Mikroprozessorsysteme

Prof. Dr.-Ing. Irenäus Schoppa

HTWG Konstanz

Weiterführende Literatur

- Digilent: *Digilent Pmod™* Interface Specification, 2011.
- M. Jiménez, R. Palomera und I. Couvertier: *Introduction to Embedded Systems: Using Microcontrollers and the MSP430*, Springer Verlag, 2014.
- C. Nagy: *Embedded Systems Design Using the TI MSP430 Series*, Elsevier Verlag, 2013.
- J. H. Davies: *MSP430 Microcontroller Basics*, Newnes, 2008.

- M. Schaefer, A. Gnedina und weitere: *Programmierregeln für die Erstellung von Software für Steuerungen mit Sicherheitsaufgaben*, Bundesanstalt für Arbeitsschutz und Arbeitsmedizin, 1998.

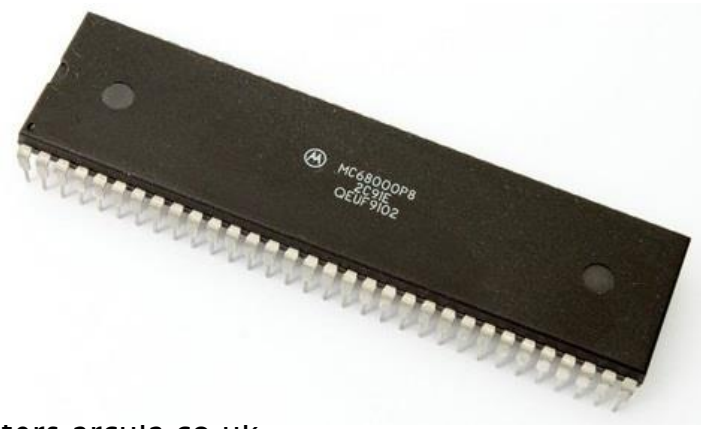
Weiterführende Literatur

- TI: *MSP430FR572x Mixed-Signal Microcontrollers*, Datenblatt, Texas Instrumenst, 2016, (msp430fr5729.pdf).
- TI: *MSP430FR57xx Family*, User's Guide, Texas Instruments, 2013, (SLAU272C.pdf).
- L. Bierl: *MSP430 Family Mixed-Signal Microcontroller Application Reports*, Texas Instruments, 2000, (SLAA024.pdf).
- TI: MSP430 *Optimizing C/C++ Compiler*, User's Guide, Texas Instruments, 2021, (SLAU132Y.pdf).
- K. Quiring: *Software Coding Techniques for MSP430™ MCUs*, Texas Instruments, 2018, (SLAA294B.pdf)
- AMS: *AS1108 4-Digit LED Display Driver*, 2012, (AS1108_Datasheet_EN_v2.pdf).

Prozessor vs. Mikroprozessor

- Prozessor:
  16-Bit-CPU PDP-11
- Hersteller: DEC
- Jahrgang: 1970

- Mikroprozessor:
  16-Bit CPU MC68000
- Hersteller: Motorola
- Jahrgang: 1979



Quellen: http://www.oldcomputers.arcuia.co.uk
http://cbmmuseum.kuto.de/cpu_mc68000.html

- Digilent Pmod™ Interface Specification

Pmod Type 1 (GPIO)

Pin	Signal	Direction
1	IO1	In/Out
2	IO2	In/Out
3	IO3	In/Out
4	IO4	In/Out
5	GND	
6	VCC	(+3,3V)

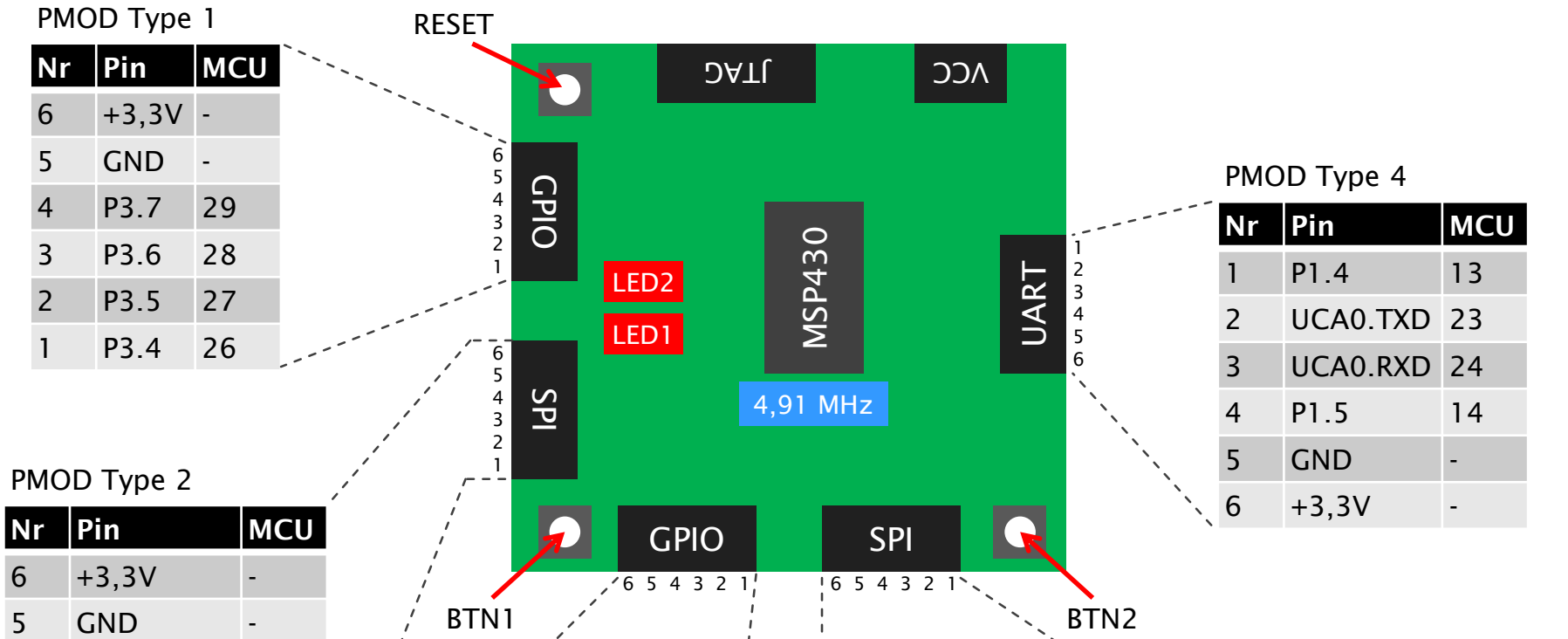
Pmod Type 2 (SPI)

Pin	Signal	Direction
1	SS	Out
2	MOSI	Out
3	MISO	In
4	SCK	Out
5	GND	
6	VCC	(+3,3V)

Pmod Type 4 (UART)

Pin	Signal	Direction
1	CTS	In
2	TXD	Out
3	RXD	In
4	RTS	Out
5	GND	
6	VCC	(+3,3V)

Mainboard: Überblick



PMOD Type 1

Nr	Pin	MCU
6	+3,3V	-
5	GND	-
4	P3.7	29
3	P3.6	28
2	P3.5	27
1	P3.4	26

PMOD Type 4

Nr	Pin	MCU
1	P1.4	13
2	UCA0.TXD	23
3	UCA0.RXD	24
4	P1.5	14
5	GND	-
6	+3,3V	-

PMOD Type 2

Nr	Pin	MCU
6	+3,3V	-
5	GND	-
4	UCB0.CLK	25
3	UCB0.SOMI	31
2	UCB0.SIMO	30
1	UCB0.STE	12

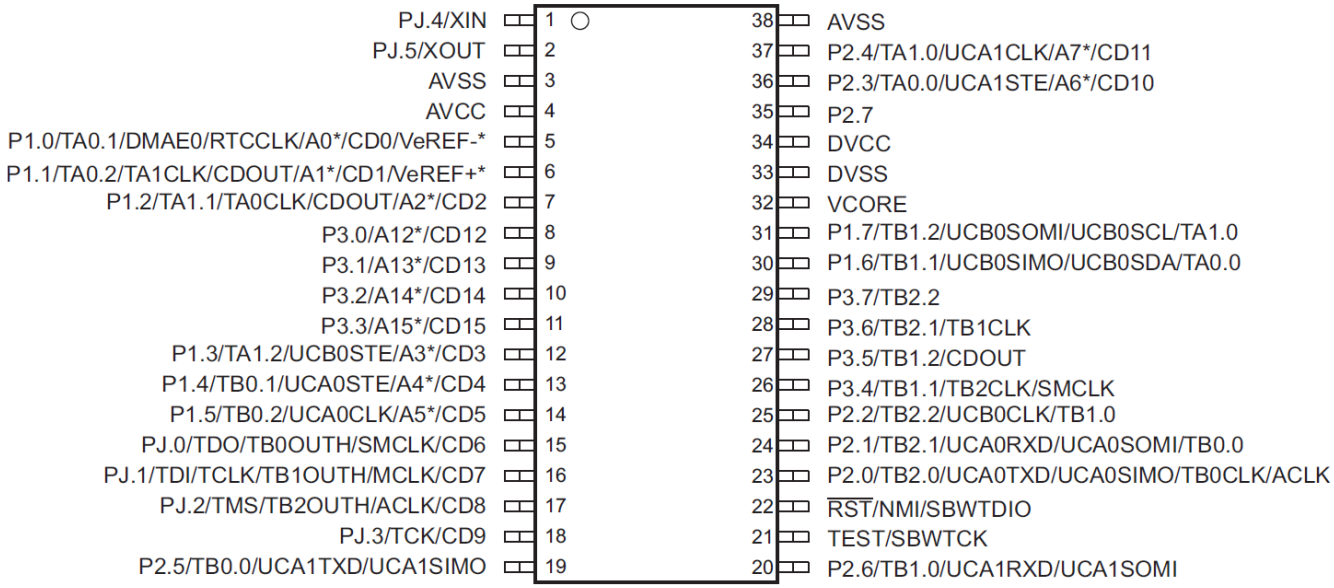
PMOD Type 1

Nr	Pin	MCU
6	+3,3V	-
5	GND	-
4	P3.3	11
3	P3.2	10
2	P3.1	9
1	P3.0	8

PMOD Type 2

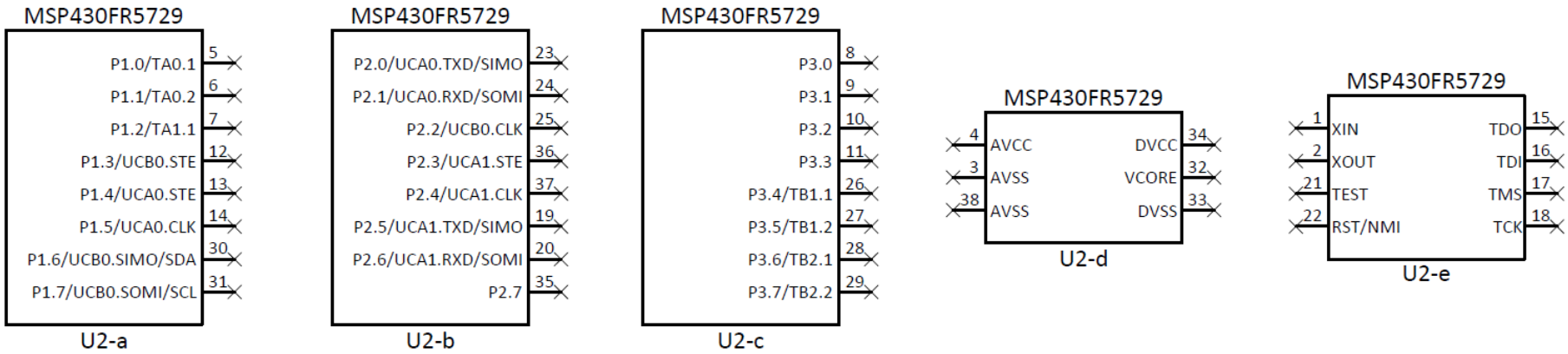
Nr	Pin	MCU
6	+3,3V	-
5	GND	-
4	UCA1.CLK	37
3	UCA1.SOMI	20
2	UCA1.SIMO	19
1	UCA1.STE	36

	Pin	MCU
Reset	RST	22
LED1	P2.7	35
LED2	P1.2	7
BTN1	P1.1	6
BTN2	P1.0	5

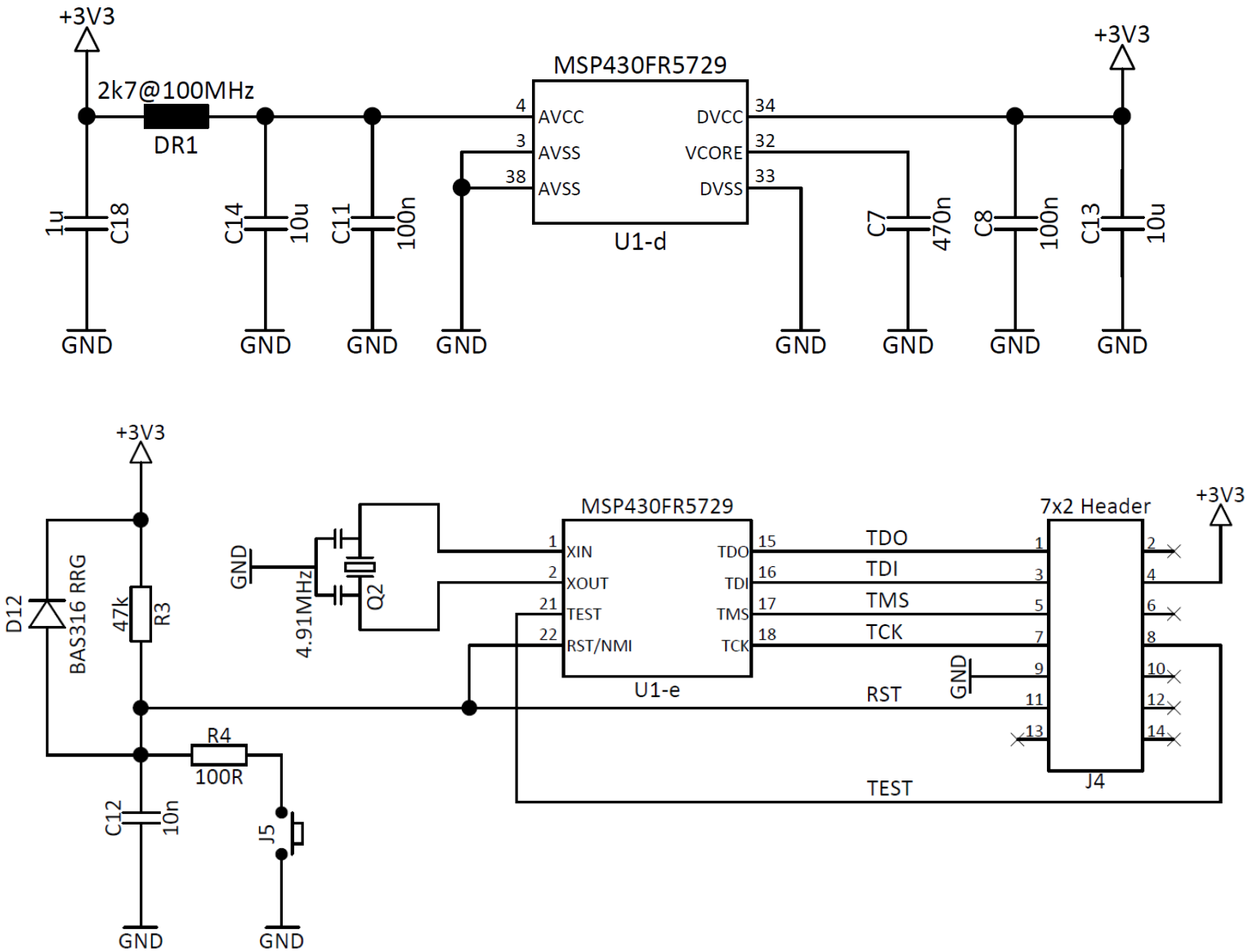


* Not available on MSP430FR5727, MSP430FR5723

Figure 4-2. 38-Pin DA Package (Top View)



