register	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	
UARTx_BDH	LBKDIE	RXEDGIE	0	SBR [12:8]					baud div H
UARTx_BDL	SBR [7:0]								baud div L
UARTx_C2	TIE	TCIE	RIE	ILIE	TE	RE	RWU	SBK	control reg 2
UARTx_C3	R8	T8	TXDIR	TXINF	ORIE	NEIE	FEIE	PEIE	control reg 3
UARTx_C4	MAEN1	MAEN2	M10	BRFA [4:0]					control reg 4
UARTx_S1	TDRE	TC	RDRF	IDLE	OR	NF	FE	PF	status reg 1
UARTx_D	Data								data reg r/W

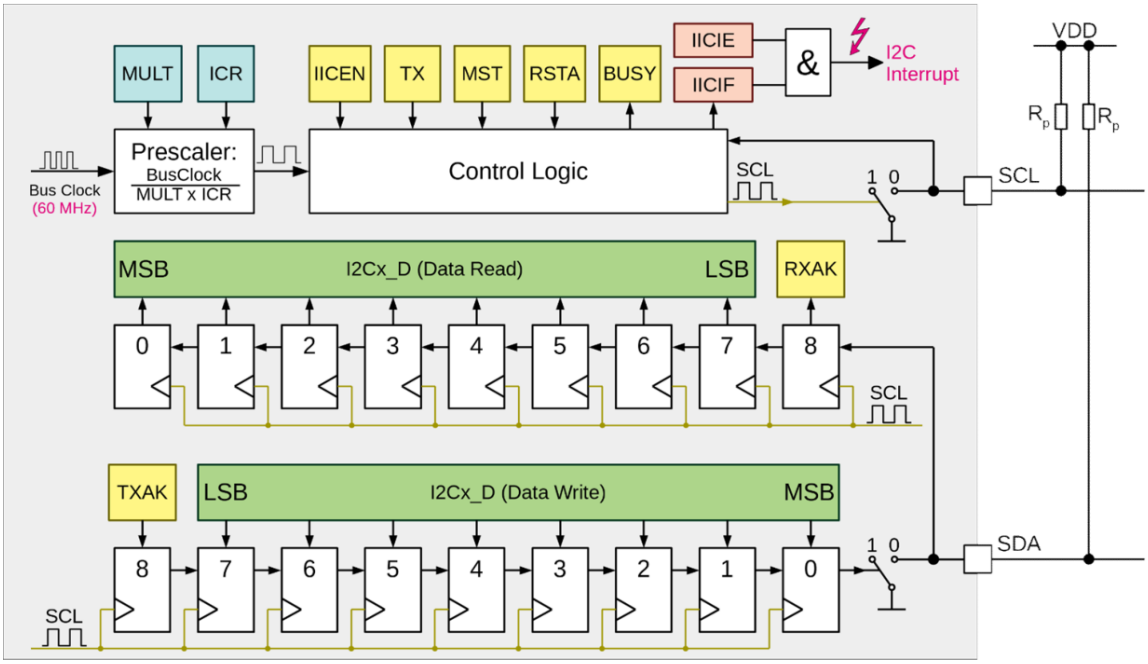
$$\text{Baud}_{UART} = \frac{f_{core(120\text{MHz})}}{16 \cdot (\text{SBR}[12:0] + \text{BRFD})} \qquad (\text{SBR}[12:0] + \text{BRFD}) = \frac{f_{core(120\text{MHz})}}{16 \cdot \text{Baud}_{UART}}$$

BRFA	BRFD	BRFA	BRFD	BRFA	BRFD	BRFA	BRFD
00000	$\frac{0}{32} = 0$	01000	$\frac{8}{32} = 0.25$	10000	$\frac{16}{32} = 0.5$	11000	$\frac{24}{32} = 0.75$
00001	$\frac{1}{32} = 0.03125$	01001	$\frac{9}{32} = 0.28125$	10001	$\frac{17}{32} = 0.53125$	11001	$\frac{25}{32} = 0.78125$

BRFA	BRFD	BRFA	BRFD	BRFA	BRFD	BRFA	BRFD
00010	$\frac{2}{32} = 0.0625$	01010	$\frac{10}{32} = 0.3125$	10010	$\frac{18}{32} = 0.5625$	11010	$\frac{26}{32} = 0.8125$
00011	$\frac{3}{32} = 0.09375$	01011	$\frac{11}{32} = 0.34375$	10011	$\frac{19}{32} = 0.59375$	11011	$\frac{27}{32} = 0.84375$
00100	$\frac{4}{32} = 0.125$	01100	$\frac{12}{32} = 0.375$	10100	$\frac{20}{32} = 0.625$	11100	$\frac{28}{32} = 0.875$
00101	$\frac{5}{32} = 0.15625$	01101	$\frac{13}{32} = 0.40625$	10101	$\frac{21}{32} = 0.65625$	11101	$\frac{29}{32} = 0.90625$
00110	$\frac{6}{32} = 0.1875$	01110	$\frac{14}{32} = 0.4375$	10110	$\frac{22}{32} = 0.6875$	11110	$\frac{30}{32} = 0.9375$
00111	$\frac{7}{32} = 0.21875$	01111	$\frac{15}{32} = 0.46875$	10111	$\frac{23}{32} = 0.71875$	11111	$\frac{31}{32} = 0.96875$

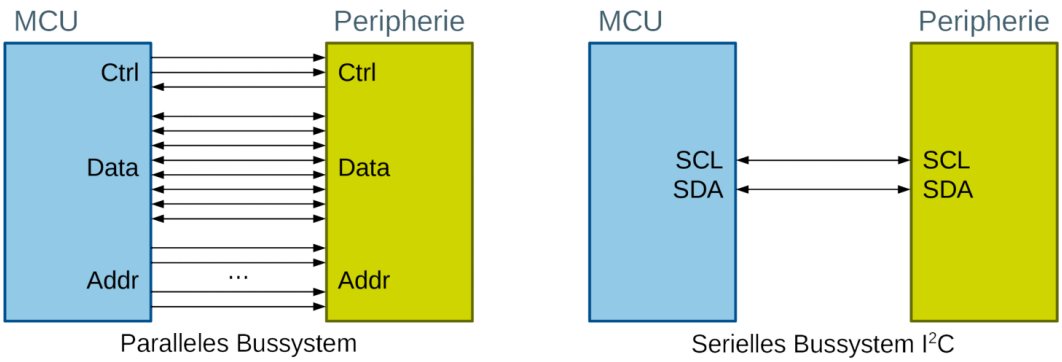
I²C

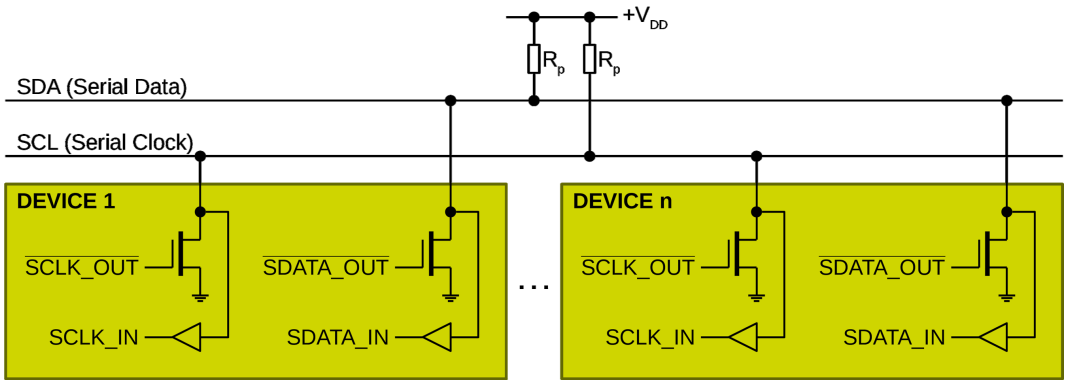
Übersicht



Register	7	6	5	4	3	2	1	0	Beschreibung
I2Cx_A1	AD[7:0]							-	slave address
I2Cx_F	MULT		ICR						freq. divider
I2Cx_C1	IICEN	IICIE	MST	TX	TXAK	RSTA	WUEN	DMAEN	control register
I2Cx_S	TCF	IIAS	BUSY	ARBL	RAM	SRW	IICIF	RXAK	status register
I2C_D	DATA								data register (r/w)

Hardware Verbindung





Berechnung

$$\text{I2C}_{baudrate} = \frac{f_{module}}{\text{MUL} \cdot \text{SCL dividier}}$$

> **!** Wichtig
>
> Für `ICR` wird nicht der Wert SCL divider genommen, sondern von der Tabelle *I2C divider and hold values*.

