

Inhaltsübersicht

1. Einführung in Mikrocontroller
2. Der Cortex-M0-Mikrocontroller
- 3. Programmierung des Cortex-M0**
4. Nutzung von Peripherieeinheiten
5. Exceptions und Interrupts

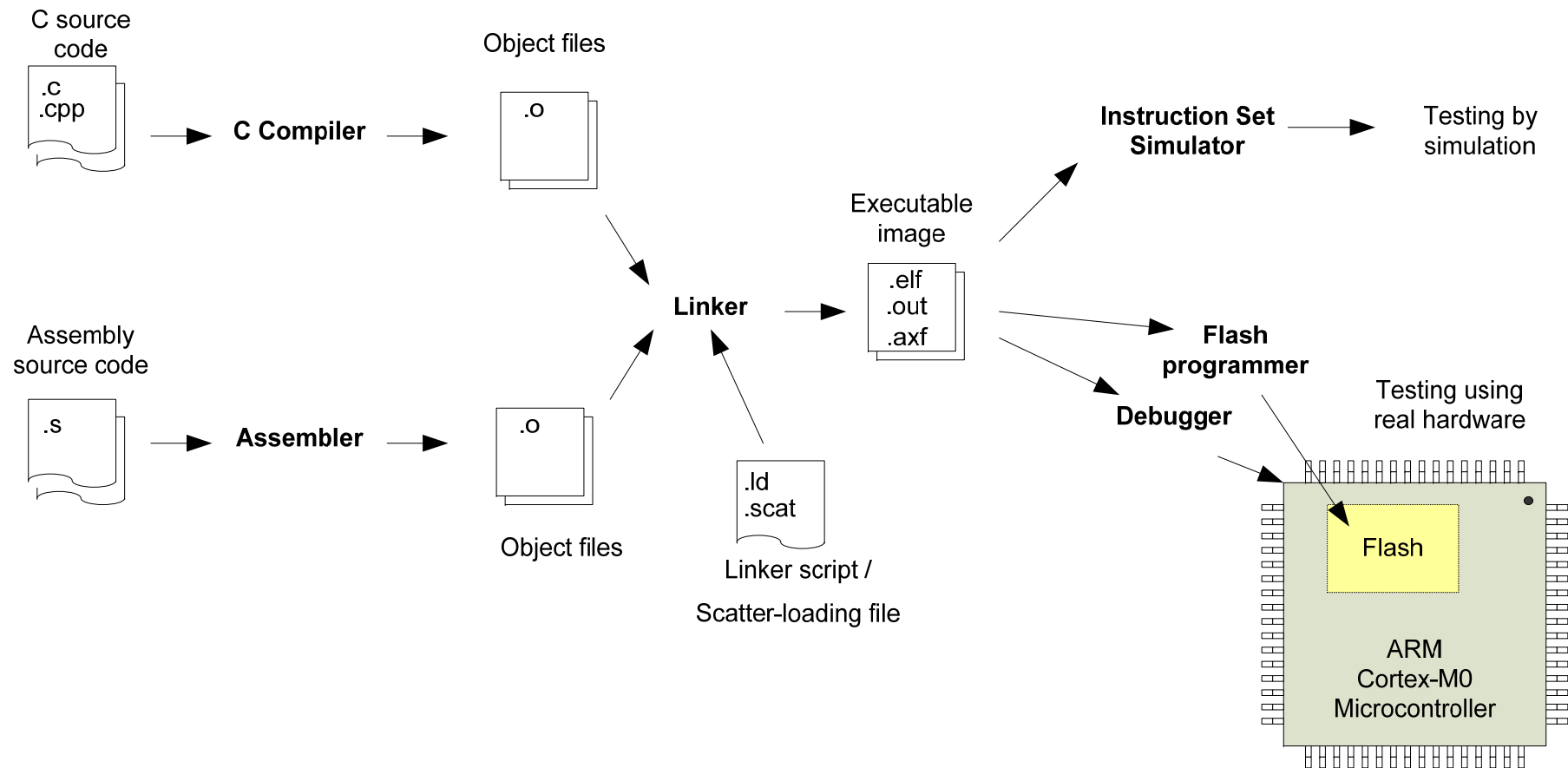
Kapitelübersicht

- I. Vom Quellcode zum Programm
- II. Vom Reset zum Hauptprogramm
- III. Zugriff auf Peripherieregister in C
- IV. CMSIS
- V. Programmiertechniken

Was sind die Bestandteile eines Programms?