Mainboard: Power & Service



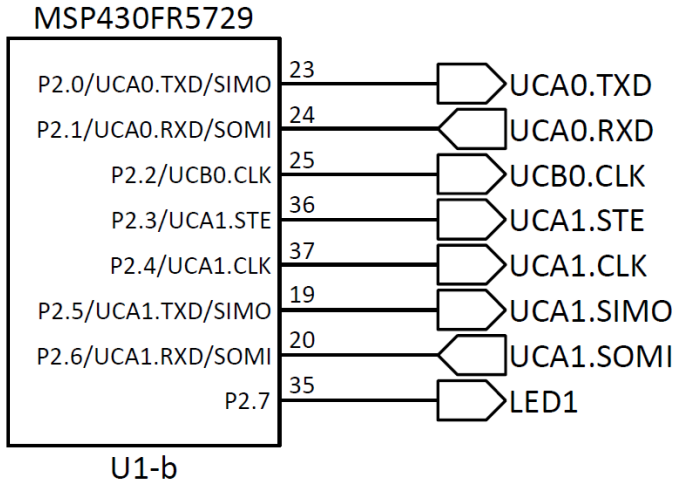
4.5 Signal Descriptions

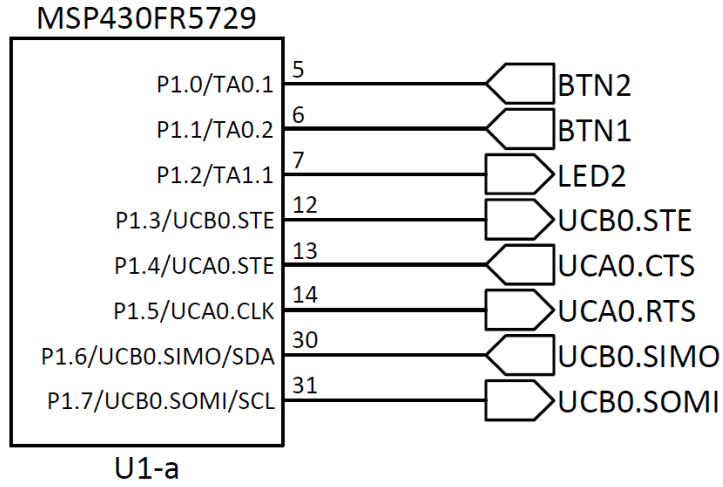
Table 4-1 describes the signals for all device variants and packages.

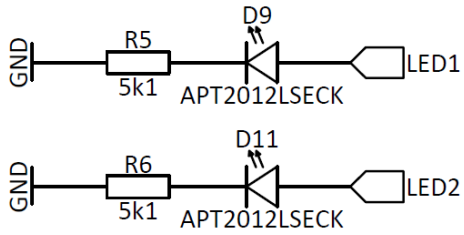
Table 4-1. Signal Descriptions

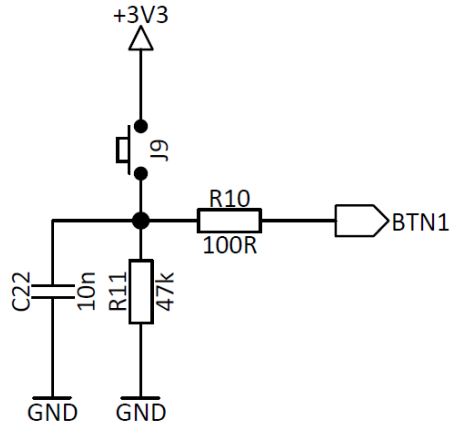
TERMINAL					I/O (1)	DESCRIPTION
NAME	NO.					
	RHA	RGE	DA	PW		
P1.0/TA0.1/DMAE0/ RTCCLK/A0/CD0/VeREF-	1	1	5	5	I/O	General-purpose digital I/O with port interrupt and wake up from LPMx.5 TA0 CCR1 capture: CCI1A input, compare: Out1 External DMA trigger RTC clock calibration output Analog input A0 – ADC (not available on devices without ADC) Comparator_D input CD0 External applied reference voltage (not available on devices without ADC)
P1.1/TA0.2/TA1CLK/ CDOUT/A1/CD1/VeREF+	2	2	6	6	I/O	General-purpose digital I/O with port interrupt and wake up from LPMx.5 TA0 CCR2 capture: CCI2A input, compare: Out2 TA1 input clock Comparator_D output Analog input A1 – ADC (not available on devices without ADC) Comparator_D input CD1 Input for an external reference voltage to the ADC (not available on devices without ADC)

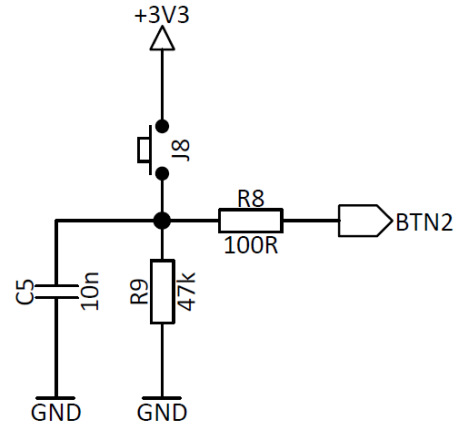
Mainboard: Taster und LEDs







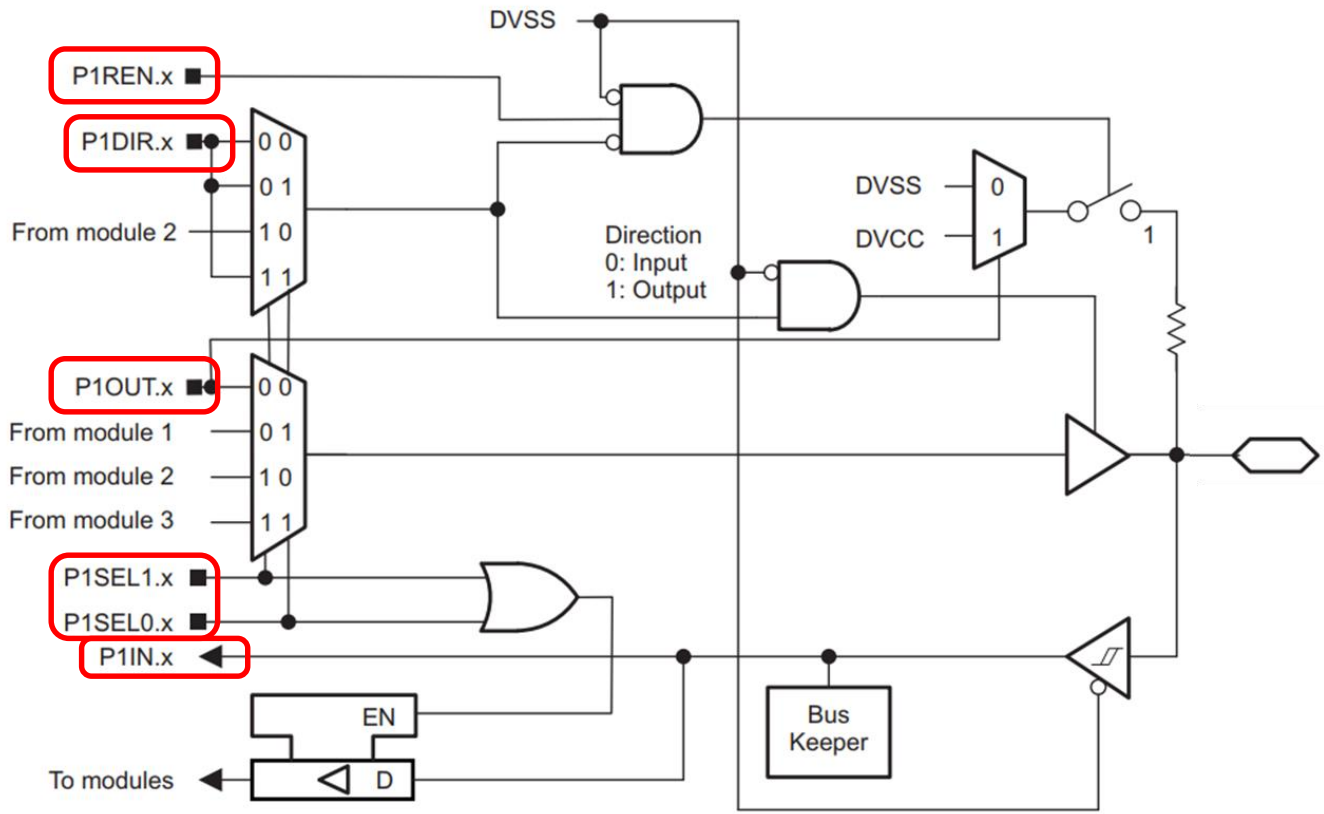




MSP430: Register und IO-Ports

	Registerbeschreibung
PxIN	Input
PxOUT	Output
PxDIR	Direction
PxREN	Pullup/Pulldown Enable
PxSEL0	Selection 0
PxSEL1	Selection 1
PxIV	Interrupt Vector
PxSELC	Complement Selection
PxES	Interrupt Edge Select
PxIE	Interrupt Enable
PxIFG	Interrupt Flag

x ist 1, 2, 3, oder 4



MSP430: Pin 7 im Port P2 als GPIO

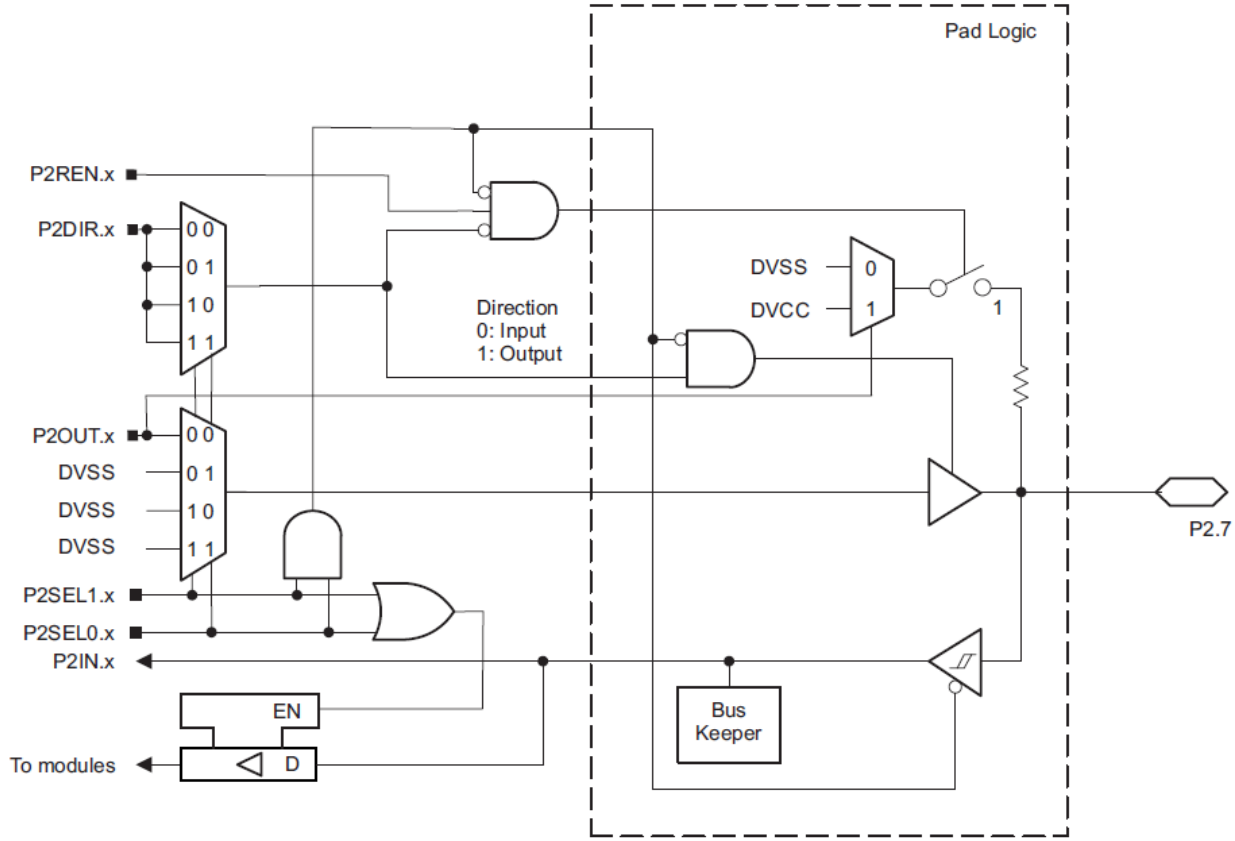


Figure 6-15. Port P2 (P2.7) Diagram

Table 8-1. I/O Configuration

PxDIR	PxREN	PxOUT	I/O Configuration
0	0	x	Input
0	1	0	Input with pulldown resistor
0	1	1	Input with pullup resistor
1	x	x	Output