Eigenschaften

Bit-Geschwindigkeiten: 100 kBit/s, 400 kBit/s, 1 MBit/s, 3.4 MBit/s

Start-Stop Bedingungen: werden immer **vom Master generiert** und können durch andere Master und Slaves als **Protokollverletzung** von normalen Datenbits unterschieden werden.

Nach einer Start-Bedingung ist der Bus **busy**, nach einer Stop-Bedingung wieder **idle**.

> **i** Master Collision
>
> Wenn zwei Masters gleichzeitig eine Start-Konditionen ausführen und anfangen die Daten zu senden, muss entschieden werden, welcher Master weiterfahren darf. Da I2C eine Open-Drain Schaltung benötigt, gewinnt der Master, welcher die Leitung auf **LOW** zieht, während der andere ein **HIGH** senden möchte.

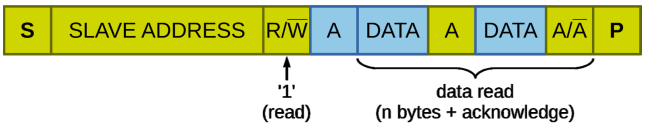
Clock Stretching: bedeutet, dass der Slave die Clock-Leitung auf **Low** zieht, da der Slave beschäftigt ist mit Datenverarbeitung oder einer Real-Time Funktion.

> **!** Wichtig
>
> **Clock Stretching** wird nicht durch ein NACK angedeutet! Aber der Slave **kann** NACK brauchen um einen *Busy*-Status anzugeben (Real-Time Processing).

Lesen / Read

```
adr = (uint8_t)adr << 1;
adr |= true;
```

Für die Lese-Kondition wird das $R/\overline{W}$ auf 1 gesetzt. Beim Lesen steuert der Master die Acknowledge-Antwort. `ACK` gilt für weiterlesen, `NACK` ($A = 0$) gilt für Ende.

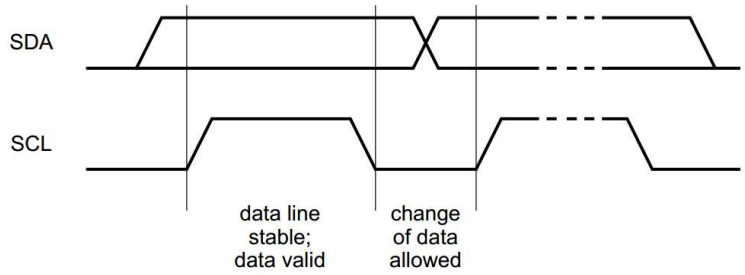


Schreiben / Write

```
adr = (uint8_t)adr << 1;
adr |= false;
```

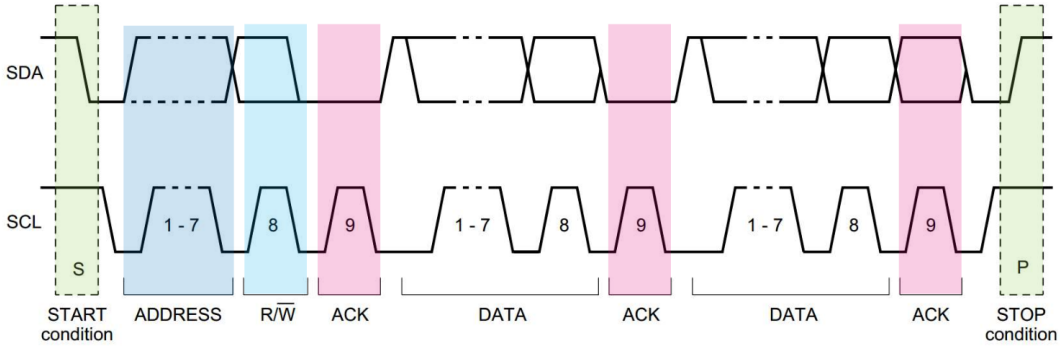
Für die Lese-Kondition wird das $R/\overline{W}$ auf 0 gesetzt. Beim Schreiben antwortet der Slave mit Acknowledge-Antwort. `ACK` gilt für weiterschreiben, `NACK` ($A = 0$) gilt für (1) Ungültige Daten/Befehl, (2) kann keine Daten mehr annehmen (3) Beschäftigt mit Verarbeitung.

Bit-Transfer



SDA wird nur geändert während SCL = 0 ist, und ausgewertet wenn SCL = 1 ist.

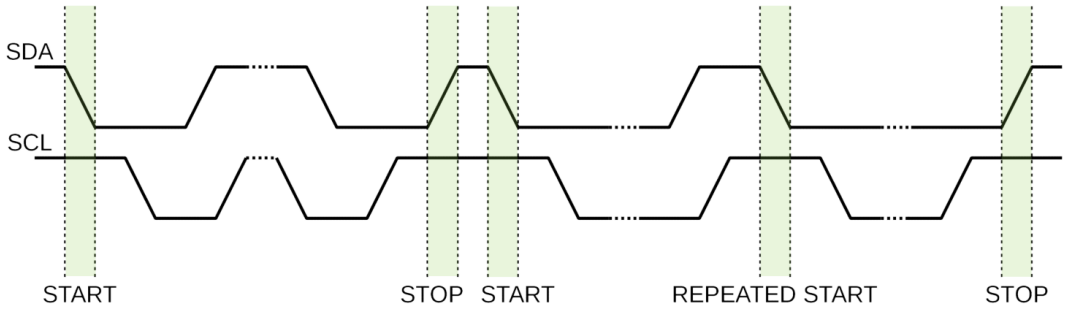
Byte-Transfer



Die Datenübetragung vom Transmitter zum Receiver erfolgt **Byte-weise** mit **MSB-first**. Nach der Abbildung entspricht Bit **1** dem **MSB** und Bit **8** dem **LSB**.

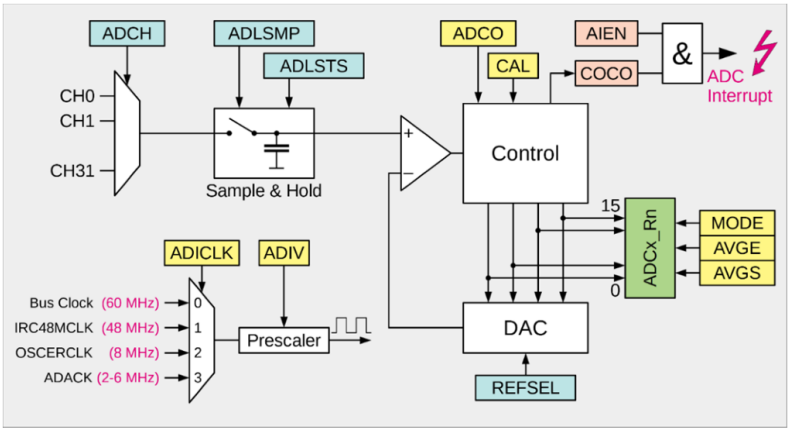
Start, Stop & Repeated Start

Das **Repeated Start** oder **Restart** wird verwendet, um den I2C-Bus nicht aufzugeben, was zum Beispiel bei Registerhüpfen und folglich Lesen verwendet wird.