- Ein auf dem Cortex lauffähiges Programm besteht normalerweise aus verschiedenen Modulen.
- Die Quellcodes der Module können in Assembler oder in C programmiert sein.
- Einige Module sind zu Beginn eines Projektes schon vorhanden (Startup-Code, Bibliotheksfunktionen), andere müssen selbst programmiert werden (zumeist in C).

Erzeugen eines ablauffähigen Programms



Bildquelle: Joseph Yiu, The Definitive Guide to the ARM Cortex-M0

Erzeugen eines ablauffähigen Programms (2)

- Der Quellcode jedes Moduls muss übersetzt werden in eine so genannte „Objektdatei“.
 - Eine Objektdatei (Endung „.obj“) enthält schon den Binärcode (Maschinencode) für den gewählten Prozessor, allerdings ist dieser i.d.R. noch verschiebbar (engl.: relocatable).
 - C-Code wird mit einem C-Compiler übersetzt, Assembler-Code mit einem Assembler.

Erzeugen eines ablauffähigen Programms (2)

- Der so genannten „Linker“ bindet die Module zu einem auf dem Ziel-System ausführbaren Programm zusammen („Executable Image“).
 - Der Linker legt insbesondere Adressen fest - für die Programmteile und auch für die Daten (statische Daten, Stack, Heap) - und ordnet damit die Module im Speicher an
 - Der Linker benötigt dazu Informationen zum Speichersystem (z.B. Flash Speicher für Programm und Konstanten, SRAM Speicher für Daten)
 - Diese Informationen können in den Tools eingestellt werden oder über ein „Linker-Skript“ (auch: „Scatter-Loading-File“) eingegeben werden.

Absolute und verschiebbare Module

Quellcode

```
    mov A, #01h
LOOP: add A, #01h
    mov R1, A
    ljmp LOOP
```

Assembler



Modul

```
74 01
24 01
F9
02 0002
```

Adresse	Inhalt
Start+7	+2
Start+6	Start
Start+5	02
Start+4	F9
Start+3	01
Start+2	24
Start+1	01
Start+0	74

verschiebbares
Modul:
Linker legt
Start-Adresse
später fest

Adresse	Inhalt
0007	02
0006	00
0005	02
0004	F9
0003	01
0002	24
0001	01
0000	74

absolutes
Modul

Beispiel: Verschieben des Codes

Start = 0200

0200	7401	4		mov A, #01h
0202	2401	5	LOOP:	add A, #01h
0204	F9	6		mov R1, A
0205	020202	7		ljmp LOOP

Start = 8000

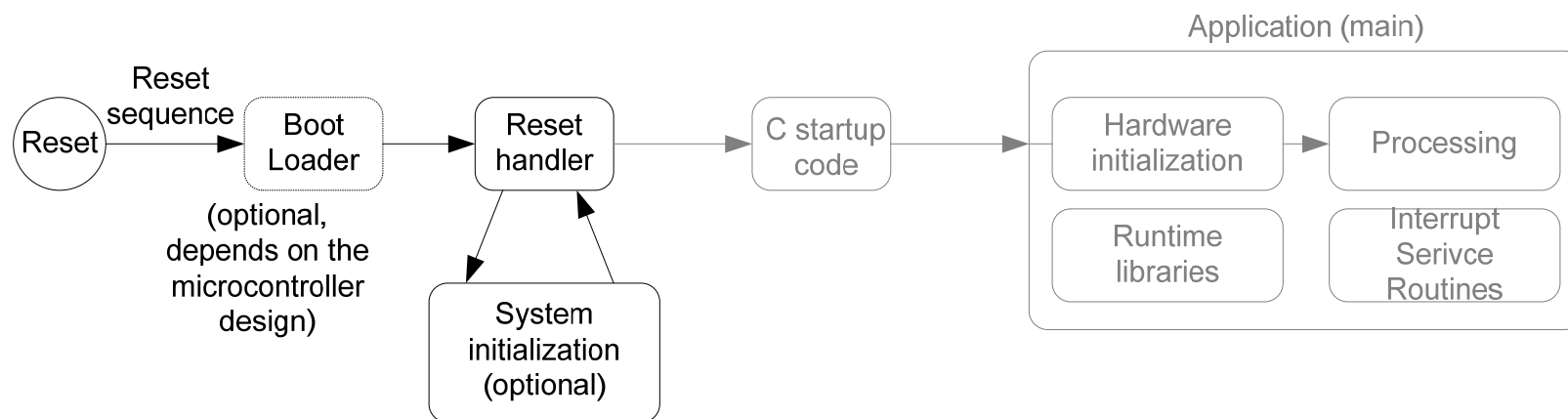
8000	7401	4		mov A, #01h
8002	2401	5	LOOP:	add A, #01h
8004	F9	6		mov R1, A
8005	028002	7		ljmp LOOP