Table 8-2. I/O Function Selection

PxSEL1	PxSEL0	I/O Function
0	0	General purpose I/O is selected
0	1	Primary module function is selected
1	0	Secondary module function is selected
1	1	Tertiary module function is selected

MSP430: Steuerbits im Port P2

Registerbeschreibung	Registername	Offset
Port P2 input	P2IN	01h
Port P2 output	P2OUT	03h
Port P2 direction	P2DIR	05h
Port P2 pullup/pulldown enable	P2REN	07h
Port P2 selection 0	P2SEL0	0Bh
Port P2 selection 1	P2SEL1	0Dh
Port P2 complement selection	P2SELC	17h
Port P2 interrupt vector word	P2IV	1Eh
Port P2 interrupt edge select	P2IES	19h
Port P2 interrupt enable	P2IE	1Bh
Port P2 interrupt flag	P2IFG	1Dh

Datentypen im Compiler

Table 5-1. MSP430 C/C++ Data Types

Type	Size	Alignment	Representation	Range	
				Minimum	Maximum
signed char	8 bits	8	Binary	-128	127
char	8 bits	8	ASCII	0 or -128 (1)	255 or 127 (1)
unsigned char	8 bits	8	Binary	0	255
bool (C99)	8 bits	8	Binary	0 (false)	1 (true)
_Bool (C99)	8 bits	8	Binary	0 (false)	1 (true)
bool (C++)	8 bits	8	Binary	0 (false)	1 (true)
short, signed short	16 bits	16	2s complement	-32 768	32 767
unsigned short	16 bits	16	Binary	0	65 535
int, signed int	16 bits	16	2s complement	-32 768	32 767
unsigned int	16 bits	16	Binary	0	65 535
long, signed long	32 bits	16	2s complement	-2 147 483 648	2 147 483 647
unsigned long	32 bits	16	Binary	0	4 294 967 295
long long, signed long long	64 bits	16	2s complement	-9 223 372 036 854 775 808	9 223 372 036 854 775 807
unsigned long long	64 bits	16	Binary	0	18 446 744 073 709 551 615
enum	varies (2)	16	2s complement	varies	varies
float	32 bits	16	IEEE 32-bit	1.175 494e-38 (3)	3.40 282 346e+38
double	64 bits	16	IEEE 64-bit	2.22 507 385e-308 (3)	1.79 769 313e+308
long double	64 bits	16	IEEE 64-bit	2.22 507 385e-308 (3)	1.79 769 313e+308
function and data pointers	varies (see Table 5-2)	16			

(1) "Plain" char has the same representation as either signed char or unsigned char. The --plain_char option specifies whether "plain" char is signed or unsigned. The default is unsigned.

(2) For details about the size of an enum type, see Section 5.6.1.

(3) Figures are minimum precision.

base.h

- Auszug aus der Datei base.h mit nützlichen Definitionen, Makros und Datentypen

```
#define EQ      ==
#define NE      !=
#define GE      >=
#define GT      >
#define LE      <=
#define LT      <
#define NOT     !
#define AND     &&
#define OR      ||
#define BAND    &
#define BOR     |
#define BNOT    ~
#define XOR     ^
```

```
#define SETBIT(p,b) ((p)|=(b))
#define CLRBIT(p,b) ((p)&= ~(b))
#define TSTBIT(p,b) ((p)&(b))
#define TGLBIT(p,b) ((p)^=(b))
```

```
#define Void    void
#define LOCAL   static
#define GLOBAL
#define EXTERN  extern

/* primitive types */
typedef int             Int;
typedef unsigned int    UInt;
typedef long            Long;
typedef unsigned long   ULong;
typedef short           Short;
typedef unsigned short  UShort;
typedef char            Char;
typedef char *          String;
typedef unsigned char   UChar;
typedef double          Double;
typedef float           Float;
```

Ansteuerung der LED1 an P2.7

Beispiel 1

```
#include <msp430.h> /* --> msp430fr5729.h */
#include "..\base.h"

// die LED am Pin 7 vom Port 2 soll mit
// einer Frequenz von ca. 1.0 Hz blinken
// => High dauert 0.5 Sek, Low dauert 0.5 Sek.

// die notwendige Verzögerung wird meistens
// experimentell bestimmt
#define DLY 480000

//#define NOBITMACROS

LOCAL Void exec(Void);

/**
 * main.c
 */
GLOBAL Void main(Void) {

   // stop watchdog timer
   WDTCTL = WDTPW + WDTHOLD;

   exec();

}
```

```
LOCAL Void exec(Void) {

#ifdef NOBITMACROS
   P2OUT  &= ~BIT7;
   P2DIR  |=  BIT7;
   P2SEL0 &= ~BIT7;
   P2SEL1 &= ~BIT7;
   P2REN  &= ~BIT7;

   while(TRUE) {
      __delay_cycles(DLY);
      P2OUT |= BIT7;
      __delay_cycles(DLY);
      P2OUT &= ~BIT7;
   }
```

ohne Bit-Makros

```
#else
   CLRBIT(P2OUT,  BIT7);
   SETBIT(P2DIR,  BIT7);
   CLRBIT(P2SEL0, BIT7);
   CLRBIT(P2SEL1, BIT7);
   CLRBIT(P2REN,  BIT7);

   while(TRUE) {
      __delay_cycles(DLY);
      TGLBIT(P2OUT, BIT7);
   }
```

mit Bit-Makros

```
#endif

}
```

Hinweise zum Programmierstil

- Initialisierung/Konfiguration von IO-Ports:
  - bitweise
    - einzelne Bits eines IO-Registers lassen sich mit Bitoperationen setzen oder löschen, z.B. mit Makros SETBIT() oder CLRBIT().
  - byteweise
    - alle acht Bitpositionen eines IO-Registers werden mit einem Befehl und einem 8-Bit-Operanden aktualisiert.
  - wortweise
    - zwei benachbarte IO-Register bilden ein 16-Bit-Register und lassen sich somit mit einem Befehl und einem 16-Bit-Operanden aktualisieren.
- Empfehlung: Die Initialisierung/Konfiguration aller IO-Ports soll in einer Datei und an einer zentralen Stelle in der Applikation wortweise vorgenommen werden. Dadurch lässt sich die Anzahl der Maschinenbefehle und somit die Programmgröße reduzieren.

Hinweise zum Programmierstil

- byteweise Initialisierung von Registern

Beispiel 2

```
// Port 2: Pin 7 => output, LED1

//     Bit 76543210
P1OUT  = 0b00000000; // clear all outputs
P1DIR  = 0b00000000; // direction, set outputs
P1IFG  = 0b00000000; // clear all interrupt flags
P1IE   = 0b00000000; // disable all GPIO interrupts
P1SEL0 = 0b00000000;
P1SEL1 = 0b00000000;
P1REN  = 0b00000000; // without pull up

//     Bit 76543210
P2OUT  = 0b00000000; // clear all outputs
P2DIR  = 0b10000000; // direction, set outputs
P2IFG  = 0b00000000; // clear all interrupt flags
P2IE   = 0b00000000; // disable all GPIO interrupts
P2SEL0 = 0b00000000;
P2SEL1 = 0b00000000;
P2REN  = 0b00000000; // without pull up
```

```
//     Bit 76543210
P3OUT  = 0b00000000; // clear all outputs
P3DIR  = 0b00000000; // direction, set outputs
P3IFG  = 0b00000000; // clear all interrupt flags
P3IE   = 0b00000000; // disable all GPIO interrupts
P3SEL0 = 0b00000000;
P3SEL1 = 0b00000000;
P3REN  = 0b00000000; // without pull up

//     Bit 76543210
P4OUT  = 0b00000000; // clear all outputs
P4DIR  = 0b00000000; // direction, set outputs
P4IFG  = 0b00000000; // clear all interrupt flags
P4IE   = 0b00000000; // disable all GPIO interrupts
P4SEL0 = 0b00000000;
P4SEL1 = 0b00000000;
P4REN  = 0b00000000; // without pull up
```

Hinweise zum Programmierstil

Beispiel 3

- wortweise Initialisierung von Registerpaaren

```
#define VAL_16BIT(arg1, arg2) ((unsigned)(((arg1) << 8) | (arg2)))

// Port 1: Pin 0 => input, BTN2
// Port 1: Pin 1 => input, BTN1
// Port 1: Pin 2 => output, LED2
// Port 2: Pin 7 => output, LED1

//                   Port2       Port1
//                Bit 76543210    76543210
PAOUT  = VAL_16BIT(0b00000000, 0b00000000); // clear all outputs
PADIR  = VAL_16BIT(0b10000000, 0b00000100); // direction, set outputs
PAIFG  = VAL_16BIT(0b00000000, 0b00000000); // clear all interrupt flags
PAIE   = VAL_16BIT(0b00000000, 0b00000000); // disable all GPIO interrupts
PASEL0 = VAL_16BIT(0b00000000, 0b00000000);
PASEL1 = VAL_16BIT(0b00000000, 0b00000000);
PAREN  = VAL_16BIT(0b00000000, 0b00000000); // without pull up

//                   Port4       Port3
//                Bit 76543210    76543210
PBOUT  = VAL_16BIT(0b00000000, 0b00000000); // clear all outputs
PBDIR  = VAL_16BIT(0b00000000, 0b00010000); // direction, set outputs
PBIFG  = VAL_16BIT(0b00000000, 0b00000000); // clear all interrupt flags
PBIE   = VAL_16BIT(0b00000000, 0b00000000); // disable all GPIO interrupts
PBSEL0 = VAL_16BIT(0b00000000, 0b00000000);
PBSEL1 = VAL_16BIT(0b00000000, 0b00000000);
PBREN  = VAL_16BIT(0b00000000, 0b00000000); // without pull up
```

Power-On-/Reset

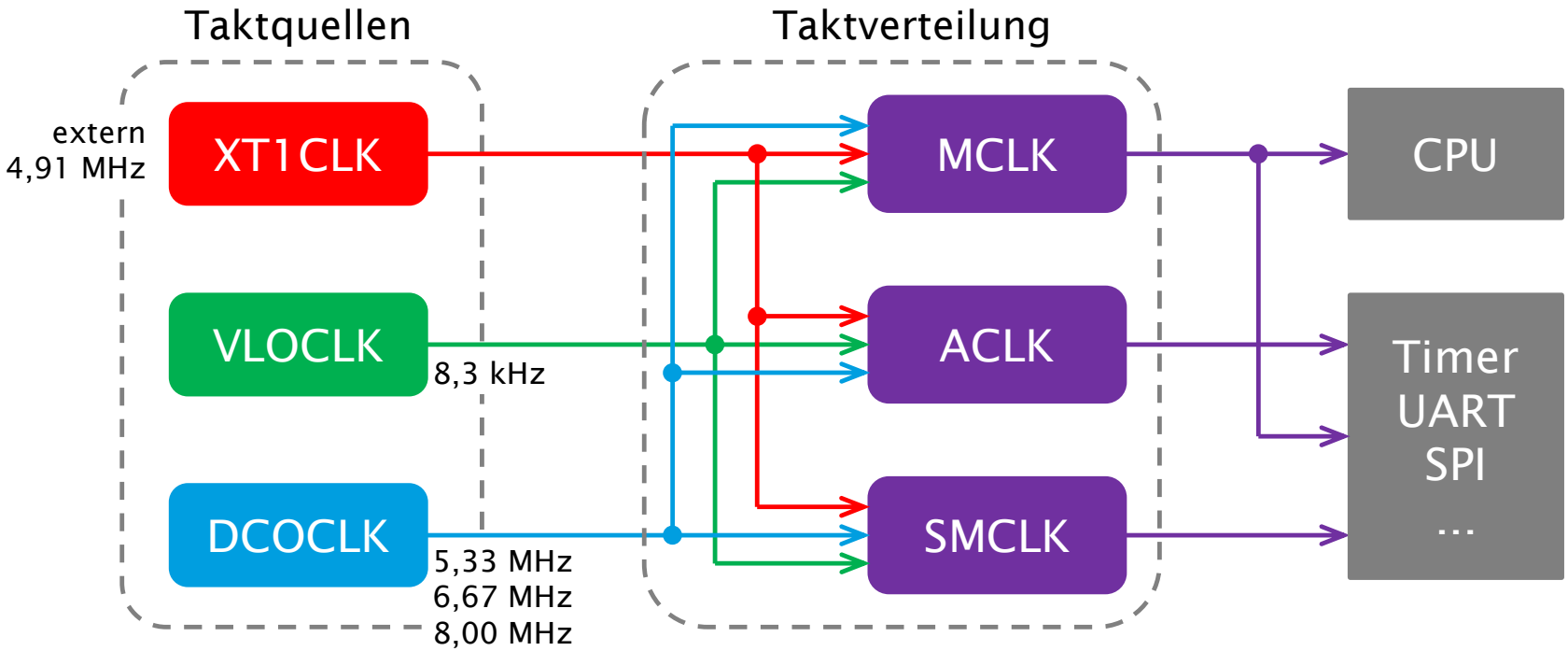
- Was passiert in einem MSP430 nach einem Power-On/Reset?
  - Eine C/C++-Applikation muss mit einer Bootstrap-Routine zusammen gelinkt werden. In der Bootstrap-Routine (Run-Time-Initialisierung) erfolgt die Initialisierung der Umgebung und der Start der Applikation.

```
void _c_int00 (. . .) {
```

1. Der Stackpointer SP wird initialisiert
2. Wenn erforderlich, wird die MPU (Memory Protection Unit) initialisiert.
3. Die Funktion `_system_pre_init()` wird aufgerufen.
   Diese Funktion kann in der Applikation überladen werden.
4. Wenn erforderlich wird die Funktion `_auto_init()` aufgerufen
5. Wenn erforderlich wird die Parameterliste (argc, argv) der Funktion `main()` ausgewertet.
6. Die Funktion `main()` wird aufgerufen.
   Ab hier fängt die Ausführung der eigentlichen Applikation an.