A/D-Wandler

Blockdiagramm



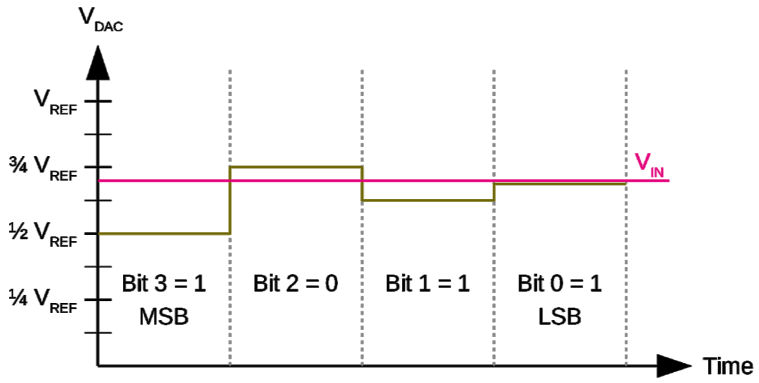
COCO: conversion complete
AIEN: Int. Enable
ADIV: 1/2/4/8 Divider
ADLSMP:
0:Short Sample Time
1:Long Sample Time
MODE: 8/12/10/16 Bits
ADLSTS: Long Sample Time select
REFSEL:
0:VREFH/L (ext. Pin)
1:VREF1 (internal)
CAL: Calibration
ADCO: 1x/continous
AVGE: HW Average Enable
AVGS: 4/8/16/32 Samples

Register	7	6	5	4	3	2	1	0	Beschreibung
ADCx_SC1A	COCO	AIEN	DIFF	ADCH					Status & Control 1
ADCx_CFG1	ADLPC	ADIV		ADLSMP	MODE		ADICLK		Configuration Reg. 1
ADCx_CFG2	0			MUXSEL	ADACKEN	ADHSC	ADLSTS		Configuration Reg. 2
ADCx_RA	D (Data result 16 Bit [15:0])								Data Result Register
ADC1_SC2	ADACT	ADTRG	ACFE	ACFGT	ACREN	DMAEN	REFSEL		Status & Control 2
ADC1_SC3	CAL	CALF	0		ADCO	AVGE	AVGS		Status & Control 3

Berechnung

$$U_{MESS} = \frac{U_{ref}}{B_{max}} \cdot B_R$$

- $U_{Resultat}$: Gemessene Spannung in Volt
- U_{ref}: Referenzspannung (1.195V)
- $B_{max} = 2^n - 1$: Maximale Auflösung mit n-Bits (single-ended 16, 12, 10, 8-Bits, differential 16, 13, 11, 9-Bits)
- B_R: Resultat vom AD-Wandler



- Das Verfahren fängt mit dem **MSB** an und nähert sich Schrittweise der Eingangsspannung an.
- Je **mehr Bits**, desto **hochauflösender** ist das **Resultat**.
- Während dem Verfahren wird die Eingangsspannung konstant gehalten (**Sample & Hold**)

Kontrollfragen

SW1 - MCU

1. Stellen Sie -4 im Zweier-Komplement als 16-bit Hex Zahl dar

$$\begin{array}{ll} 4 & 0\cdots0100 \\ \overline{4} & 1\cdots1011 \\ -4 & 1\cdots1100 \end{array}$$

$-4 \rightarrow$ `0xFFFC`

2. Wie viele Bit-Speicherplätze beinhaltet ein 32K x 8 Speicher?

$32\text{K} \cdot 8 = 256\text{K}$ Speicherplätze

- Wie viele Adressleitungen besitzt der Speicher?

$32\text{K} = 2^{15} \rightarrow 15$ Adressleitungen

- Wie lautet die höchste Adresse (Hex)?

$2^{15} =$ `0x8000` $\rightarrow 2^{15} - 1$ `0x7FFF`

3. Worin besteht der Unterschied zwischen einem Mikrocontroller und einem Mikroprozessor?

MCU besteht aus Peripherie, Speicher und CPU. MPU besteht nur aus CPU (& Cache).

4. Was sind die 3 Haupt-Systemteile eines jeden Mikrocontrollers?

CPU, Speicher, Peripherien

5. Welche Busse unterscheidet man einer MCU?

Adress-, Steuer-, Datenbus

- Welche der Busse sind Bidirektional?

Alle

6. Welche Funktionseinheiten beinhaltet eine CPU?

Rechenwerk (Arithmetic Logic Unit = ALU), Steuerwerk (CPU Control), Adresswerk (Adress Unit AU), Registern (Registers)

7. Welche Schritte beinhaltet der Befehlszyklus?

Fetch, Decode, Execute, nächste Adresse.

8. Welche Arten von Registern existieren in einer CPU?

Datenregister, Adressregister, Linkregister, Spezialregister, Freie Register (GPR = General Purpose Register)

SW2 - GPIO

1. Welche Haupt-Funktionsgruppen des MC-Car kennen Sie?
2. Wie kann man ein neues Image auf den MC-Car laden?
3. Unter welchen Umständen würden Sie eine MCU in Assembler programmieren?
4. Welche Informationen sind im Linkerfile (`*.ln`) abgelegt?
5. Wodurch unterscheiden sich "Befehl" und "Direktive" in einem Assembler Source File?

Befehl = An CPU gerichtet Direktive = An Assembler gerichtet

6. Was sind die Vorteile von Thumb 2 Assembler?

Mix zwischen 16- & 32-Bit Instruktionen.

7. Was ist UAL?

Unified Assembly Language

SW3

Programm Snippets

GPIO

</> LED/Ausgang konfigurieren & setzen

```
int main(void) {
  PORTC->PCR[8] = PORT_PCR_MUX(1); // set pin 8 to GPIO
  PORTC->PCR[11] = PORT_PCR_MUX(1); // set pin 11 to GPIO

  GPIOC->PSOR = (1 << 8) | (1 << 11); // since LEDs are active low. Set to turn off
  GPIOC->PDDR |= (1 << 8) | (1 << 11); // Configure pin 8 & 11 to output

  while(true) {
    GPIOC->PTOR = (1 << 8) | (1 << 11);
  }
}
```

</> Joystick/Input konfigurieren & auslesen

```
int main(void)
{
  PORTB->PCR[0] = PORT_PCR_MUX(1) | // Joystick Right
                  PORT_PCR_PE(1)  | // Enable Pull-Up/Down resistor
                  PORT_PCR_PS(1);   // Set to Pull-Up resistor
  PORTB->PCR[1] = PORT_PCR_MUX(1) | // Joystick Down
                  PORT_PCR_PE(1)  | // Enable Pull-Up/Down resistor
                  PORT_PCR_PS(1);   // Set to Pull-Up resistor

  PORTC->PCR[10] = PORT_PCR_MUX(1);
  PORTC->PCR[11] = PORT_PCR_MUX(1);

  GPIOC->PDDR |= (1 << 10) | (1 << 11);

  while(true)
  {
    uint32_t input = GPIOB->PDIR; // read input
    GPIOC->PDOR = (input & 0x03) << 10; // mask and shift to LED output
  }
```

```
}
```

Timer FTM3

</> Interrupt Handler

```
#define CHF_CHIE_MASK       (FTM_CnSC_CHF_MASK | FTM_CnSC_CHIE_MASK)
#define TOF_TOIE_MASK       (FTM_SC_TOF_MASK | FTM_SC_TOIE_MASK)

/**
 * Interrupt handler to distribute the different interrupt sources of the FTM:
 * - channel 0..7
 * - timer overflow
 */
void FTM3_IRQHandler(void)
{
  if ((FTM3->CONTROLS[0].CnSC & CHF_CHIE_MASK) == CHF_CHIE_MASK) FTM3CH0_IRQHandler();
  if ((FTM3->CONTROLS[1].CnSC & CHF_CHIE_MASK) == CHF_CHIE_MASK) FTM3CH1_IRQHandler();
  if ((FTM3->CONTROLS[2].CnSC & CHF_CHIE_MASK) == CHF_CHIE_MASK) FTM3CH2_IRQHandler();
  if ((FTM3->CONTROLS[3].CnSC & CHF_CHIE_MASK) == CHF_CHIE_MASK) FTM3CH3_IRQHandler();
  if ((FTM3->CONTROLS[4].CnSC & CHF_CHIE_MASK) == CHF_CHIE_MASK) FTM3CH4_IRQHandler();
  if ((FTM3->CONTROLS[5].CnSC & CHF_CHIE_MASK) == CHF_CHIE_MASK) FTM3CH5_IRQHandler();
  if ((FTM3->CONTROLS[6].CnSC & CHF_CHIE_MASK) == CHF_CHIE_MASK) FTM3CH6_IRQHandler();
  if ((FTM3->CONTROLS[7].CnSC & CHF_CHIE_MASK) == CHF_CHIE_MASK) FTM3CH7_IRQHandler();
  if ((FTM3->SC & TOF_TOIE_MASK) == TOF_TOIE_MASK) FTM3TOF_IRQHandler();
}
```