Kapitelübersicht

- I. Vom Quellcode zum Programm
- II. Vom Reset zum Hauptprogramm**
- III. Zugriff auf Peripherieregister in C
- IV. CMSIS
- V. Programmiertechniken

Was passiert beim Reset des Prozessors?

- Wir benutzen kein Betriebssystem auf dem Cortex-M0.
- Es muss daher einen Weg vom Reset des Prozessors bis zum Ausführen der „main“-Funktion unseres C-Programms geben.
- Erster Schritt: Prozessor springt zum „Reset Handler“ (siehe letztes Kapitel: Reset-Sequenz)



Bildquelle: Joseph Yiu, The Definitive Guide to the ARM Cortex-M0

Inhalt des Startup-Codes

- Startup-Code in „startup_NUC1xx.s“
- Reset-Handler
 - Sprung nach „__main“
- Größe von Heap und Stack
- Vektortabelle, u.v.m.

Ausschnitt aus „startup_NUC1xx.s

```
...
Stack_Size      EQU      0x00000400
...

Heap_Size       EQU      0x00000000
...

Reset_Handler   PROC
...

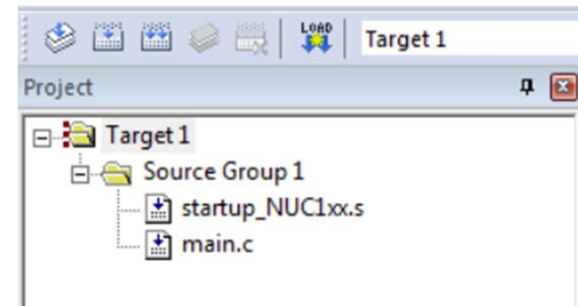
                LDR      R0, = __main
                BX       R0
                ENDP
```

Initialisierungsfunktion

- Der Reset-Handler führt die Initialisierungsfunktion „_main“ aus
- Dies ist der so genannte C-Startup-Code
- Initialisierung von globalen Variablen etc.
- Dieser Code wird vom C-Compiler/Linker automatisch eingefügt. Er erscheint nicht in der Projektverwaltung!
- Von hier erfolgt der Sprung zur vom Anwender programmierten „main“-Funktion.

Notwendige Quellen für Projekt

- Um ein minimal lauffähiges Programm für den Simulator zu erhalten benötigen wir:
 - Startup-Code:
„startup_NUC1xx.s“
 - Main-Programm:
„main.c“
 - Weitere Header-Dateien und Quellen werden für die Arbeit mit der Peripherie notwendig (später).



```
int main (void) {  
    //Hier geht's los  
}
```

Kapitelübersicht

- I. Vom Quellcode zum Programm
- II. Vom Reset zum Hauptprogramm
- III. Zugriff auf Peripherieregister in C**
- IV. CMSIS
- V. Programmiertechniken

Register der Peripherieeinheiten

- Die Steuer- und Daten-Register von Peripherieeinheiten liegen an bestimmten Adressen im Speicher (siehe Datenblatt des Chips, Ausschnitt aus NUC130_Datasheet)