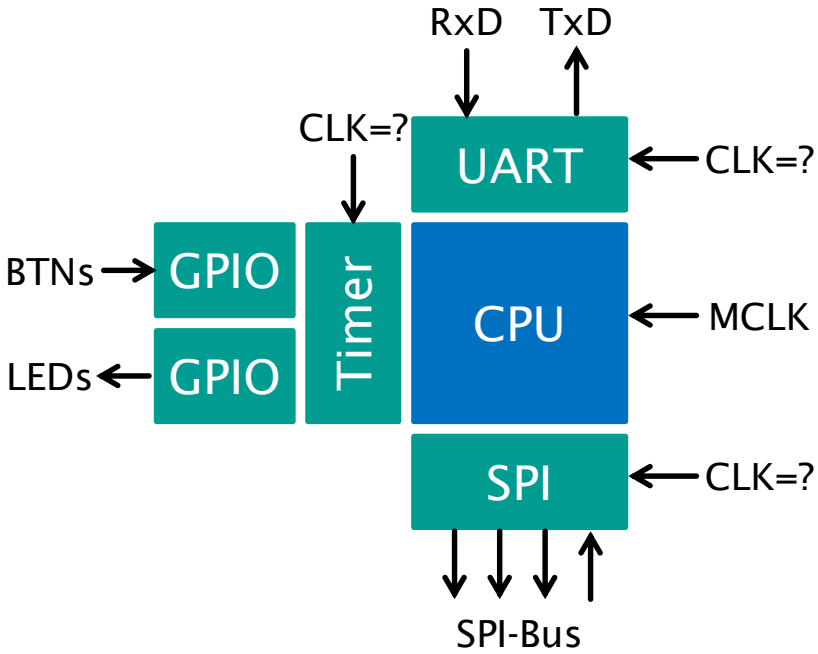
```
  exit(1);
}
```

Clock System



- Nach dem Reset ist das Clock System so konfiguriert:
  - DCOCLK mit 8,00 MHz
  - SMCLK und MCLK haben einen Taktteiler 8 (SMCLK und MCLK laufen also mit 1,0 MHz)

Clock System



CPU-Takt: default MCLK, 8,0 MHz
abgeleitet von DCO 8,0 MHz, Taktteiler =1
CPU aktiv nur beim Bedarf => Low Power Modus

UART-Takt wird von 4,9152 MHz abgeleitet,
UART-Receiver muss immer aktiv sein
=> UART bekommt ACLK mit XT1CLK

BTN-Abfrage im ca. 10 ms Intervall
BTN-Abfrage muss immer aktiv sein
=> BTN-Abfrage über Timer mit ACLK mit XT1CLK

SPI-Bus wird als Master beim Bedarf aktiviert,
Takt muss an SPI-Timing angepasst werden
=> SPI bekommt SMCLK mit DCO 8,0 MHz

LEDs sollen mit Blinkmustern verschiedene
(Fehler-)Zustände anzeigen
LEDs sind immer aktiv
=> LED-Ansteuerung mit Timer
und ACLK mit XT1CLK

Mode	CPU	MCLK	DCO	SMCLK	ACLK
AM	active	active	active	active	active
LPM0	disabled	disabled	active	active	active
LPM1	disabled	disabled	disabled	active	active
LPM2	disabled	disabled	disabled	disabled	active
LPM3	disabled	disabled	disabled	disabled	active
LPM4	disabled	disabled	disabled	disabled	disabled

- Das Mikrocontrollersystem wird im Low Power Mode LPM3 betrieben
  - die CPU und die meisten Peripherie-Komponenten befinden sich im Schlafmodus und werden beim Bedarf durch verschiedene Ereignisse geweckt.

Clock System

- Der Zugang zum Clock System ist über sieben Steuerregister CSCTL0, CSCTL1, ..., CSCTL6 möglich, und er ist password geschützt

- CSCTL0 (Das Password-Register)
  - Durch Schreiben von 0xA500 in das Steuerregister CSCTL0 wird der Zugang zum Clock System freigeschaltet.
  - Die Sperrung erfolgt durch Schreiben eines anderen, beliebigen Wertes (Bytes!) in das Steuerregister CSCTL0_H.

- CSCTL1 (DCO-Frequenzauswahl)
  - Mit dem Steuerbit DCORSEL wird der High-/Low-Speed-Generator aktiviert (für MSP430FR5729 nur Low-Speed)
  - Über zwei Steuerbits DCOFSEL wird die DCO-Frequenz Low-Speed: (5,33 MHz, 6,67 MHz oder 8,00 MHz) oder High-Speed (16 MHz, 20 MHz oder 24 MHz) selektiert.

Clock System

- CSCTL2
  - Auswahl der Signalquelle für MCLK, ACLK und SMCLK
  - Steuerbits SELM, SELA und SELS

000b = XT1CLK
001b = VLOCLK
010b = Reserved. Defaults to VLOCLK.
011b = DCOCLK
100b = Reserved. Defaults to DCOCLK.
101b = XT2CLK when available, otherwise DCOCLK
110b = Reserved. Defaults to XT2CLK when available, otherwise DCOCLK.
111b = Reserved. Defaults to XT2CLK when available, otherwise DCOCLK.

- CSCTL3
  - Auswahl des Frequenzteilers für MCLK, ACLK und SMCLK
  - Steuerbits DIVM, DIVA und DIVS

MCLK source divider. Divides the frequency of the MCLK clock source.
000b = f(MCLK)/1
001b = f(MCLK)/2
010b = f(MCLK)/4
011b = f(MCLK)/8
100b = f(MCLK)/16
101b = f(MCLK)/32
110b = Reserved. Defaults to f(MCLK)/32.
111b = Reserved. Defaults to f(MCLK)/32.

Clock System

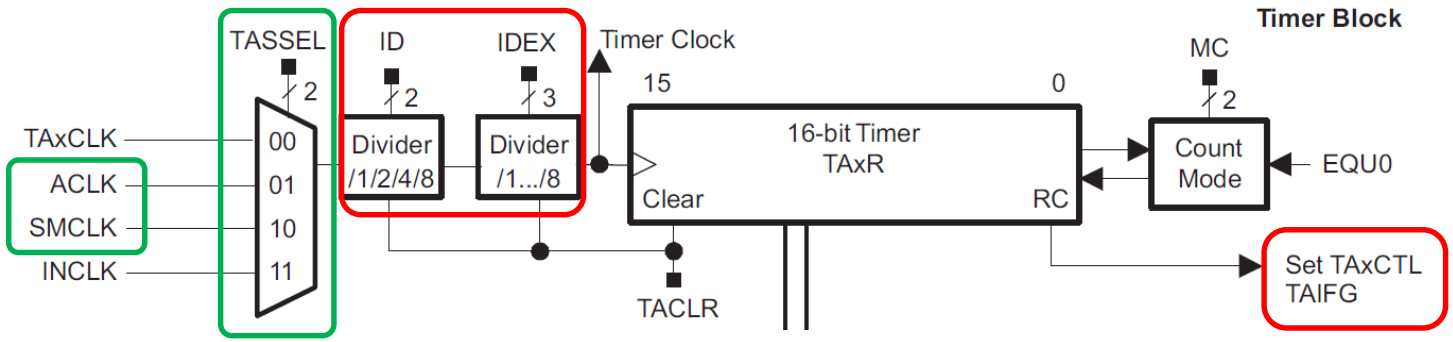
Beispiel 2

- Vorläufige Takteinstellung (ohne UART)
  - DCO-Takt mit 8,0 MHz
  - ACLK = 250,0 kHz: DCO-Quelle mit Teiler=32
  - SMCLK = 250,0 kHz: DCO-Quelle mit Teiler=32
  - MCLK = 8,00 MHz: DCO-Quelle mit Teiler=1

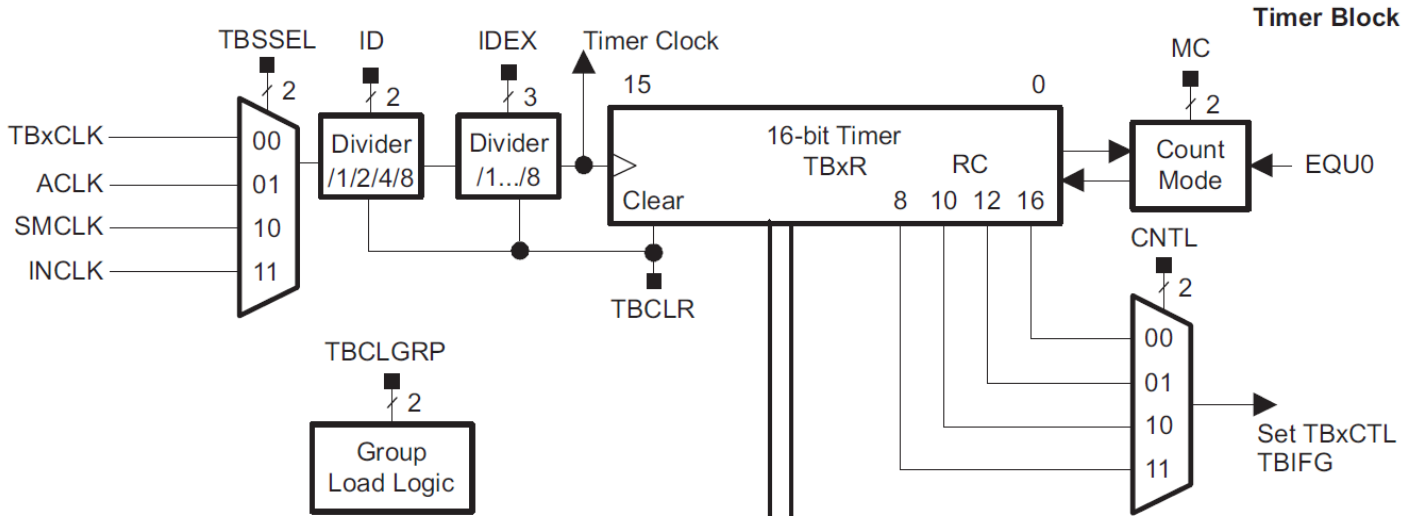
```
CSCTL0 = CSKEY;          // enable clock system

CSCTL1 = DCOFSEL_3;      // DCO frequency = 8.0 MHz

                         // select clock sources
CSCTL2 = SELA__DCOCLK    // ACLK  <- DCO
       | SELS__DCOCLK    // SMCLK <- DCO
       | SELM__DCOCLK;   // MCLK  <- DCO

                         // set frequency divider
CSCTL3 = DIVA__32        // ACLK  : /32 = 250.0 kHz
       | DIVS__32        // SMCLK : /32 = 250.0 kHz
       | DIVM__1;        // MCLK  : /1  =   8.0 MHz

CSCTL4 = XT1OFF          // XT1 disabled
       | XT2OFF;         // XT2 disabled

CSCTL0_H = 0;            // disable clock system
```

Timer: Struktur

- Timer_A: TA0 und TA1: 16-Bit-Zäher mit je 3 CCR



- Timer_B: TB0, TB1 und TB2: 8-/10-/12-/16-Bit-Register mit je 3 CCR



Timer: Arbeitsmodis

Table 11-1. Timer Modes

MC	Mode	Description
00	Stop	The timer is halted.
01	Up	The timer repeatedly counts from zero to the value of TAxCCR0
10	Continuous	The timer repeatedly counts from zero to 0FFFFh.
11	Up/down	The timer repeatedly counts from zero up to the value of TAxCCR0 and back down to zero.

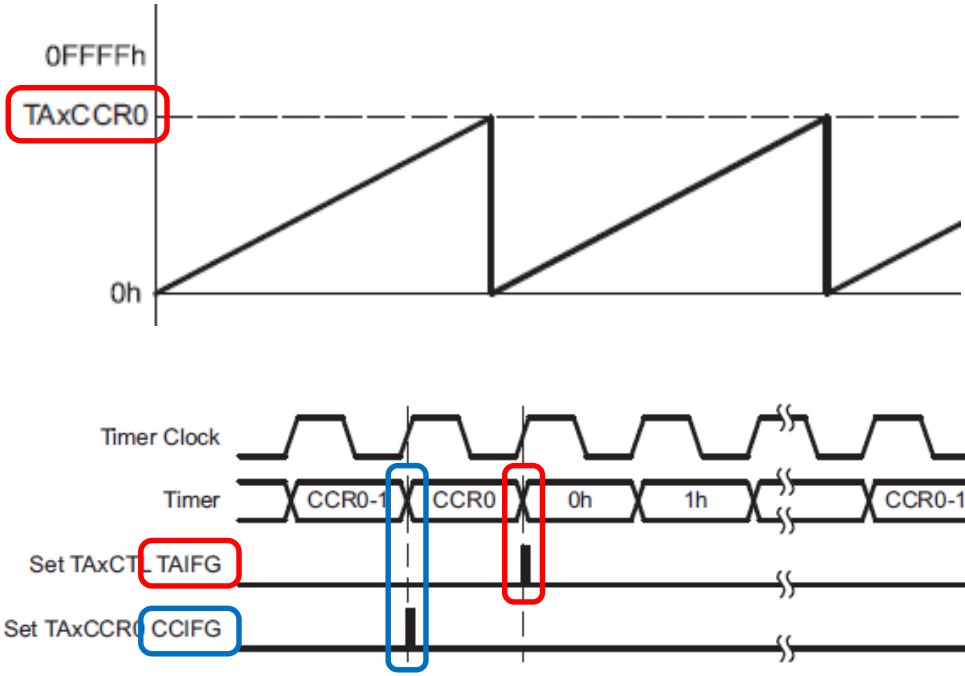
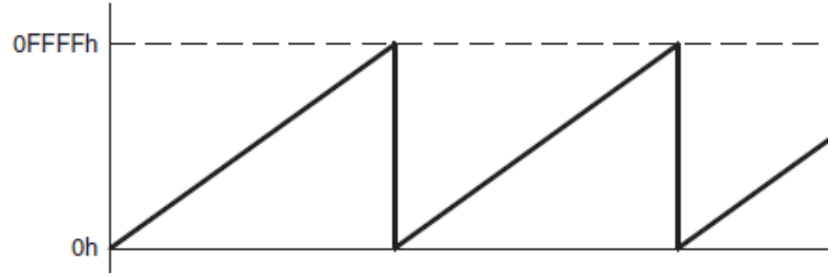
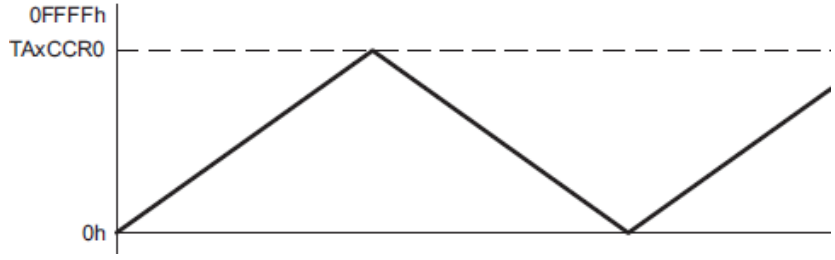
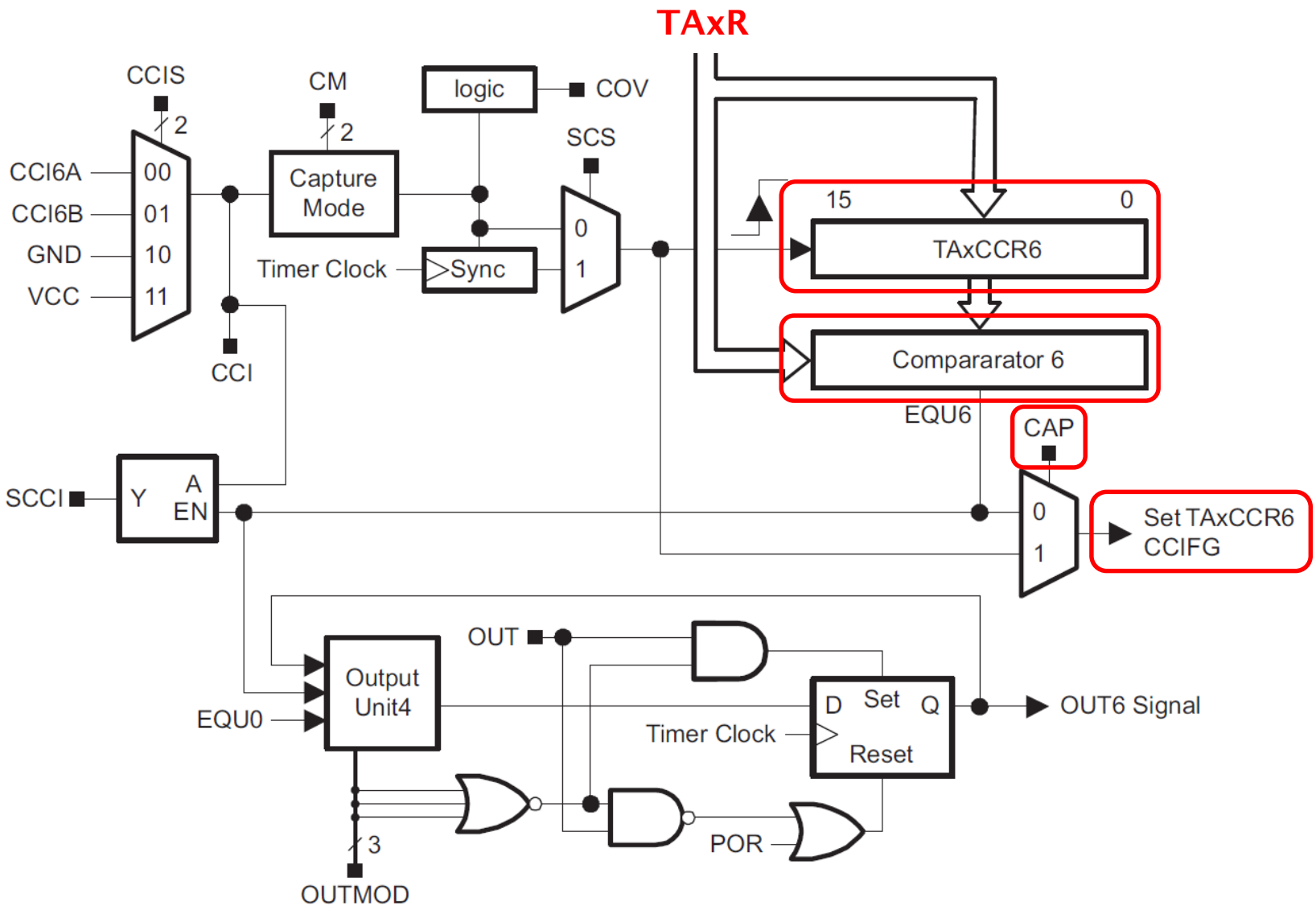