</> Initialisieren

```
void ftm3Init(void)
{
  // set clockgating for FTM3
  SIM->SCGC6 |= SIM_SCGC6_FTM3(1);

  // sets the modulo
  FTM3->MOD = FTM3_MODULO;

  // configure the timer with "system clock" as clocksource and with a "Prescaler"...
  // ...of 1 => 60 MHz
  FTM3->SC = FTM_SC_CLKS(1) |  FTM_SC_PS(0);

  // Enable FTM3 interrupt on NVIC with Prio: PRIO_FTM3 (defined in platform.h)
  NVIC_SetPriority(FTM3_IRQn, PRIO_FTM3);       // set interrupt priority
  NVIC_EnableIRQ(FTM3_IRQn);                    // enable interrupt
}
```

</> Kanäle

```
void init_ftm(void);

#define FTM3CH3_INTERVAL (6249)  // ~0.2ms -> T ~ 0.4ms
#define FTM3CH6_INTERVAL (31249) // ~1s -> T ~ 2s
```

```c
int main(void) {
  init_ftm();
  while(true){}
}

void init_ftm(void) {
  // enable clock gating for the FTM3 Module
  SIM->SCGC6 |= SIM_SCGC6_FTM3(1);

  // Set Channel Pins
  PORTD->PCR[3] = PORT_PCR_MUX(4); // FTM3-CH3
  PORTC->PCR[10] = PORT_PCR_MUX(3); // FTM3-CH6

  // before FTM3 is configure, channels are configured first
  // Configure Channels as 'Output-Compare Toggle Pin'
  FTM3->CONTROLS[3].CnSC |= FTM_CnSC_CHIE(0x1) | FTM_CnSC_MSx(0x1) | FTM_CnSC_ELSx(0x1);
  FTM3->CONTROLS[6].CnSC |= FTM_CnSC_CHIE(0x1) | FTM_CnSC_MSx(0x1) | FTM_CnSC_ELSx(0x1);

  // configure FTM3
  FTM3->MOD = 0; // 0x0 -> 0xFFFF
  FTM3->SC |= FTM_SC_CLKS(0x2) | FTM_SC_PS(0x3) | FTM_SC_TOIE(0x1);

  // Enable NVIC
  NVIC_SetPriority(FTM3_IRQn, 8);
  NVIC_EnableIRQ(FTM3_IRQn);
}

void FTM3_IRQHandler(void) {
  //check if channel flags have been raised.
  if(FTM3->CONTROLS[3].CnSC & FTM_CnSC_CHF_MASK) {
    FTM3->CONTROLS[3].CnSC &= ~FTM_CnSC_CHF_MASK;
    FTM3->CONTROLS[3].CnV += FTM3CH3_INTERVAL;
  }
  if(FTM3->CONTROLS[6].CnSC & FTM_CnSC_CHF_MASK) {
    FTM3->CONTROLS[6].CnSC &= ~FTM_CnSC_CHF_MASK;
    FTM3->CONTROLS[6].CnV += FTM3CH6_INTERVAL;
  }
}
```

UART UART0

```c
/**
 * the receive queue of this driver, implemented as ring buffer
 */
static char rxBuf[UART0_RX_BUF_SIZE];
static volatile uint16_t rxBufCount;
static uint16_t rxBufWritePos;
static uint16_t rxBufReadPos;

/**
 * the transmit queue of this driver, implemented as ring buffer
```

```
 */
static char txBuf[UART0_TX_BUF_SIZE];
static volatile uint16_t txBufCount;
static uint16_t txBufWritePos;
static uint16_t txBufReadPos;

#define ENABLE_UART0_INTERRUPTS()     NVIC_EnableIRQ(UART0_RX_TX_IRQn)
#define DISABLE_UART0_INTERRUPTS()    NVIC_DisableIRQEx(UART0_RX_TX_IRQn)
```

</> Interrupt Handler RX / TX

```
/**
 * @brief UART Interrupt Service Routine
 * - Received bytes are stored in the queue rxBuf
 * - Bytes in the queue txBuf are sent
 */
void UART0_RX_TX_IRQHandler(void)
{
  OnEnterUart0RxTxISR();
  uint8_t status = UART0->S1;
  uint8_t data = UART0->D;
  if (status & UART_S1_RDRF_MASK)
  {
    // store the received byte into receiver Queue (rxBuf)
    // but only if the queue isn't full!
    if (rxBufCount < UART0_RX_BUF_SIZE)
    {
      rxBuf[rxBufWritePos++] = data;
      rxBufCount++;
      if (rxBufWritePos == UART0_RX_BUF_SIZE) rxBufWritePos = 0;
    }
  }

  if (status & UART_S1_TDRE_MASK)
  {
    if (txBufCount > 0)
    {
      UART0->D = txBuf[txBufReadPos++];
      txBufCount--;
      if (txBufReadPos == UART0_TX_BUF_SIZE) txBufReadPos = 0;
    }
    else
    {
      UART0->C2 &= ~UART_C2_TIE_MASK;
    }
  }
  OnExitUart0RxTxISR();
}
```

Interrupt Handler Error

```
/**
 * Error Interrupt Service Routine
 * Clears the error flags.
 */
void UART0_ERR_IRQHandler(void)
{
  (void)UART0->S1;
  (void)UART0->D;
}
```

Zeichen schreiben

```
/**
 * Writes one Byte in the transmit buffer.
 *
 * @details
 *   Switching on the TIE interrupt causes an interrupt to be
 *   triggered immediately. The function blocks until there is
 *   space in the txBuf queue.
 *
 * @param[in] ch
 *   the byte to send
 */
void uart0WriteChar(char ch)
{
  while(txBufCount >= UART0_TX_BUF_SIZE);
  txBuf[txBufWritePos++] = ch;
  if (txBufWritePos == UART0_TX_BUF_SIZE) txBufWritePos = 0;
  DISABLE_UART0_INTERRUPTS();
  txBufCount++;
  ENABLE_UART0_INTERRUPTS();
  UART0->C2 |= UART_C2_TIE_MASK;
}
```

Buffer/Zeichenkette schrieben

```
/**
 * Writes a null terminated string in the send buffer. If the
 * string is null, the function returns immediately.
 *
 * @param[in] str
 *   the null terminated string to send
 */
void uart0Write(const char *str)
{
  if (str == NULL) return;
  while (*str != '\0') uart0WriteChar(*str++);
}
```

Zeichenkette schreiben mit Newline

```
/**
 * Writes a null terminated string in the send buffer. If the
 * string is null, only a new new line character is sent.
 *
 * @param[in] str
 *   the null terminated string to send
 */
void uart0WriteLine(const char *str)
{
  uart0Write(str);
  uart0WriteChar(NEW_LINE);
}
```

Zeichen lesen

```
/**
 * Reads one char out of the rxBuf queue. The function blocks
 * until there is at least one byte in the queue.
 *
 * @return
 *   the received byte
 */
char uart0ReadChar(void)
{
  char ch;
  while (rxBufCount == 0);
  ch = rxBuf[rxBufReadPos++];
  if (rxBufReadPos == UART0_RX_BUF_SIZE) rxBufReadPos = 0;
  DISABLE_UART0_INTERRUPTS();
  rxBufCount--;
  ENABLE_UART0_INTERRUPTS();
  return ch;
}
```

Zeichenkette lesen

```
/**
 * Reads a null terminated string out of the rxBuf queue. The
 * function blocks until a new Line character has been received
 * or the length has been exceeded.
 *
 * @details
 *   the new line character will be replaced with a '\0' to
 *   terminate the string.
 *
 * @param[out] *str
 *   pointer to a char array to store the received string
 * @param[in] length
 *   the length of the str char array.
 * @returns
 *   the length of the received string.
```

```
 */
uint16_t uart0ReadLine(char *str, uint16_t length)
{
  uint16_t i;
  for (i=1; i<length; i++)
  {
    *str = uart0ReadChar();
    if (*str == NEW_LINE)
    {
      *str = '\0';
      break;
    }
    str++;
  }
  return i;
}
```

</> Newline erhalten

```
/**
 * This functions checks, if there is a new line character
 * in the rxBuf queue.
 *
 * @returns
 *   TRUE, if there is a new line character, otherweise FALSE.
 */
bool uart0HasLineReceived(void)
{
  uint16_t i;
  uint16_t index = rxBufReadPos;

  for (i=0; i<rxBufCount; i++)
  {
    if (rxBuf[index++] == NEW_LINE) return TRUE;
    if (index == UART0_RX_BUF_SIZE) index = 0;
  }
  return FALSE;
}
```