5.5.4 Register Map

R: read only, W: write only, R/W: both read and write

Register	Offset	R/W	Description	Reset Value
GP_BA = 0x5000_4000			Basisadresse	
GPIOA_PMD	GP_BA+0x000	R/W	GPIO Port A Pin I/O Mode Control	0xFFFF_FFFF
GPIOA_OFFD	GP_BA+0x004	R/W	GPIO Port A Pin OFF Digital Enable	0x0000_0000
GPIOA_DOUT	GP_BA+0x008	R/W	GPIO Port A Data Output Value	0x0000_FFFF
GPIOA_DMASK	GP_BA+0x00C	R/W	GPIO Port A Data Output Write Mask	0x0000_0000
GPIOA_PIN	GP_BA+0x010	R	GPIO Port A Pin Value	0x0000_XXXX
GPIOA_DBEN	GP_BA+0x014	R/W	GPIO Port A De-bounce Enable	0x0000_0000
GPIOA_IMD	GP_BA+0x018	R/W	GPIO Port A Interrupt Mode Control	0x0000_0000
GPIOA_IEN	GP_BA+0x01C	R/W	GPIO Port A Interrupt Enable	0x0000_0000
GPIOA_ISRC	GP_BA+0x020	R/W	GPIO Port A Interrupt Source Flag	0XXXXX_XXXX

Zugriff auf Peripherieregister in C

- Um aus C auf die Register zugreifen zu können, definiert man einen Zeiger auf eine Adresse.
- Der Datentyp des Zeigers bestimmt dabei die Breite der Daten.

Beispiel

Adresse	Inhalt	Register
0x4000_0008	0x00	
0x4000_0007	0x00	
0x4000_0006	0x00	REG3
0x4000_0005	0x00	REG2
0x4000_0004	0x00	REG2
0x4000_0003	0x00	REG1
0x4000_0002	0x00	REG1
0x4000_0001	0x00	REG1
0x4000_0000	0x00	REG1

Basisadresse



Beispiel

```
#include "stdint.h"

#define PERIPH_BASE 0x40000000
#define REG1 *((volatile uint32_t *) PERIPH_BASE)
#define REG2 *((volatile uint16_t *) (PERIPH_BASE+4))
#define REG3 *((volatile uint8_t *) (PERIPH_BASE+6))

//Main called by startup_NUC1xx.s
int main (void){
    uint32_t a;
    uint16_t b;
    uint8_t c;

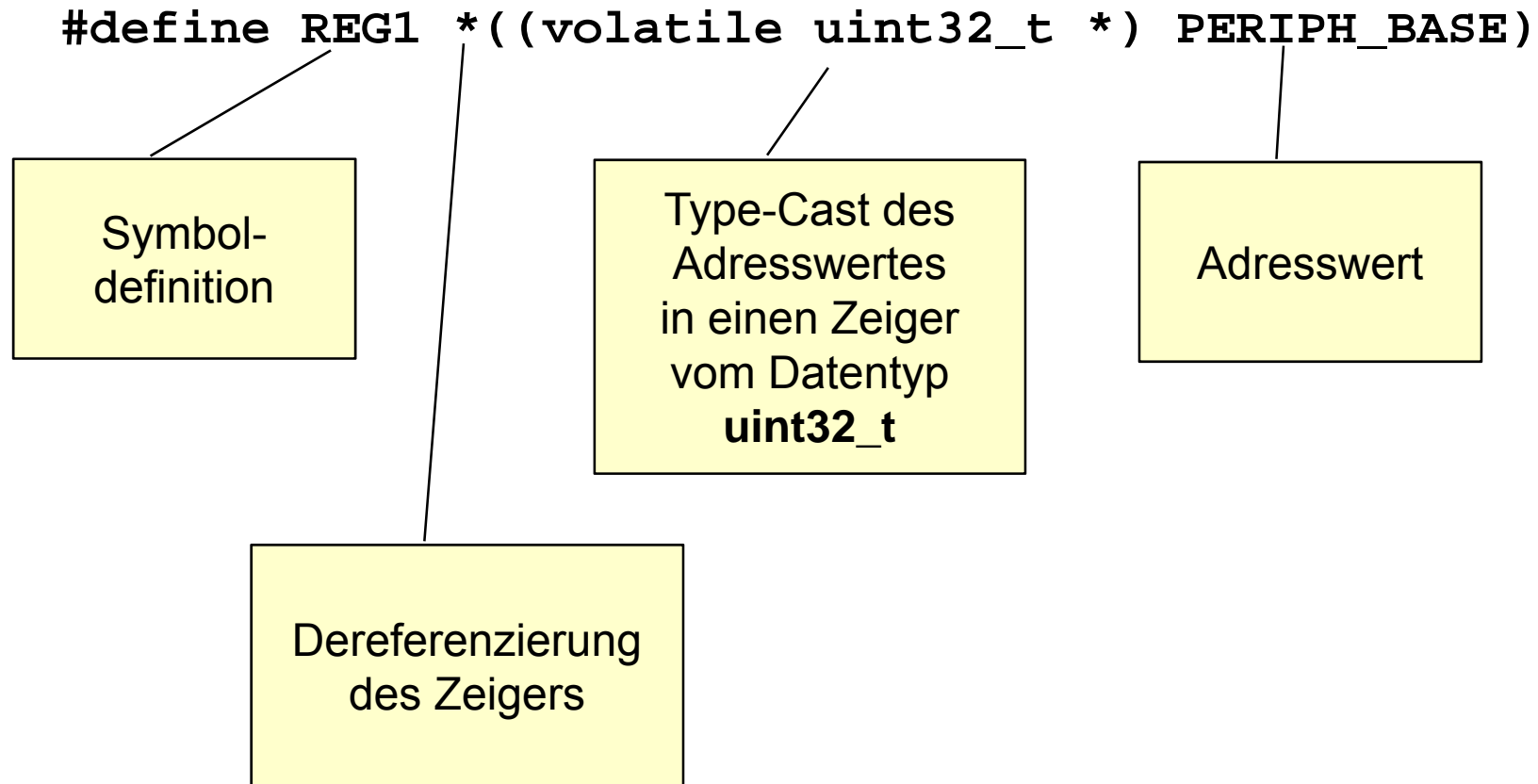
    REG1 = 0x12345678;
    REG2 = 0x55aa;
    REG3 = 0xbb;

    a = REG1;
    b = REG2;
    c = REG3;
}
```