Figure 11-3. Up Mode Flag Setting





Timer: Capture/Compare Register

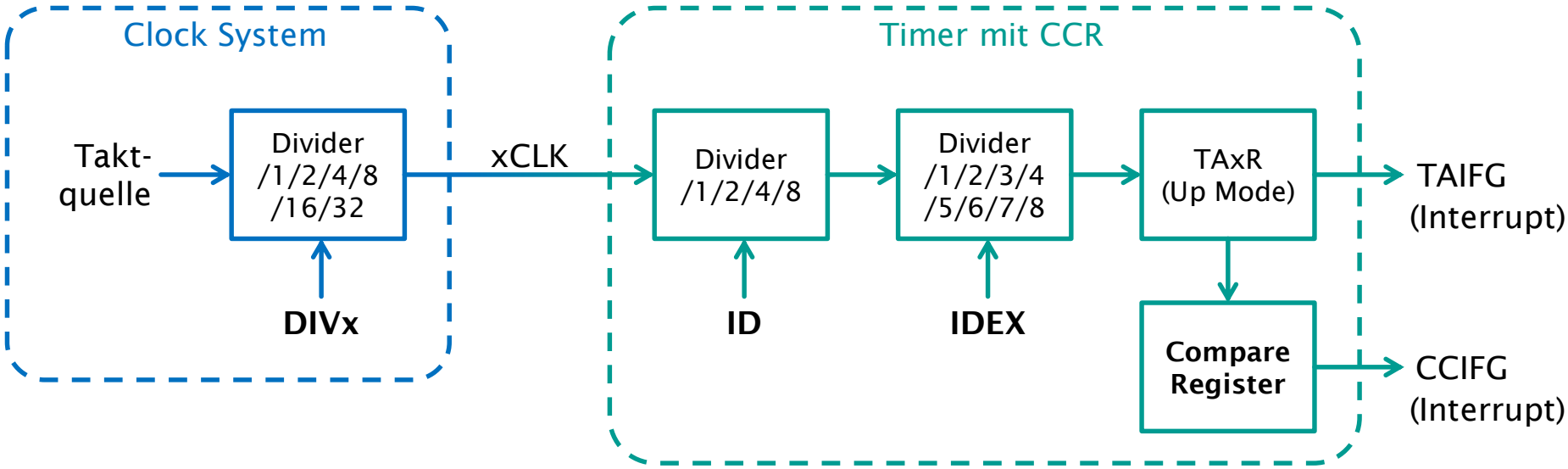


Timer: Register

TA0 Registers (Base Address: 0340h)

Registerbeschreibung	Registername	Offset	Access	Reset
TA0 control	TA0CTL	00h	Word	0000h
Capture/compare control 0	TA0CCTL0	02h	Word	0000h
Capture/compare control 1	TA0CCTL1	04h	Word	0000h
Capture/compare control 2	TA0CCTL2	06h	Word	0000h
TA0 counter	TA0R	10h	Word	0000h
Capture/compare 0	TA0CCR0	12h	Word	0000h
Capture/compare 1	TA0CCR1	14h	Word	0000h
Capture/compare 2	TA0CCR2	16h	Word	0000h
TA0 expansion 0	TA0EX0	20h	Word	0000h
TA0 interrupt vector	TA0IV	2Eh	Word	0000h

Timer: Einstellung



- Die Einstellung eines Timer erfolgt durch die Bestimmung drei Frequenzteiler (**DIVx**, **ID** und **IDEX**) und eines ganzzahligen Wertes für das Compare-Register

Timer: Einstellung

- Berechnungsschritte bei der Timer-Einstellung:
  - Zeitbasis festlegen (z.B. 250 us, 1 ms, 20 ms)
  - Teilungsfaktor berechnen: Produkt aus der Frequenz derjenigen Taktquelle (XT1CLK, DCOCLK, LVOCLK), mit der ein Timer getaktet wird, und der festgelegten Zeitbasis
  - Einstellungswert für das Compare-Register bestimmen: Teilungsfaktor mit Hilfe der Werte aus den Mengen

    DIVx in CSCTL3: {/1, /2, /4, /8, /16, /32}

    ID in TAxCTL: {/1, /2, /4, /8}

    IDEX in TAxEX0: {/1, /2, /3, /4, /5, /6, /7, /8}

    so zerlegen, dass der Einstellungswert für das Compare-Register TAxCCR0 die kleinstmögliche natürliche Zahl (also ohne Nachkommastellen!) ergibt

Timer: Einstellung

- Beispiel für die Timer-Einstellung TA0
  - LED soll mit 1,0 Hz blinken
  - Dauer der On-Phase = 500,0 ms
  - Dauer der Off-Phase = 500,0 ms
  - Taktquelle: DCOCLK mit 8,0 MHz
  - Wie müssen Frequenzteiler eingestellt werden?
  - Welcher Wert muss im Compare-Register gespeichert werden?

- Zeitbasis = 500,0 ms
- Teilungsfaktor: 8,0 MHz * 500 ms = $4*10^6$
- TA0CCR0 := $4*10^6$ {/32} {/8} {/5} = 3125
  - Clock: CSCTL3.DIVx = /32
  - Timer: TA0CTL.ID = /8
  - Timer: TA0EX0.IDEX = /5
  - Timer: TA0CCR0 = 3125-1

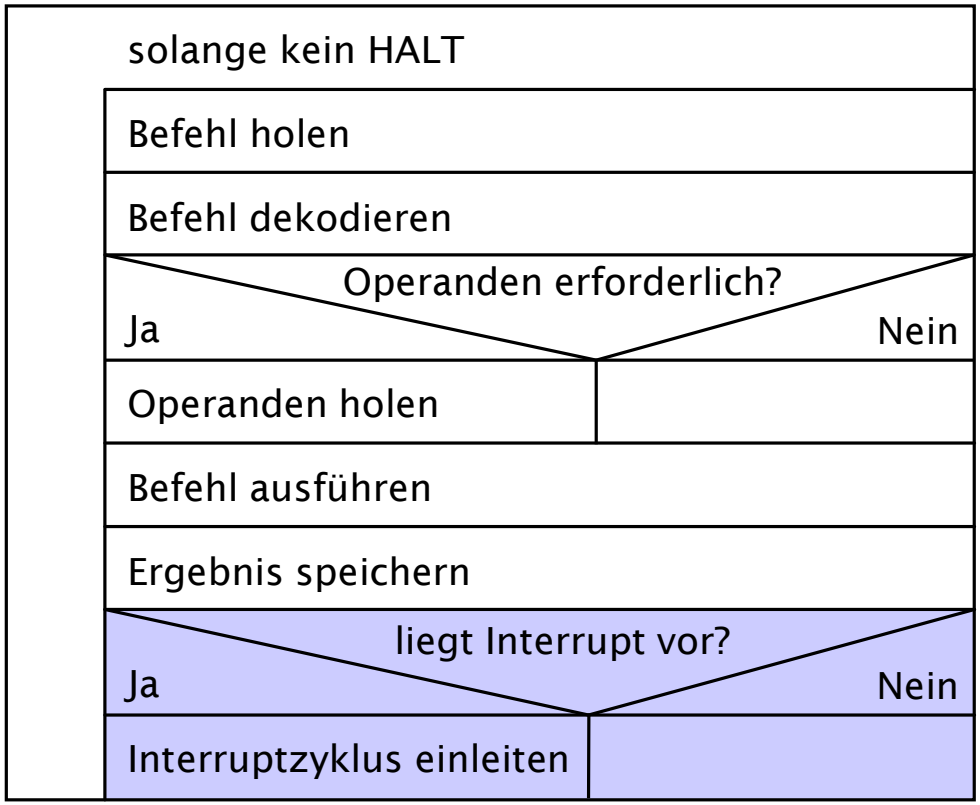
Timer: Einstellung

Beispiel 2

```
#pragma FUNC_ALWAYS_INLINE(Timer_init)
LOCAL Void Timer_init(Void) {
   CLRBIT(TA0CTL,   MC0 | MC1  // stop mode
                  | TAIE       // disable interrupt
                  | TAIFG);    // clear interrupt flag
   CLRBIT(TA0CCTL0, CM1 | CM0  // no capture mode
                  | CAP        // compare mode
                  | CCIE       // disable interrupt
                  | CCIFG);    // clear interrupt flag
   TA0CCR0  = 3125-1;          // set up Compare Register CCR0
   TA0EX0   = TAIDEX_4;        // set up expansion register /5
   TA0CTL   = TASSEL__ACLK     // 250.0 kHz
            | MC__UP           // Up Mode
            | ID__8            // /8
            | TACLR;           // clear and start Timer
}

...

while(TRUE) {
   while (TSTBIT(TA0CTL, TAIFG) EQ 0) ;  // wait on interrupt
   TGLBIT(P2OUT,  BIT7);                 // toggle led
   CLRBIT(TA0CTL, TAIFG);                // clear interrupt flag
}
```

Interrupts (Programmunterbrechungen)

- Prinzipieller Ablauf eines Befehlszyklus

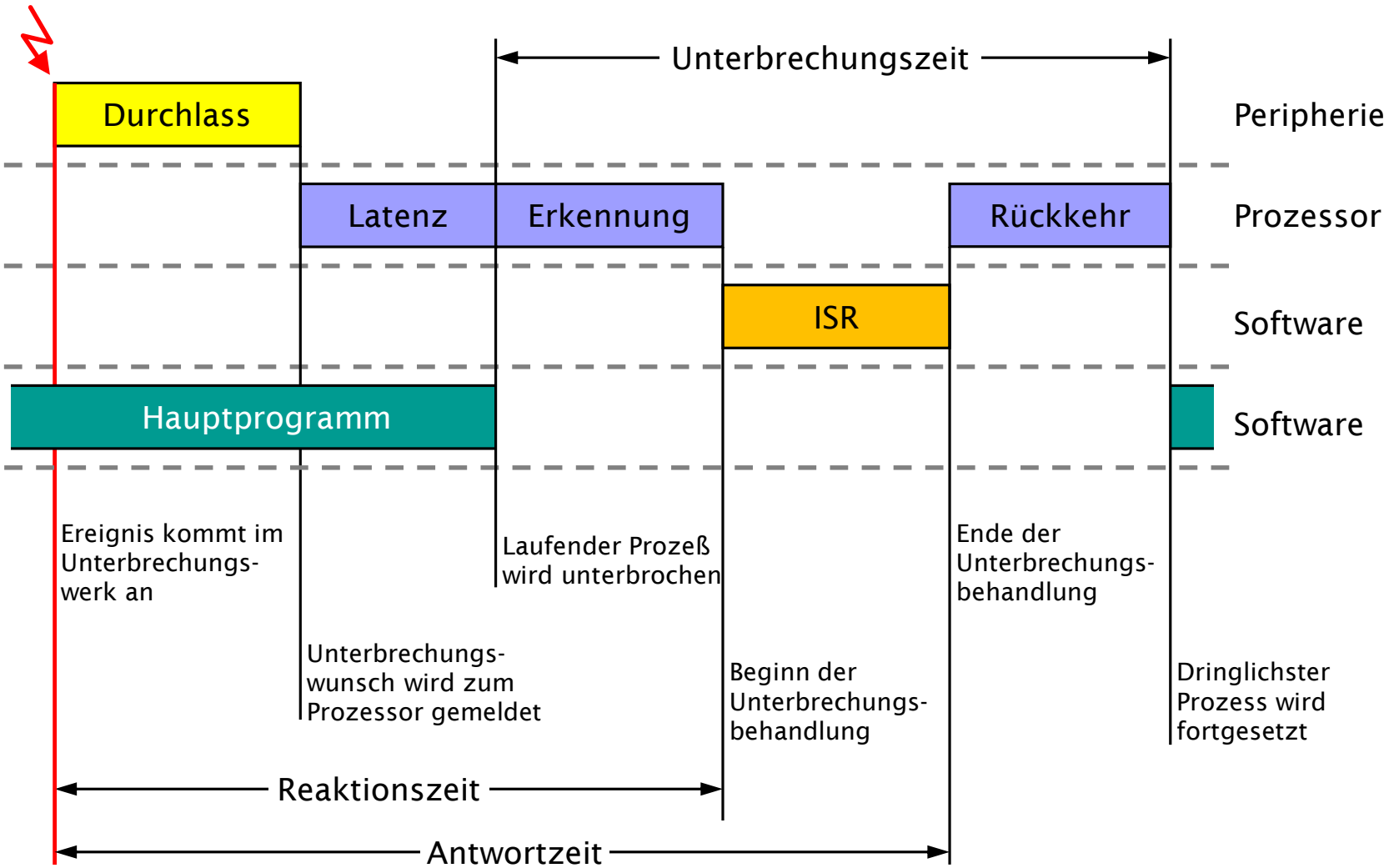


(Nassi-Shneiderman-Diagramm)