</> Anzahl zulesende Zeichen

```
/**
 * Returns the number of bytes in the receiver queue.
 *
 * @returns
 *   the number of bytes in the receiver queue.
 */
uint16_t uart0RxBufCount(void)
{
  return rxBufCount;
}
```

```
/**
 * initializes the uart with the desired baud rate.
 *
 * @details
 *   The uart connection between the debugger and the target
 *   device is on the MC-Car and the tinyK22 different:
 *
 *   MC-Car:
 *   - PTA1 Mux2:UART0_RX
 *   - PTA2 Mux2:UART0_TX
 *
 *   tinyK22:
 *   - PTC3 Mux3:UART1_RX, (MUX7:LPUART1_RX)
 *   - PTC4 Mux3:UART1_TX, (MUX7:LPUART1_TX)
 */
void uart0Init(uint16_t baudrate)
{
  txBufReadPos = txBufWritePos = txBufCount = 0;
  rxBufReadPos = rxBufWritePos = rxBufCount = 0;

  // configure clock gating (Kinetis Reference Manual p277) KRM277
  SIM->SCGC4 |= SIM_SCGC4_UART0_MASK;

  // configure port multiplexing, enable Pull-Ups and enable OpenDrain (ODE)!
  // OpenDrain is needed to ensure that no current flows from Target-uC to the Debugger-uC
  PORTA->PCR[1] = PORT_PCR_MUX(2) | PORT_PCR_PE(1) | PORT_PCR_PS(1) | PORT_PCR_ODE_MASK;
  PORTA->PCR[2] = PORT_PCR_MUX(2) | PORT_PCR_PE(1) | PORT_PCR_PS(1) | PORT_PCR_ODE_MASK;

  // set the baudrate into the BDH (first) and BDL (second) register. KRM1215ff
  uint32_t bd = (CORECLOCK / (16 * baudrate));
  UART0->BDH = (bd >> 8) & 0x1F;
  UART0->BDL = bd & 0xFF;

  // enable uart receiver, receiver interrupt and transmitter as well as
  // enable and set the rx/tx interrupt in the nested vector interrupt controller (NVIC)
  UART0->C2 = UART_C2_RIE_MASK | UART_C2_RE_MASK | UART_C2_TE_MASK;
  NVIC_SetPriority(UART0_RX_TX_IRQn, PRIO_UART0);
  NVIC_EnableIRQ(UART0_RX_TX_IRQn);

  // enable the error interrupts of the uart and configure the NVIC
  UART0->C3 = UART_C3_ORIE_MASK | UART_C3_NEIE_MASK | UART_C3_FEIE_MASK;
  NVIC_SetPriority(UART0_ERR_IRQn, PRIO_UART0);
  NVIC_EnableIRQ(UART0_ERR_IRQn);
}
```

I²C

Start

```
/**
 * Generates a start condition on the I2C-Bus
 *
 * @param [in] adr
 *   the 7 bit slave address
 * @param [in] read
 *   FALSE => write
 *   TRUE => read
 * @return
 *   EC_SUCCESS       the slave answered with an ACK
 *   C_I2C_NO_ANSWER  no answer from a slave
 */
tError i2cStart(uint8_t adr, bool read)
{
    adr = (uint8_t)adr << 1;                // combine address with r/w bit
    adr |= read;                            // add r/w bit

    while (I2C0->S & I2C_S_BUSY_MASK);      // wait until i2c bus is idle. Necessary if two
                                            // transmissions follow one after the other

    I2C0->S |= I2C_S_IICIF_MASK;            // clear the interrupt flag
    I2C0->C1 |= I2C_C1_TX_MASK;             // change to transmit mode
    I2C0->C1 |= I2C_C1_MST_MASK;            // generate the start-condition
    I2C0->D = adr;                          // send address with r/w bit
    while (!(I2C0->S & I2C_S_IICIF_MASK));  // wait for the transfer to complete
    I2C0->S |= I2C_S_IICIF_MASK;            // clear the interrupt flag

    if (I2C0->S & I2C_S_RXAK_MASK) {        // Check if an ACK has been received
        I2C0->C1 &= ~I2C_C1_MST_MASK;       // generate Stop-Condition
        return EC_I2C_NO_ANSWER;            // NAK => abort
    }
    return EC_SUCCESS;
}
```

Repeated Start

```
/**
 * Generates a repeated start condition on the I2C-Bus
 *
 * @param [in] adr
 *   the slave address (7bit)
 * @param [in] read
 *   FALSE => write mode
 *   TRUE =>  read mode
 * @return
 *    EC_SUCCESS        if the slave answered with an ACK
 *    EC_I2C_NO_ANSWER  no answer from a slave
 */
tError i2cRepeatedStart(uint8_t adr, bool read)
```

```
{
  adr = (uint8_t)adr << 1;
  adr |= read;

  I2C0->C1 |= I2C_C1_RSTA_MASK;              // generate a repeated start condition

  I2C0->D = adr;                             // sends the address with the read/write bit
  while (!(I2C0->S & I2C_S_IICIF_MASK));     // wait for the transfer to complete
  I2C0->S |= I2C_S_IICIF_MASK;               // clear the interrupt flag

  if (I2C0->S & I2C_S_RXAK_MASK) {           // Check if an ACK has been received
    I2C0->C1 &= ~I2C_C1_MST_MASK;            // generate Stop-Condition
    return EC_I2C_NO_ANSWER;                // NAK => abort
  }
  return EC_SUCCESS;
}
```

</> Letztes Byte auslesen

```
/**
 * Receives the last byte from a slave
 *
 * For the last byte don't send an ACK to generate
 * the stop condition after
 *
 * @return
 *  the received byte
 */
uint8_t i2cReceiveLastByte(void)
{
  uint8_t data;
  I2C0->C1 &= ~I2C_C1_TX_MASK;            // change to receive mode
  I2C0->C1 |= I2C_C1_TXAK_MASK;           // Don't send an ACK after the last byte received
                                          // to generate a stop condition after

  data = I2C0->D;                         // dummy read to start the last transmission to
                                          // read the last byte after

  while (!(I2C0->S & I2C_S_IICIF_MASK));  // wait for the transfer to complete
  I2C0->S |= I2C_S_IICIF_MASK;            // clear the interrupt flag
  I2C0->C1 &= ~I2C_C1_MST_MASK;           // generate Stop-Condition
  data = I2C0->D;                         // read last received byte
  return data;
}
```

</> Bytes von Buffer senden

```
/**
 * Transmitts a buffer. It is assumed that
 * i2cStart() was successful before.
 *
 * @param [in] buf
 *   the data to send
```

```
 * @param [in] length
 *   the number of bytes to send
 *
 * @return
 *    EC_SUCCESS if the slave answered with an ACK
 *    EC_I2C_NAK if the slave answered with a NAK
 */
tError i2cSendData(uint8_t *buf, uint8_t length)
{
  uint8_t i;
    for (i=0; i<length; i++)
    {
        I2C0->D = buf[i];                       // start the transmission of a databyte
        while (!(I2C0->S & I2C_S_IICIF_MASK));  // wait for the transfer to complete
        I2C0->S |= I2C_S_IICIF_MASK;            // clear the interrupt flag
        if (I2C0->S & I2C_S_RXAK_MASK) {        // Check if an ACK has been received
            I2C0->C1 &= ~I2C_C1_MST_MASK;       // generate Stop-Condition
            return EC_I2C_NAK;                  // NAK => abort
        }
    }
    return EC_SUCCESS;
}
```