- Was passiert in einem Prozessor beim Auftreten eines Interrupts?
  1. Der momentan ausgeführte Maschinenbefehl wird bis zum Ende ausgeführt.
  2. Der PC, der auf den nächsten auszuführenden Befehl zeigt, wird auf dem Stack geschrieben.
  3. Das Statusregister wird ebenfalls auf den Stack geschrieben.
  4. Der Interrupt mit der momentanen höchsten Prioritär wird selektiert, falls mehrere Interrupts vorliegen.
  5. Die Adresse der ISR wird in den PC geladen.
  6. Der Prozessor beginnt mit der Ausführung des ersten Befehls in der ISR.

Interrupts (Programmunterbrechungen)

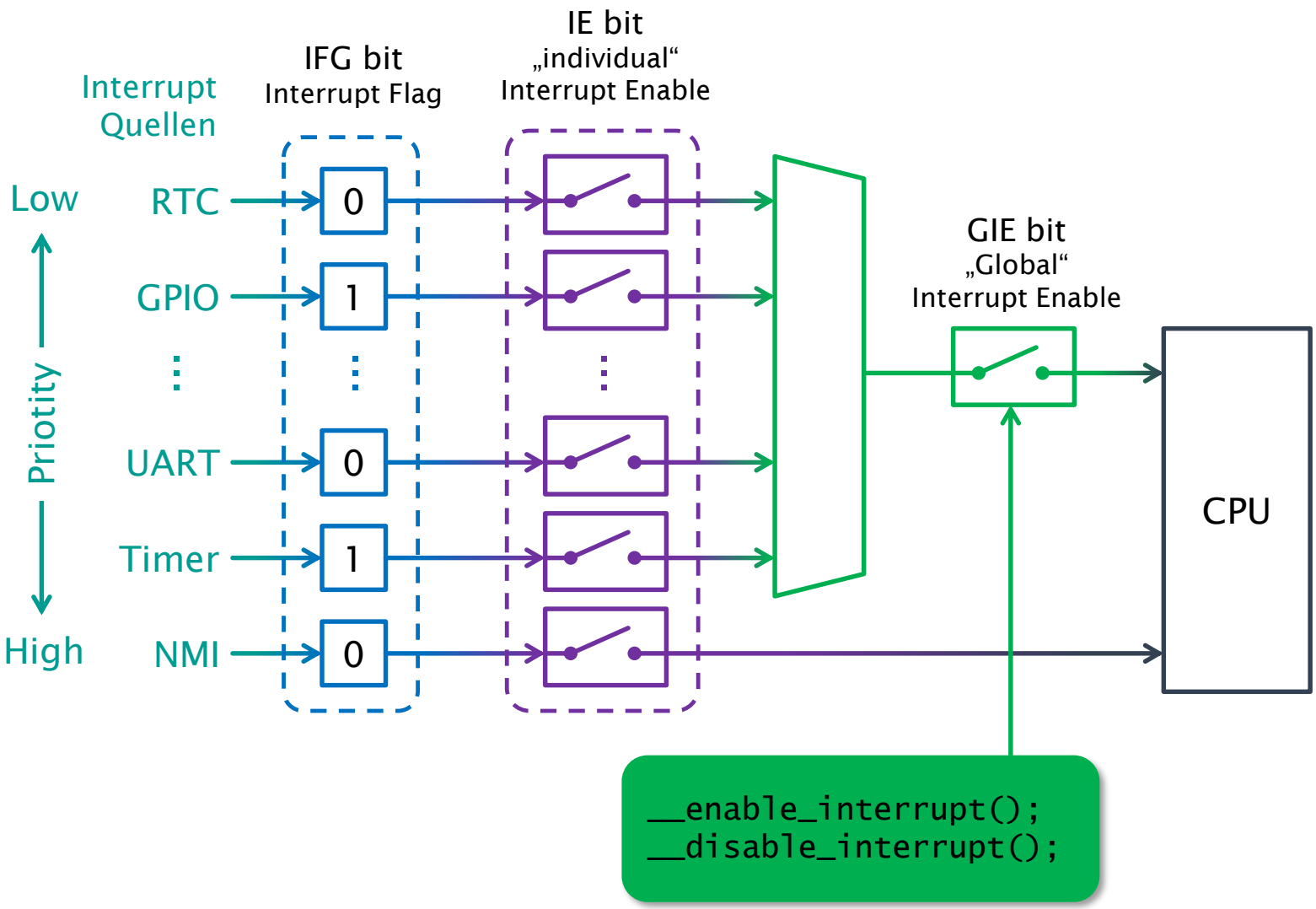
- zeitlicher Ablauf einer Programmunterbrechung



Interrupts (Programmunterbrechungen)

- DIN 66216 definiert den zeitlichen Ablauf einer Programmunterbrechung zwischen einer Peripherie-Einheit und einem Prozessor
  - Latenz: Zeit, die der Prozessor braucht, um in seinem Steuerwerk an eine unterbrechbare Stelle zu kommen
    - Abhängig vom Befehl und/oder Adressierungsarten
  - Erkennungszeit: Unterbrechungsquelle identifizieren, Zustand des Prozessors (PC, SR) retten, ggf. Interrupt-Acknowledge-Zyklus durchführen.
  - Reaktionszeit gib an, wie lange es nach dem Eintreten eines Ereignisses dauert, bis der Prozessor mit dessen Behandlung beginnt.
  - Antwortzeit bestimmt, wie lange es nach dem Eintreten eines Ereignisses dauert, bis es vollständig behandelt worden ist.
  - Unterbrechungszeit gibt an, wie lange ein laufendes Programm zugunsten einer ISR unterbrochen war.

Interrupts


IE bit
„individual“
Interrupt Enable
IFG bit
Interrupt Flag
Interrupt
Quellen
Low
RTC
GPIO
UART
Timer
NMI
High
Priotity
0
1
0
1
0
GIE bit
„Global“
Interrupt Enable
CPU
__enable_interrupt();
__disable_interrupt();

- Vordefinierte Macros für Interrupts

Makro	Bedeutung
__enable_interrupt()	Interrupts werden freigeschaltet: das globale Interrupt-Enable-Flag GIE wird aus 1 gesetzt.
__disable_interrupt()	Interrupts werden gespert: das globale Interrupt-Enable-Flag GIE wird auf 0 zurückgesetzt.
__low_power_mode_3()	Die CPU wird in den LPM3 versetzt, das globale Interrupt-Enable-Flag GIE wird aus 1 gesetzt.
__low_power_mode_off_on_exit()	Das Makro verhindet, dass die CPU beim Verlassen einer ISR in einen LPM zurückkehrt.
__even_in_range()	Das Makro ermöglicht es dem Compiler eine effiziente Code-Generierung in Switch-Statemens bei ISR => Tabelle mit Vektoren
__never_executed()	Das Makro ermöglicht es dem Compiler eine effiziente Code-Generierung in Switch-Statemens bei ISR.

Quelle: MSP430 Optimizing C/C++ Compiler, User's Guide
(über Help – Code Composer Studio)

Hinweise zum Programmierstil


```
char flag = 0;

#pragma vector = XXXXXXX_VECTOR
__interrupt void XXXXXXX_ISR(void) {
   flag = 1;
   // ggf. das Interrupt Flag löschen
   ...
   __low_power_mode_off_on_exit();
}
```

- ISR wird mit dem Wort __interrupt eingeleitet.
- ISR hat keine Parameter und keinen Rückgabewert.
- Unmittelbar vor der ISR steht die #pragma-Compilerdirektive, die die ISR mit einem Interruptvektor verbindet.
- ISR „kommuniziert“ mit der Umgebung mit Hilfe globaler Variablen und/oder Flags.
- ISR darf/soll keine weiteren Funktionsaufrufe enthalten.

Hinweise zum Programmierstil

- ISR darf selbst nicht direkt aufgerufen werden.
- ISR kann aber indirekt aufgerufen werden, und zwar über das Setzen des dazugehörigen IFG-Bits per Software.
- Sollte das IFG nicht automatisch zurückgesetzt werden, so muss man das explizit „von Hand“ in der ISR machen.
- Interrupts können nicht verschachtelt werden: ISR kann durch keine weiteren (auch keine höher priorisierten) Interrupts unterbrochen werden, es sei denn, das wird in der ISR explizit freigegeben.
- Der Ablauf einer ISR soll so kurz wie möglich sein,
  - ggf. die weitere Verarbeitung an die CPU delegieren, d.h. die CPU mit der Anweisung __low_power_mode_off_on_exit() aus dem Schlafmodus in den Aktivmodus versetzen. Dort können auch komplexe Abläufe (mit Hilfe von Zustandsmaschinen) bearbeitet werden.

Interrupts

- Manche Interrupt-Quellen bei MSP430 sind direkt einem Interrupt-vektor zugeordnet und führen unmittelbar zu einer ISR
  - Reset und alle nicht maskierbaren Interrupts,
  - Watchdog Timer (WDTIFG),
  - Timers (TB0, TB1, TB2, TA0, TA1) mit Compare/Capture Register CCR0 (CCIFG0).

- Restliche Interrupt-Quellen werden gebündelt und sind ebenfalls Interruptvektoren zugeordnet, müssen aber über ein dazugehöriges Interruptvektor-Register (IV) selektiert werden.
  - Timer TA0 (drei IFGs: CCIFG1, CCIFG2 und TA0IFG über TA0IV),
  - USCI im UART-Modus (vier IFGs: UCA0STTIFG, UCA0TXCPTIFG, UCA0RXIFG und UXA0TXIFG über UCA0IV),
  - DMA, ADC, RTC

Interrupts

- Es gibt Interrupt-(Pending)-Flags IFG, die automatisch zurückgesetzt werden:
  - beim Aufruf der dazugehörigen ISR (z.B. das CCIFG bei Timer A/B aber nur mit Compare/Capture-Register CCR0),
  - beim Lesen eines dazugehörigen Peripherieregisters (z.B. das UCRXIFG beim Lesen UCAxRXBUF),
  - beim Schreiben in ein dazugehöriges Peripherieregister (z.B. das UCTXIFG beim Schreiben ins UCxTXBUF).
  - beim Lesen eines Interrupt-Vector-Registers (z.B. TAxIV).
- Andere Interrupt-(Pending)-Flags (z.B. TAxIFG) müssen per Software innerhalb der ISR zurückgesetzt werden.

- Interrupts sollen so selten wie möglich aber so häufig wie notwendig auftreten.

Timer: Interrupts

- Jedem Timer sind je zwei Interrupt-Vektoren zugeteilt:
  - Fünf Interrupt-Vektor für Interrupt-Flags CCIFG0 von TAxCCR0 bzw. TBxCCR0
  - Fünf Interrupt-Vektor für restliche Interrupt-Flags CCIFG und TAIFG von TAxCCR0 bzw. TBIFG von TBxCCR0

IR-Quelle	Priorität	Control-Register und IR-Flags	Vektornamen in C
TB0	59	TB0CCR0 CCIFG0	TIMER0_B0_VECTOR
TB0	58	TB0CCR1 CCIFG1 bis TB0CCR2 CCIFG2, TB0IFG	TIMER0_B1_VECTOR
TA0	53	TA0CCR0 CCIFG0	TIMER0_A0_VECTOR
TA0	52	TA0CCR1 CCIFG1 bis TA0CCR2 CCIFG2, TA0IFG	TIMER0_A1_VECTOR
TA1	49	TA1CCR0 CCIFG0	TIMER1_A0_VECTOR
TA1	48	TA1CCR1 CCIFG1 bis TA1CCR2 CCIFG2, TA1IFG	TIMER1_A1_VECTOR
TB1	46	TB1CCR0 CCIFG0	TIMER1_B0_VECTOR
TB1	45	TB1CCR1 CCIFG1 bis TB1CCR2 CCIFG2, TB1IFG	TIMER1_B1_VECTOR
TB2	43	TB2CCR0 CCIFG0	TIMER2_B0_VECTOR
TB2	42	TB2CCR1 CCIFG1 bis TB2CCR2 CCIFG2, TB2IFG	TIMER2_B1_VECTOR

Timer: Interrupts

Beispiel 3

```
// die LED am Pin 7 vom Port 2 soll mit
// einer Frequenz von ca. 1 Hz blinken
// => High dauert 0.5 Sek, Low dauert 0.5 Sek.

. . .
CSCTL0 = CSKEY;          // enable clock system
CSCTL1 = DCOFSEL_3;      // DCO frequency = 8.0 MHz
                         // select clock sources
CSCTL2 = SELA__DCOCLK    // ACLK  <- DCO
       | SELS__DCOCLK    // SMCLK <- DCO
       | SELM__DCOCLK;   // MCLK  <- DCO
                         // set frequency divider
CSCTL3 = DIVA__32        // ACLK  : /32 = 250.0 kHz
       | DIVS__32        // SMCLK : /32 = 250.0 kHz
       | DIVM__1;        // MCLK  : /1  =   8.0 MHz
CSCTL4 = XT1OFF          // XT1 disabled
       | XT2OFF;         // XT2 disabled
CSCTL0_H = 0;            // disable clock system
. . .


GLOBAL Void main(Void) {
   . . .
   TA0_init();
   while(TRUE) {
      _low_power_mode_3(); // do nothing
   }
}
```

```
GLOBAL Void TA0_init(Void) {
   CLRBIT(TA0CTL,   MC0 | MC1 // stop mode
                  | TAIE      // disable interrupt
                  | TAIFG);   // clear interrupt flag
   CLRBIT(TA0CCTL0, CM1 | CM0 // no capture mode
                  | CAP       // compare mode
                  | CCIE      // disable interrupt
                  | CCIFG);   // clear interrupt flag
   TA0EX0   = TAIDEX_4;       // set up expansion register /5
   TA0CCR0  = 3125-1;         // set up Compare Register CCR0
   TA0CTL   = TASSEL__ACLK    // 250.0 kHz
            | MC__UP          // Up Mode
            | ID__8           // /8
            | TACLR;          // clear and start Timer
   SETBIT(TA0CTL, TAIE);      // enable interrupt
}
```

```
#pragma vector = TIMER0_A1_VECTOR
__interrupt Void TIMER0_A1_ISR(Void) {
   TGLBIT(P2OUT, BIT7);       // toggle led
   CLRBIT(TA0CTL, TAIFG);     // clear interrupt flag
}
```

Timer: Interrupts

- Beispiel für Timer-Einstellung:
  - Abtastung von Tastern (Buttons)
  - Häufigkeit der Interrupts: 100x bis 10x pro Sekunde
  - der genaue Wert muss ggf. experimentell bestimmt werden
  - eine flexible Einstellung wäre wünschenswert
  - Taktquelle: XT1CLK mit 4,9152 MHz
  - Diese Taktquelle wird primär für die UART-Komponente mit Baudraten von 9600 bps bis 38400 bps benutzt

  - Wie müssen Frequenzteiler eingestellt werden?
  - Welcher Wert muss im Compare-Register gespeichert werden?

Timer: Interrupts

- Beispiel für Timer-Einstellung:
  - Wegen LPM3-Modus wird ACLK als Takt für UART und Timer selektiert
  - Takt für UART: 38400 x 16 = 614,4 kHz
  - Taktteiler DIVA := 4,9152 MHz / 614,4 kHz = 8
  - ACLK hat also eine Taktfrequenz von 614,4 kHz

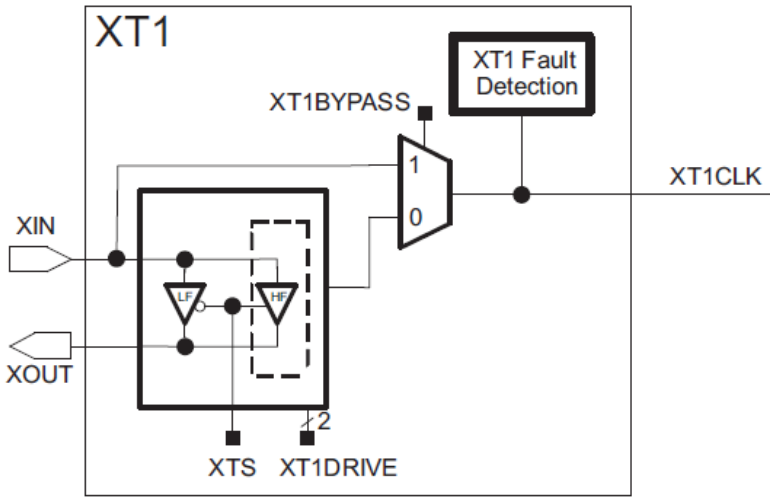
	10 ms => 100 Hz	100 ms => 10 Hz
Divider für Timer	/8 (76,8 kHz)	/8 (76,8 kHz)
Expansion Register	/8 (9,6 kHz)	/8 (9,6 kHz)
Compare Register	96	960

  - Einstellungen der Frequenzteiler bleiben konstant
  - Der Wert im Compare Register kann flexibel modifiziert werden

Timer: Interrupts

Beispiel 4

4,9152 MHz




```
CSCTL0 = CSKEY;          // enable clock system

CSCTL1 = DCOFSEL_3;      // DCO frequency = 8.0 MHz

                         // select clock sources
CSCTL2 = SELA__XT1CLK    // ACLK  <- XT1
       | SELS__DCOCLK    // SMCLK <- DCO
       | SELM__DCOCLK;   // MCLK  <- DCO

                         // set frequency divider
CSCTL3 = DIVA__8         // ACLK  : /8  = 614.4 kHz
       | DIVS__32        // SMCLK : /32 = 250.0 kHz
       | DIVM__1;        // MCLK  : /1  =   8.0 MHz

CSCTL4 = XT2OFF          // XT2 disabled
       | XTS             // XT1 HF mode
       | XT1DRIVE_0;     // XT1 low power, no bypass

CSCTL0_H = 0;            // disable clock system
```

Timer: Interrupts

Beispiel 4

Timer TA1

```
// Zeitbasis = 10 ms
// Teilungsfaktor = 614.4 kHz * Zeitbasis = 6144
// Skalierungsfaktor = Teilungsfaktor {/8} {/8} = 96

#pragma FUNC_ALWAYS_INLINE(TA1_init)
GLOBAL Void TA1_init(Void) {
   CLRBIT(TA1CTL,   MC0 | MC1  // stop mode
                  | TAIE       // disable interrupt
                  | TAIFG);    // clear interrupt flag
   CLRBIT(TA1CCTL0, CM1 | CM0  // no capture mode
                  | CAP        // compare mode
                  | CCIE       // disable interrupt
                  | CCIFG);    // clear interrupt flag
   TA1CCR0  = 96-1;            // set up Compare Register
   TA1EX0   = TAIDEX_7;        // set up expansion register
   TA1CTL   = TASSEL__ACLK     // 614.4 kHz
            | MC__UP           // Up Mode
            | ID__8            // /8
            | TACLR;           // clear and start Timer
   SETBIT(TA1CTL, TAIE);       // enable interrupt
}

#pragma vector = TIMER1_A1_VECTOR
__interrupt Void TIMER1_A1_ISR(Void) {
   if (TSTBIT(P1IN, BIT0 | BIT1)) {
      SETBIT(P1OUT, BIT2);
   } else {
      CLRBIT(P1OUT, BIT2);
   }
   CLRBIT(TA1CTL, TAIFG); // clear interrupt flag
}
```

Timer TA0

```
// Zeitbasis = 500 ms
// Teilungsfaktor = 614.4 kHz * Zeitbasis = 307200
// Skalierungsfaktor = Teilungsfaktor {/8} {/8} = 4800

#pragma FUNC_ALWAYS_INLINE(TA0_init)
GLOBAL Void TA0_init(Void) {
   CLRBIT(TA0CTL,   MC0 | MC1  // stop mode
                  | TAIE       // disable interrupt
                  | TAIFG);    // clear interrupt Flag
   CLRBIT(TA0CCTL0, CM1 | CM0  // no capture mode
                  | CAP        // compare mode
                  | CCIE       // disable interrupt
                  | CCIFG);    // clear interrupt flag
   TA0CCR0  = 4800-1;          // set up Compare Register
   TA0EX0   = TAIDEX_7;        // set up expansion register
   TA0CTL   = TASSEL__ACLK     // 614.4 kHz
            | MC__UP           // Up Mode
            | ID__8            // /8
            | TACLR;           // clear and start Timer
   SETBIT(TA0CTL, TAIE);       // enable interrupt
}

#pragma vector = TIMER0_A1_VECTOR
__interrupt Void TIMER0_A1_ISR(Void) {
   TGLBIT(P2OUT, BIT7);      // toggle led
   CLRBIT(TA0CTL, TAIFG);    // clear interrupt flag
}
```

Hinweise zum Programmierstil

- Komplexe Anweisungsfolgen oder Schleifen innerhalb einer ISR

ISR mit einer langen Folge von komplexen Operationen

```
#define N 32

#pragma vector = TIMER0_A0_VECTOR
__interrupt Void TA0_ISR(Void) {

   func(Daten[0]);
   func(Daten[1]);
   . . .
   func(Daten[N-1]);

}
```

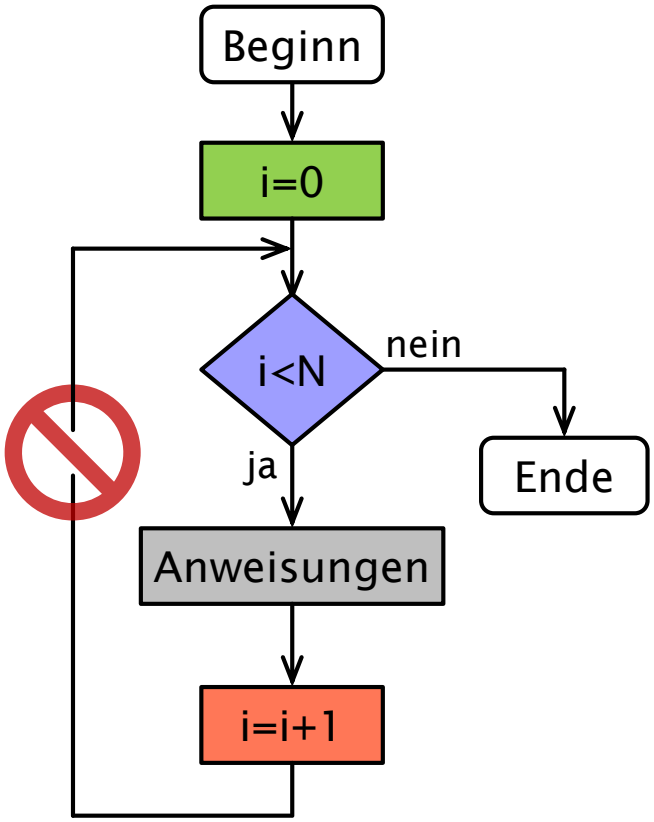
ISR mit einer for-Schleife mit komplexen Operationen

```
#define N 32

#pragma vector = TIMER0_A0_VECTOR
__interrupt Void TA0_ISR(Void) {

   unsigned int i;

   for (i=0; i<N; i++) {
      func(Daten[i]);
   }
}
```

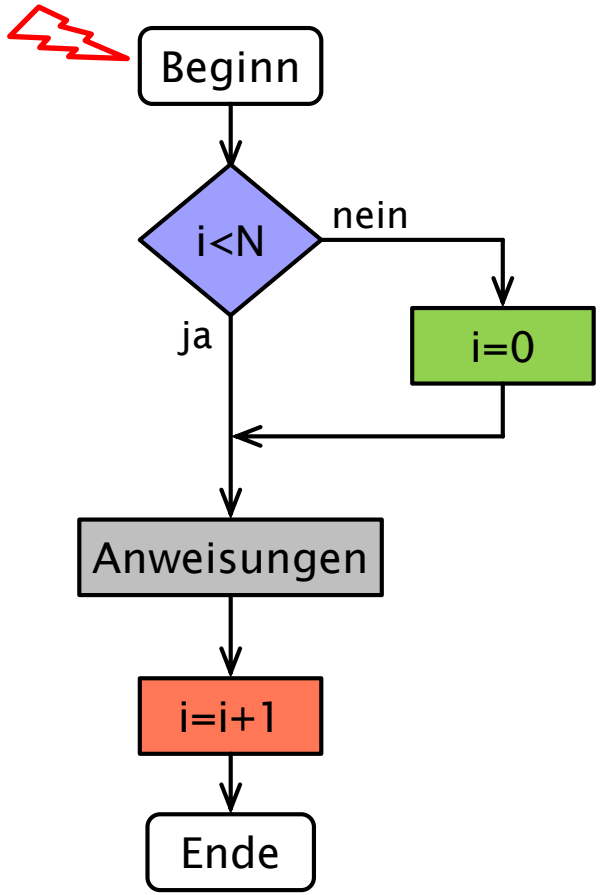
- Die Funktion func() steht hier stellvertretend für beliebige Funktionen oder komplexe Operationen (z.B. Modulo- oder Divisionsoperatoren)
- durch die relativ lange Dauer der Ausführung blockiert die ISR andere Interrupts (auch höher Priorisiert) und auch die Funktion main()
- Diese Art der Programmierung ist zu vermeiden

Hinweise zum Programmierstil

```
. . .
for ( i=0; i<N ; i++ ) {
   Anweisungen
}
. . .
```




```
. . .
Timer initialisieren
i=0;
Interrupt freischalten
. . .
```

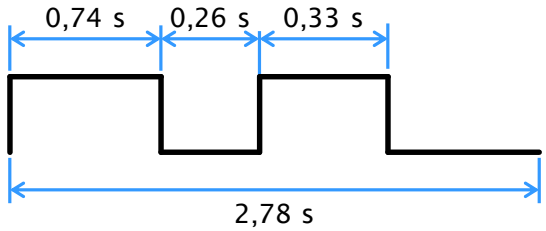


Timer-gesteuerte for-Schleife

```
#define N 32

static unsigned i=0;

#pragma vector = TIMER0_A0_VECTOR
__interrupt Void TA0_ISR(Void) {
   if (i >= N) {
      i = 0;
   }
   Anweisungen;
   i++;
}
```

Timer: Interrupts

- Wie erzeugt man ein periodisches, binäres Signals mit einem Timer?