</> Mehrere Bytes lesen und in Buffer abspeichern

```
/**
 * Reads data from a I2C-Device. It is assumed that
 * i2cStart() was successful before.
 *
 * This function generates a stop condition at the end!
 *
 * @param [out] buf
 *   the buffer for the receiving data
 * @param [in] length
 *   the number of bytes to receive
 */
void i2cReceiveData(uint8_t *buf, uint8_t length)
{
  uint8_t i;
  I2C0->C1 &= ~I2C_C1_TX_MASK;               // change to receive mode

  if (length > 1) {
    I2C0->C1 &= ~I2C_C1_TXAK_MASK;           // generates ACK's from now on
    buf[0] = I2C0->D;                        // dummy read to start the transmission and
                                             // to read the first byte

    while (!(I2C0->S & I2C_S_IICIF_MASK));   // wait for the transfer to complete
    I2C0->S |= I2C_S_IICIF_MASK;             // clear the interrupt flag

    for (i=0; i<length-2; i++) {             // read the bytes in a loop
      buf[i] = I2C0->D;                      // read the received byte and start a...
                                             // ...new transmission...
      while (!(I2C0->S & I2C_S_IICIF_MASK)); // wait for the transfer to complete
```

```
      I2C0->S |= I2C_S_IICIF_MASK;           // clear the interrupt flag
    }

    I2C0->C1 |= I2C_C1_TXAK_MASK;            // Don't send an ACK after the last byte received
                                             // to generate a stop condition after

    buf[length - 2] = I2C0->D;               // read the received byte and start a new
                                             // transmission...
    while (!(I2C0->S & I2C_S_IICIF_MASK));   // wait for the transfer to complete
    I2C0->S |= I2C_S_IICIF_MASK;             // clear the interrupt flag

    I2C0->C1 &= ~I2C_C1_MST_MASK;            // generate the stop condition
    buf[length-1] = I2C0->D;                 // read the last byte
  }
  else {
    I2C0->C1 |= I2C_C1_TXAK_MASK;            // Don't send an ACK after the last byte received
                                             // to generate a stop condition after

    buf[0] = I2C0->D;                        // dummy read to start the last transmission to
                                             // read the last byte after

    while (!(I2C0->S & I2C_S_IICIF_MASK));   // wait for the transfer to complete
    I2C0->S |= I2C_S_IICIF_MASK;             // clear the interrupt flag

    I2C0->C1 &= ~I2C_C1_MST_MASK;            // generate Stop-Condition
    buf[0] = I2C0->D;                        // read last received byte
  }
}
```

</> Stop

```
/**
 * Generates the stop condition
 */
void i2cStop()
{
  I2C0->C1 &= ~I2C_C1_MST_MASK;              // Generate Stop-Condition
  I2C0->S |= I2C_S_IICIF_MASK;               // Clear interrupt flag
}
```

</> Test (Komposit Funktionen)

```
/**
 * Checks if a device is on the I2C-Bus and answers with an ACK
 *
 * @param [in] adr
 *   the I2C-Bus address of the device
 *
 * @return
 *   EC_SUCCESS        the slave answered with an ACK
 *   EC_I2C_NO_ANSWER  no answer from a slave
 */
tError i2cTest(uint8_t adr)
```

```
{
  tError result;
  result = i2cStart(adr, FALSE);
  if (result != EC_SUCCESS) return result;

  i2cStop();
  return EC_SUCCESS;
}
```

</> Daten auslesen

```
 /**
 * Reads data from an I2C-Device with an additional command byte
 * which is send before
 *
 * @param [in] adr
 *   the I2C-Bus address of the device
 * @param [in] cmd
 *   the command byte or register byte to send before
 * @param [out] data
 *   the data to send
 * @param [in] length
 *   the number of bytes to send
 *
 * @return
 *   EC_SUCCESS       the slave answered with an ACK
 *   EC_I2C_NAK       if the slave answered with a NAK
 *   EC_I2C_NO_ANSWER no answer from the slave
 */
tError i2cReadCmdData(uint8_t adr, uint8_t cmd, uint8_t *data, uint8_t length)
{
  tError result;
  result = i2cStart(adr, FALSE);            // send I2C-Address
  if (result != EC_SUCCESS) return result;

  result = i2cSendData(&cmd, 1);            // send command byte
  if (result != EC_SUCCESS) return result;

  result = i2cRepeatedStart(adr, TRUE);     // repeated start to change the direction
                                            // from write to read
  if (result != EC_SUCCESS) return result;

  i2cReceiveData(data, length);             // read the data & generate the stop condition
  return EC_SUCCESS;
}
```

</> Daten schreiben

```
/**
 * Writes data to an I2C-Device with an additional command byte
 * which is send before
 *
 * @param [in] adr
```

```
 *   the I2C-Bus address of the device
 * @param [in] cmd
 *   the command byte or register byte to send before
 * @param [in] data
 *   the buffer for the receiving data
 * @param [in] length
 *   the number of bytes to receive
 * @return
 *   EC_SUCCESS       the slave answered with an ACK
 *   EC_I2C_NAK       if the slave answered with a NAK
 *   EC_I2C_NO_ANSWER no answer from the slave
 */
tError i2cWriteCmdData(uint8_t adr, uint8_t cmd, uint8_t *data, uint8_t length)
{
  tError result;
  result = i2cStart(adr, FALSE);            // send I2C-Address
  if (result != EC_SUCCESS) return result;

  result = i2cSendData(&cmd, 1);            // send command byte
  if (result != EC_SUCCESS) return result;

  result = i2cSendData(data, length);       // send data
  if (result != EC_SUCCESS) return result;

  i2cStop();                                // generate stop condition
  return EC_SUCCESS;
}
```

</> Initialisieren (400kBit/s)

```
/**
 * Initializes the I2C-Bus I2C0 to ~400kBit/s
 */
void i2cInit(void)
{
  // configure clock gating for i2c bus
  SIM->SCGC4 |= SIM_SCGC4_I2C0(1);

  // configure MUX for SCL & SDA
  PORTE->PCR[24] = PORT_PCR_MUX(5) | PORT_PCR_ODE(1);
  PORTE->PCR[25] = PORT_PCR_MUX(5) | PORT_PCR_ODE(1);

  // configure i2c clock (frequency divider register) to 400 kHz
  I2C0->F = 0x1C;  // 60 MHz/(1*160) = 375kHz => MULT=0, ICR=1D

// KRM138 => High Drive only PTB0-1 PTD4-7, PTC3-4
//  I2C0_C2 |= I2C_C2_HDRS(1);
//  PORTE_PCR24 |= PORT_PCR_DSE(1);
//  PORTE_PCR25 |= PORT_PCR_DSE(1);

  // enable i2c bus
  I2C0->C1 |= I2C_C1_IICEN(1);
}
```

A/D Wandler

ADC Kanäle

```
// determine the ADC1 channels
// see KRM99ff
#define ADC_CH_TEMPERATURE        26
#define ADC_CH_CURRENT            19
#define ADC_CH_BAT_VOLTAGE         0

#define REFERENCE_VOLTAGE    1195000  // 1'195'000uV = 1.195V
```

16-Bit Wert lesen

```
/**
 * Performs one A/D conversion for the specified channel with 16 bit resolution.
 * The function blocks until the conversion has been finished.
 *
 * @param[in]
 *   the channel number to convert
 * @return
 *   a 16bit value
 */
uint16_t adcGet16BitValue(uint8_t channel)
{
  // implement the function code as follows:
  // 1. start a conversion
  // 2. wait until conversion has been finished
  // 3. return the result.
  ADC1->SC1[0] = channel;                          // ADC1_SC1A
  while (!(ADC1->SC1[0] & ADC_SC1_COCO_MASK)){}    // wait until conversion has been finished
  return ADC1->R[0];                               // ADC1_RA;
}
```

Analogspannung lesen

```
/**
 * Performs a A/D conversation and returns the measured voltage of this channel.
 * The maximum voltage is the REFERENCE_VOLTAGE defined above.
 *
 * @param[in]
 *   the channel number to convert
 * @return
 *   the measured voltage in uV (0..REFERENCE_VOLTAGE)
 */
uint32_t adcGetVoltage(uint8_t channel)
{
  // calculate the measured voltage
  // 1. perform a conversion of the desired channel
  // 2. calculate and return the voltage in uV.
  uint16_t value = adcGet16BitValue(channel);
  uint64_t uV = ((uint64_t)REFERENCE_VOLTAGE * value) / 65535;
  return (uint32_t)uV;
```

```
}
```

Strom messen

```
/**
 * Returns the power consumption of the MC-Car
 *
 *            Ushunt [mV]         Ushunt
 * I [mA] = --------------- = ------------ x Yshunt
 *            Rshunt [Ohm]      100 * 1000
 *
 * @return
 *   the power consumption in mA
 */
uint16_t adcGetCurrent(void)
{
  uint32_t uShunt = adcGetVoltage(ADC_CH_CURRENT);
  uint16_t iShunt = uShunt * 68 / (1000 * 100) - 15;
  return iShunt;
}
```