- Zerlegung des Signals in einzelne High-/Low-Phasen
  - 740 ms für High, 260 ms für Low, 330 ms für High, 1450 ms für Low
- Zeitbasis = GGT(740, 260, 330, 1450) = 10 ms
- Timer solle mit ACLK getaktet werden.
- Der Tack ACLK wird von XT1 mit 4.9152 MHz und mit einem Teiler von 8 bereitgestellt.
- ACLK: 4.9152 MHz / 8 = 614.4 kHz

Timer: Interrupts

Beispiel 5

- TA0CCR0 := 614400 {/8} {/8} = 96 (für 10 ms)
  - Clock: CSCTL3.DIVx = /32
  - Timer: TA0CTL.ID = /8
  - Timer: TA0EX0.IDEX = /8
- Aufbau einer geeigneten Datenstruktur

```
#define HIGH 0x8000
#define LOW  0x0000

#define ACKFRQ   614.4  // kHz
#define TIMEBASE 10     // ms
#define SCALING  ((UInt)(ACKFRQ * TIMEBASE))
#define TICK(t)  (((SCALING / 8) / 8) * ((t) / TIMEBASE) - 1)
#define TABSIZE 5

// Zeiten in der Tabelle sind in ms
LOCAL const Int muster[TABSIZE] = {
   HIGH | TICK( 740), // High  740 ms ->  7104-1
   LOW  | TICK( 260), // Low   260 ms ->  2496-1
   HIGH | TICK( 330), // High  330 ms ->  3168-1
   LOW  | TICK(1450), // Low  1450 ms -> 13920-1
   0};

LOCAL const Int *ptr;
```

Timer: Interrupts

- Timer-gesteuerte Schleife

Beispiel 5

```
#prma FUNC_ALWAYS_INLINE(TA0_init)
GLOBAL Void TA0_init(Void) {

   ptr  = &muster[0];

   CLRBIT(TA0CTL, MC0 | MC1    // stop mode
                  | TAIE       // disable interrupt
                  | TAIFG);    // clear interrupt flag
   CLRBIT(TA0CCTL0, CM1 | CM0 // no capture mode
                  | CAP        // compare mode
                  | CCIE       // disable interrupt
                  | CCIFG);    // clear interrupt flag
   TA0CCR0   = 0;              // set up Compare Register
   TA0EX0    = TAIDEX_7;       // set up expansion register
   TA0CTL    = TASSEL__ACLK    // 614.4 kHz
             | MC__UP          // Up Mode
             | ID__8           // /4
             | TACLR;          // clear and start Timer

   SETBIT(TA0CTL, TAIE         // enable interrupt
                | TAIFG);      // set interrupt flag

}
```

```
#pragma vector = TIMER0_A1_VECTOR
__interrupt Void TIMER0_A1(Void) {

   UInt cnt = *ptr++;

   if (TSTBIT(cnt, HIGH)) {
      SETBIT(P2OUT, BIT7);
   } else {
      CLRBIT(P2OUT, BIT7);
   }

   CLRBIT(TA0CTL, TAIFG); // clear interrupt flag
   TA0CCR0 = ~HIGH BAND cnt;

   if (*ptr EQ 0) {
      ptr = &muster[0];
   }

}
```

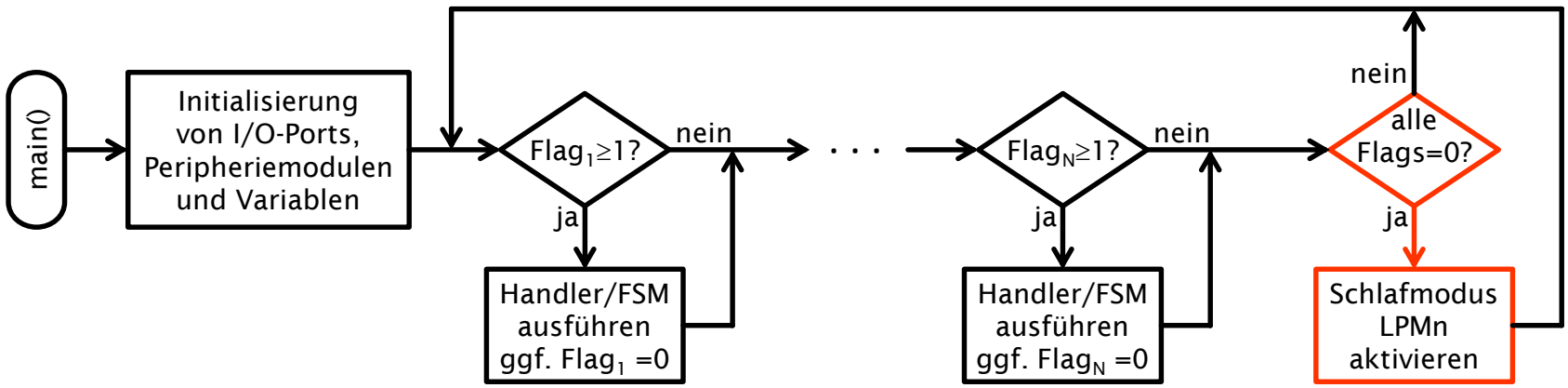
Indirekter Aufruf der ISR

Ereignisgesteuerte Abläufe

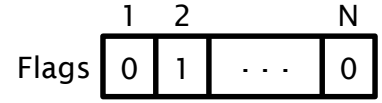
- Eventlose Verarbeitung
  - meistens triviale und relativ kurze Abläufe, die ausschließlich innerhalb einer ISR stattfinden
  - kein Event-Handler in der Funktion main() erforderlich
  - Beispiel: das Blinken einer LED in einer Timer-ISR
- Single State Event
  - das Aktivieren einer ISR führt dazu, dass die ISR ein Event generiert und den MSP430 in den Aktiv-Mode weckt. Die Verarbeitung des Events wird in die Funktion main() verlagert.
  - Die Behandlung des Interrupts ist innerhalb der ISR nicht möglich, weil zeitlich zu aufwändig, oder weil ggf. weitere Funktionen aufgerufen werden müssen.
  - Der Event-Handler in der Funktion main() kann das Event innerhalb eines einzelnen Durchlaufs der while-Schleife in der Funktion main() bearbeiten.
    Beispiel: Tastenentprellung (gesteuert durch Timer-Event)

- Multi State Event
  - Die Behandlung eines Interrupts findet in einem Handler außerhalb der ISR statt.
  - Der Ablauf kann rechen- und/oder zeitintensiv sein, und kann ggf. mehrere (häufig ineinander verschachtelte) Schleifen enthalten.
  - Der Einsatz typischer Schleifenanweisungen würde allerdings dazu führen, dass der Handler die CPU zu lange in Anspruch nimmt, und dadurch andere Handler aus der Funktion main() blockiert werden.
  - Damit die Aufrufe/Ausführung der Handler kooperativ stattfindet, ist es notwendig, Schleifen durch Zustandsmaschinen zu ersetzen.
    Beispiel: Suche nach einem String in einer Tabelle in einem Command-Interpreter.

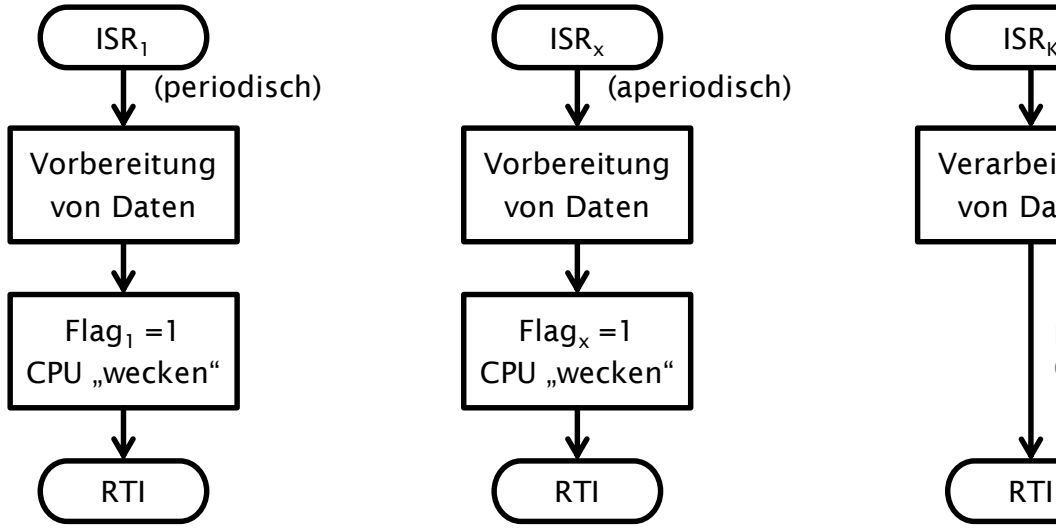
Ereignisgesteuerte Abläufe



Ablauf im Hauptprogramm



Interrupt Service Routinen



Beispiel 6

```
#include "..\base.h"

#ifndef EVENT_H_
#define EVENT_H_

typedef unsigned int TEvent;

#define NO_EVENTS   0x0000
#define EVENT_1     0x0001
#define EVENT_2     0x0002
#define EVENT_3     0x0004
  ...
#define EVENT_16    0x8000
#define ALL_EVENTS  0xFFFF

#define EVENT_IMA   EVENT_1  // I am alive
#define EVENT_BTN1  EVENT_2  // click on button 1
#define EVENT_BTN2  EVENT_3  // click on button 2

EXTERN Void Event_init(Void);
EXTERN Void Event_wait(Void);
EXTERN Void Event_set(TEvent);
EXTERN Void Event_clr(TEvent);
EXTERN Bool Event_tst(TEvent);
EXTERN Bool Event_err(Void);

#endif /* EVENT_H_ */
```

```
#include <msp430.h>
#include "..\base.h"
#include "event.h"

LOCAL TEvent event;
LOCAL TEvent errflg;

#pragma FUNC_ALWAYS_INLINE(Event_init)
GLOBAL Void Event_init(Void) {
   event  = NO_EVENTS;
   errflg = NO_EVENTS;
}

#pragma FUNC_ALWAYS_INLINE(Event_set)
GLOBAL Void Event_set(TEvent arg) {
   errflg  |= event BAND arg;
   TGLBIT(event, arg);
}

#pragma FUNC_ALWAYS_INLINE(Event_clr)
GLOBAL Void Event_clr(TEvent arg) {
   TGLBIT(event, arg);
}

. . .

#pragma FUNC_ALWAYS_INLINE(Event_err)
GLOBAL Bool Event_err(Void) {
   return (errflg NE NO_EVENTS);
}
```

Ereignisgesteuerte Abläufe

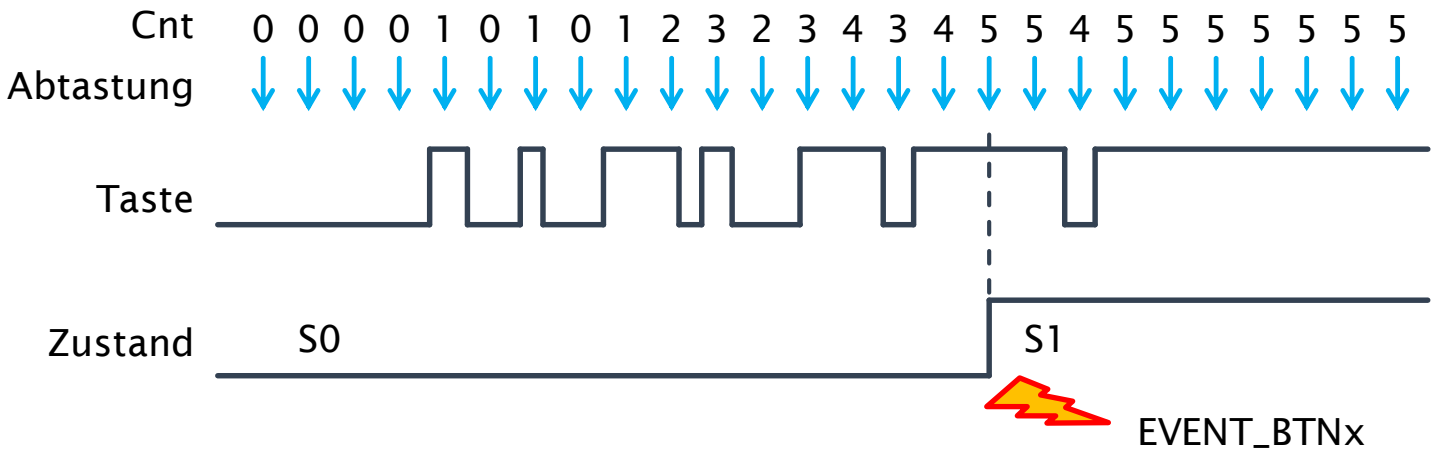
Beispiel 6

```
// TA0.c

#pragma vector = TIMER0_A1_VECTOR
__interrupt Void TIMER0_A1_ISR(Void) {

   Event_set(EVENT_IMA);              // set up event
   CLRBIT(TA0CTL, TAIFG);             // clear interrupt flag
   __low_power_mode_off_on_exit();    // restore Active Mode on return

}
```

```
// main.c
GLOBAL Void main(Void) {
   // Initialisierungen
   ...
   while(TRUE) {

      Event_wait();

      if (Event_tst(EVENT_IMA)) {
         TGLBIT(P2OUT, BIT7);
         Event_clr(EVENT_IMA);
      }
   }
}
```

```
// event.c
   ...

#pragma FUNC_ALWAYS_INLINE(Event_wait)
GLOBAL Void Event_wait(Void) {
   _disable_interrupt();
   if (event EQ NO_EVENTS) {
      _low_power_mode_3();
   }
   _enable_interrupt();
}
```

- Entprellen von Tasten mit einer Hysterese



State	Taste	Cnt		State⁺	Cnt⁺	Event
S0	0	=0		S0	Cnt	0
S0	0	>0		S0	Cnt-1	0
S0	1	<N-1		S0	Cnt+1	0
S0	1	=N-1		S1	Cnt	1
S1	1	=N-1		S1	Cnt	0
S1	1	<N-1		S1	Cnt+1	0
S1	0	>0		S1	Cnt-1	0
S1	0	=0		S0	Cnt	0

Ereignisgesteuerte Abläufe

State	Taste	Cnt		State+	Cnt+	Event
S0	0	=0		S0	Cnt	0
S0	0	>0		S0	Cnt-1	0
S0	1	<N-1		S0	Cnt+1	0
S0	1	=N-1		S1	Cnt	1
S1	1	=N-1		S1	Cnt	0
S1	1	<N-1		S1	Cnt+1	0
S1	0	>0		S1	Cnt-1	0
S1	0	=0		S0	Cnt	0

State	Taste	Cnt		State+	Cnt+	Event
-	0	=0		S0	Cnt	0
-	0	>0		State	Cnt-1	0
-	1	<N-1		State	Cnt+1	0
S0	1	=N-1		S1	Cnt	1
S1	1	=N-1		S1	Cnt	0

Taste	Cnt	State		State+	Cnt+	Event
0	=0			S0		
	else				Cnt-1	
else	<N-1				Cnt+1	
	else	S0		S1		1

```
if (Taste EQ 0) {
   if (Cnt EQ 0) {
      State = S0;
   } else {
      Cnt -= 1;
   }
} else {
   if (Cnt LT N-1) {
      Cnt += 1;
   } else {
      if (State EQ S0) {
         State = S1;
         Event();
      }
   }
}
```