Temperatur messen

```
/**
 * Reads and returns the temperature of the microcontroller
 *
 * @returns
 *   the temperature in 0.1°C => 237 = 23.7°C
 */
int16_t adcGetTemperature(void)
{
  // determine the temperature of the microcontroller
  // 1. perform a adc conversion
  // 2. calculate the temperatur. KRM804 & MK22 Datasheet p42
  int32_t vTemp = adcGetVoltage(ADC_CH_TEMPERATURE);
  int16_t temp = 250 - ((vTemp - 716000) / 162);
  return temp;
}
```

Batterie-Spannung auslesen

```
/**
 * Returns the battery voltage
 *
 * @return
 *   the battery voltage in mV
 */
uint16_t adcGetBatVoltage(void)
{
  // determine the battery voltage
  // 1. perform a adc conversion
  // 2. calculate the voltage (see also the MC-Car schematic)
  uint32_t uAdc = adcGetVoltage(ADC_CH_BAT_VOLTAGE);
```

```
  uint16_t uBat = (uAdc * 37) / (10 * 1000);
  return uBat;
}
```

</> A/D Wandler initialisieren

```
void adcInit(void)
{
  uint16_t calib;

  // init voltage reference module
  SIM->SCGC4 |= SIM_SCGC4_VREF_MASK;          // clock gating
  VREF->TRM |= VREF_TRM_CHOPEN(1);            // Chop oscillator enable
  VREF->SC = VREF_SC_VREFEN(1)                // internal voltage reference enable
      | VREF_SC_ICOMPEN(1)                    // second order curvature compensation enable
      | VREF_SC_MODE_LV(1);                   // buffer mode: high power
  utilWaitUs(1);                              // Wait >300ns
  VREF->SC |= VREF_SC_REGEN(1);               // 1.75V regulator enable
  utilWaitUs(35000);                          // Wait 35ms

  // init adc
  // configure clock gating
  SIM->SCGC6 |= SIM_SCGC6_ADC1_MASK;

  // configure adc as follows:
  // set the ADCK to 7.5 MHz using the busClock
  // and configure 16 bit conversion with long sample time
  ADC1->CFG1 = ADC_CFG1_ADIV(3)               // Div 8 => ADCK = 60MHz/8 = 7.5MHz
      | ADC_CFG1_ADLSMP(1)                    // Long sample time
      | ADC_CFG1_MODE(3)                      // 16 bit conversion
      | ADC_CFG1_ADICLK(0);                   // clock = Busclock
  ADC1->CFG2 = 0;

  // Clock p103: 00:Bus 60MHz, 01:AltCLK2=IRC48MHZ=48MHz, 10:ALTCLK=OSCERCLK=8MHz

  // select the alternate reference source (VREF)
  ADC1->SC2 = ADC_SC2_REFSEL(0x01);                  // alternate reference source (VREF)

  // configure hardware average of 32 samples
  ADC1->SC3 = (ADC_SC3_AVGE_MASK | ADC_SC3_AVGS(0x03));   // hardware average enable,
                                                          // 32 samples averaged
  // adc calibration
  ADC1->SC3 |= ADC_SC3_CAL(1);                 // Start calibration
  while (!(ADC1->SC1[0] & ADC_SC1_COCO_MASK)){}  // wait until calibration has been finished

  // set calib data
  calib = ADC1->CLP0 + ADC1->CLP1 + ADC1->CLP2 + ADC1->CLP3 + ADC1->CLP4 + ADC1->CLPS;
  calib = calib / 2;
  calib = calib | 0x8000;
  ADC1->PG = calib;

  calib = ADC1->CLM0 + ADC1->CLM1 + ADC1->CLM2 + ADC1->CLM3 + ADC1->CLM4 + ADC1->CLMS;
  calib = calib / 2;
```

```
  calib = calib | 0x8000;
  ADC1->MG = calib;
}
```

Beispiel Polling & Interrupt

```
#include "platform.h"

void init_ftm(void);
void init_gpio(void);
void init_interrupt(void);

//#define ENABLE_INTERRUPT

int main(void) {
  init_gpio();
  init_ftm();

#ifdef ENABLE_INTERRUPT
  init_interrupt();
#endif



  while (true) {
#ifndef ENABLE_INTERRUPT
    if((FTM3->SC & FTM_SC_TOF_MASK)) {
      FTM3->SC &= ~FTM_SC_TOF_MASK; // clear flag
      GPIOC->PTOR = (1 << 10);
    }
#endif
  }
}

void init_ftm(void) {
  // enable clock gating for the FTM3 Module
  SIM->SCGC6 |= SIM_SCGC6_FTM3(1);

  // configure FTM3
  FTM3->MOD = 15624; // ! SET 'MOD' BEFORE 'CLKS'!
  FTM3->SC |= FTM_SC_CLKS(0x2) | FTM_SC_PS(0x3);

}

void init_gpio(void) {
  PORTC->PCR[10] = PORT_PCR_MUX(0x1);

  GPIOC->PDDR |= (1 << 10); // set
}

void init_interrupt(void) {
  FTM3->SC |= FTM_SC_TOIE(0x1);
```

```
  NVIC_SetPriority(FTM3_IRQn, 8);
  NVIC_EnableIRQ(FTM3_IRQn);
}

void FTM3_IRQHandler(void) {
  FTM3->SC &= ~FTM_SC_TOF_MASK; // clear flag
  GPIOC->PTOR = (1 << 10);
}
```

SPI

</> Other Stuff

```
#define SPI_CLK_POLARITY_LOW      0
#define SPI_CLK_POLARITY_HIGH     1
const uint16_t BR_VALUES[16] = {2, 4, 6, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192,
                                16384, 32768};
const uint8_t ASC_VALUES[16] = {0, 0, 1, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13};

#define ASIZE(x)  (sizeof(x) / sizeof((x)[0]))

typedef struct
{
  SPI_Type *spiBase;
  PORT_Type *spiClk;
  PORT_Type *spiMiSo;
  PORT_Type *spiMoSi;

} tSpi;

static tSpiPcs peripheralchipSelect;
```

</> Deaktivieren

```
void spiDisable(void)
{
  SPI0->MCR |= SPI_MCR_HALT_MASK;          // Set Halt Mode
  SPI0->MCR |= SPI_MCR_MDIS_MASK;          // Disable Module
  while (SPI0->SR & SPI_SR_TXRXS_MASK) {}  // wait until modul is stopped
}
```

</> Aktivieren

```
void spiEnable(void)
{
  SPI0->MCR &= ~SPI_MCR_MDIS_MASK;         // Enable Module
  SPI0->MCR |= SPI_MCR_CLR_TXF_MASK;       // Clear Tx FIFO
  SPI0->MCR |= SPI_MCR_CLR_RXF_MASK;       // Clear Rx FIFO
  SPI0->SR = SPI_SR_TCF_MASK               // Clear all interrupts 0x9A0A0000U
      | SPI_SR_EOQF_MASK
      | SPI_SR_TFUF_MASK
      | SPI_SR_TFFF_MASK
```

```
      | SPI_SR_RFOF_MASK
      | SPI_SR_RFDF_MASK;
  SPI0->MCR &= ~SPI_MCR_HALT_MASK;          // Disable Halt Mode

  while (SPI0->MCR & (SPI_MCR_HALT_MASK)){}
}
```

</> Geschwindigkeit setzen

```
void spiSetSpeed(bool dbr, uint8_t pbr, uint8_t br, uint8_t pcssck, uint8_t cssck)
{
  spiDisable();
  uint32_t ctar = SPI_CTAR_DBR(dbr)
    | SPI_CTAR_CPOL(0)                // The inactive stat value of SCK is low
    | SPI_CTAR_CPHA(0)                // Data is captured on the leading edge of SCK and
                                      // changed on the following edge
    | SPI_CTAR_PBR(pbr)               // Baud Rate Prescaler: 0..3 => /2, /3, /5, /7
    | SPI_CTAR_PASC(pcssck)
    | SPI_CTAR_ASC(cssck)
    | SPI_CTAR_PCSSCK(pcssck)         //
    | SPI_CTAR_CSSCK(cssck)  //
    //| SPI_CTAR_CSSCK(ASC_VALUES[br])  //
    | SPI_CTAR_BR(br);                // Baud Rate Scaler 0..15 => 2..32768

  SPI0->CTAR[0] = ctar | SPI_CTAR_FMSZ(7);   // 8 Bit Data Frame (n+1)
  SPI0->CTAR[1] = ctar | SPI_CTAR_FMSZ(15);  // 16 Bit Data Frame (n+1)
  spiEnable();
}
```

</> Baudrate setzen

```
void spiSetBaudrate(int32_t freq)
{
  uint8_t i;
  for (i=0; i<ASIZE(BR_VALUES); i++)
  {
    if (BUSCLOCK / BR_VALUES[i] <= freq)
    {
      spiDisable();
      // Baudrate: (BUSCLOCK/PBR) x ((1+DBR)/BR)
      SPI0->CTAR[0] =
          SPI_CTAR_DBR(1)         // Double Data Rate
          | SPI_CTAR_FMSZ(7)      // 8 Bit Data Frame (n+1)
          | SPI_CTAR_CPOL(0)      // The inactive stat value of SCK is low
          | SPI_CTAR_CPHA(0)      // Data is captured on the leading edge of SCK and changed
                                  // on the following edge
          | SPI_CTAR_PBR(0)       // Baud Rate Prescaler: 0 => /2
          | SPI_CTAR_PASC(0)
          // | SPI_CTAR_ASC(ASC_VALUES[i])
          | SPI_CTAR_PCSSCK(0)
          | SPI_CTAR_CSSCK(ASC_VALUES[i])
          | SPI_CTAR_BR(i);       // Baud Rate Scaler 0..15 => 2..32768
```

```
      SPI0->CTAR[1] =
          SPI_CTAR_DBR(1)         // Double Data Rate
          | SPI_CTAR_FMSZ(15)     // 16 Bit Data Frame (n+1)
          | SPI_CTAR_CPOL(0)      // The inactive stat value of SCK is low
          | SPI_CTAR_CPHA(0)      // Data is captured on the leading edge of SCK and changed
                                  // on the following edge
          | SPI_CTAR_PBR(0)       // Baud Rate Prescaler: 0 => /2
          | SPI_CTAR_PASC(0)
          //  | SPI_CTAR_ASC(ASC_VALUES[i])
          | SPI_CTAR_PCSSCK(0)
          | SPI_CTAR_CSSCK(ASC_VALUES[i]) //

          | SPI_CTAR_BR(i);       // Baud Rate Scaler 0..15 => 2..32768

      //SPI_CTAR_REG(SPI, 0) = (SPI_CTAR_REG(SPI, 0) & ~SPI_CTAR_BR_MASK) | SPI_CTAR_BR(i);
      spiEnable();
      break;
    }
  }
}
```

</> Byte schreiben & lesen

```
uint8_t spiWriteReadByte(uint8_t data)
{
  uint8_t rxByte;
  spiWriteReadBuf(&data, 1, &rxByte);
  return rxByte;
}
```

</> Buffer schreiben & lesen

```
void spiWriteReadBuf(uint8_t *txBuf, uint16_t length, uint8_t *rxBuf)
{
  bool has8Bit = 0;
  uint16_t rxCount = length;


  SPI0->MCR |= SPI_MCR_CLR_TXF(1) | SPI_MCR_CLR_RXF(1);
  SPI0->TCR = 0;

  SPI0->SR = SPI_SR_EOQF(1);   // Clear End Of Queue Flag (EOQF)

  if (length & 1) // Send 8 bit
  {
    has8Bit = TRUE;
    length--;
    if (txBuf)
    {
      SPI0->PUSHR = SPI_PUSHR_CONT(0)//length != 0)
        | SPI_PUSHR_CTAS(0)
        | SPI_PUSHR_EOQ(length == 0)
        | SPI_PUSHR_CTCNT(0)  // Clear the TCNT field
```

```
   // | SPI_PUSHR_PCS(peripheralchipSelect)
      | SPI_PUSHR_TXDATA(*txBuf++);
  }
  else
  {
    SPI0->PUSHR = SPI_PUSHR_CONT(0)//length != 0)
      | SPI_PUSHR_CTAS(0)
      | SPI_PUSHR_EOQ(length == 0)
      | SPI_PUSHR_CTCNT(0)  // Clear the TCNT field
   // | SPI_PUSHR_PCS(peripheralchipSelect)
      | SPI_PUSHR_TXDATA(0xFF);
  }
}

while (TRUE) // Send 16 bit
{
  // Check if Transmit FIFO is not full (TFFF==1)
  //if (SPI_SR_REG(SPI) & SPI_SR_TFFF_MASK && length) funktioniert nicht!!!!
  if (((SPI0->SR & SPI_SR_TXCTR_MASK) < SPI_SR_TXCTR(3)) && length)
  {
    length -= 2;
    if (txBuf)
    {
      uint16_t data = *txBuf++ << 8;
      data |= *txBuf++;
      SPI0->PUSHR = SPI_PUSHR_CONT(1)//length != 0)
        | SPI_PUSHR_CTAS(1)
        | SPI_PUSHR_EOQ(length == 0)
        | SPI_PUSHR_CTCNT(0)  // Clear the TCNT field
      //| SPI_PUSHR_PCS(peripheralchipSelect)
        | SPI_PUSHR_TXDATA(data);
    }
    else
    {
      SPI0->PUSHR = SPI_PUSHR_CONT(1)//length != 0)
       | SPI_PUSHR_CTAS(1)
       | SPI_PUSHR_EOQ(length == 0)
       | SPI_PUSHR_CTCNT(0)  // Clear the TCNT field
     //| SPI_PUSHR_PCS(peripheralchipSelect)
       | SPI_PUSHR_TXDATA(0xFFFF);
    }
  }

  // Check if RX FIFO is not empty (RFDF == 1)
  //if (SPI_SR_REG(SPI) & SPI_SR_RFDF_MASK)
  if (SPI0->SR & SPI_SR_RXCTR_MASK)
  {
    uint16_t tmp = SPI0->POPR;
    if (has8Bit)
    {
      has8Bit = FALSE;
      rxCount--;
      if (rxBuf) *rxBuf++ = (uint8_t)tmp;
```

```
      }
      else
      {
        rxCount -= 2;
        if (rxBuf)
        {
          *rxBuf++ = (uint8_t)(tmp >> 8);
          *rxBuf++ = (uint8_t)(tmp);
        }
      }
      if (rxCount == 0)
      {
        while (SPI0->SR & SPI_SR_TXRXS_MASK){}
        break;
      }
    }
  }
}
```

</> Peripheral Chip Select

```
void spiSetPcsPin(tSpiPcs pcs)
{
  peripheralchipSelect = pcs;
}
```

</> Initialisieren

```
void spiInit(void)
{
  SIM->SCGC6 |= SIM_SCGC6_SPI0(1);

  PORTC->PCR[6] = PORT_PCR_MUX(2); // MOSI on PTC6, ALT2
  PORTC->PCR[7] = PORT_PCR_MUX(2) | PORT_PCR_PS(1) | PORT_PCR_PE(1);   // MISO on PTC7, ALT2
  PORTC->PCR[5] = PORT_PCR_MUX(2);   //CLK on PTC5, ALT2
  PORTC->PCR[1] = PORT_PCR_MUX(2); // PCS3 on PTC1, ALT2

  SPI0->MCR =
      SPI_MCR_MSTR(1)          // Master Mode
      | SPI_MCR_ROOE(0)        // On Rx FIFO Overflow: Incomming data is shifted into
                               // the register
      | SPI_MCR_PCSIS(0x3F)    // The inactive state of PCSx is high
      | SPI_MCR_DIS_TXF(0)     // Enable Transmit FIFO
      | SPI_MCR_DIS_RXF(0)     // Enable Receive FIFO
      | SPI_MCR_CLR_TXF(1)     // Flush Transmit FIFO
      | SPI_MCR_CLR_RXF(1)     // Flush Receive FIFO
      | SPI_MCR_HALT(1);       // Halt Module during init!

  SPI0->MCR &= ~SPI_MCR_HALT(1); // Disable Halt

  spiSetSpeed(TRUE, 1, 0, 0, 0); // 20 MHz
}
```