Adresse	Inhalt	Register
0x4000_0008	0x00	
0x4000_0007	0x00	
0x4000_0006	0xbb	REG3
0x4000_0005	0x55	REG2
0x4000_0004	0xaa	REG2
0x4000_0003	0x12	REG1
0x4000_0002	0x34	REG1
0x4000_0001	0x56	REG1
0x4000_0000	0x78	REG1

Achtung: Bitte dieses und die nachfolgenden Beispiele nur im Simulator laufen lassen, nicht auf dem Labor-Board!

Zeiger auf Register



Zeiger auf Register (2)

- Für den Zugriff wird ein Symbol definiert:
 - Die Adresse wird in einen Zeiger eines bestimmten Datentyps gewandelt.
 - Dieser Zeiger wird „dereferenziert“, damit Zugriff auf Speicherstellen mit diesem Symbol (lesend oder schreibend)
 - Datengröße bei Zuweisungen beachten!

Der „volatile“ Type Qualifier

- „**volatile**“ verhindert Compiler-Optimierungen, so dass die Speicherstellen auch immer gelesen oder geschrieben werden. Ist für alle Peripherieeinheiten sinnvoll, da hier die Daten von „außen“ verändert werden können.
- Mit „**volatile const**“ kann verhindert werden, dass der Prozessor auf die Speicherstellen schreiben kann, z.B. bei Inputs (Compiler-Fehler falls dies versucht wird).
- Folgende Definitionen für die Qualifier sind in CMSIS vorhanden (core_cm0.h):

```
#define    __I        volatile const    //Input, d.h. Read-Only
#define    __O        volatile          //Output
#define    __IO       volatile          //Input-Output
```

Zugriff mit Strukturen

- Bequemer wird der Zugriff auf Peripherieeinheiten, wenn C-Strukturen verwendet werden.
- Idee: Man definiert eine Struktur, welche die Register der Peripherieeinheit abbildet und konvertiert die Anfangsadresse in einen Zeiger auf die Struktur.
- Wichtig: Die Größen der Datentypen der einzelnen Bestandteile der Struktur müssen mit den Registergrößen übereinstimmen.
- Vorteil: Die Anfangsadressen der einzelnen Register müssen nicht selbst ausgerechnet werden, sondern werden über die Datentypen festgelegt.

Beispiel

```
#include "stdint.h"

typedef struct {
    volatile uint32_t REG1;
    volatile uint16_t REG2;
    volatile uint8_t  REG3;
} PERIPH_T;

#define PERIPH1  ((PERIPH_T *) 0x40000000)

//Main called by startup_NUC1xx.s
int main (void){
    uint32_t a;
    uint16_t b;
    uint8_t c;

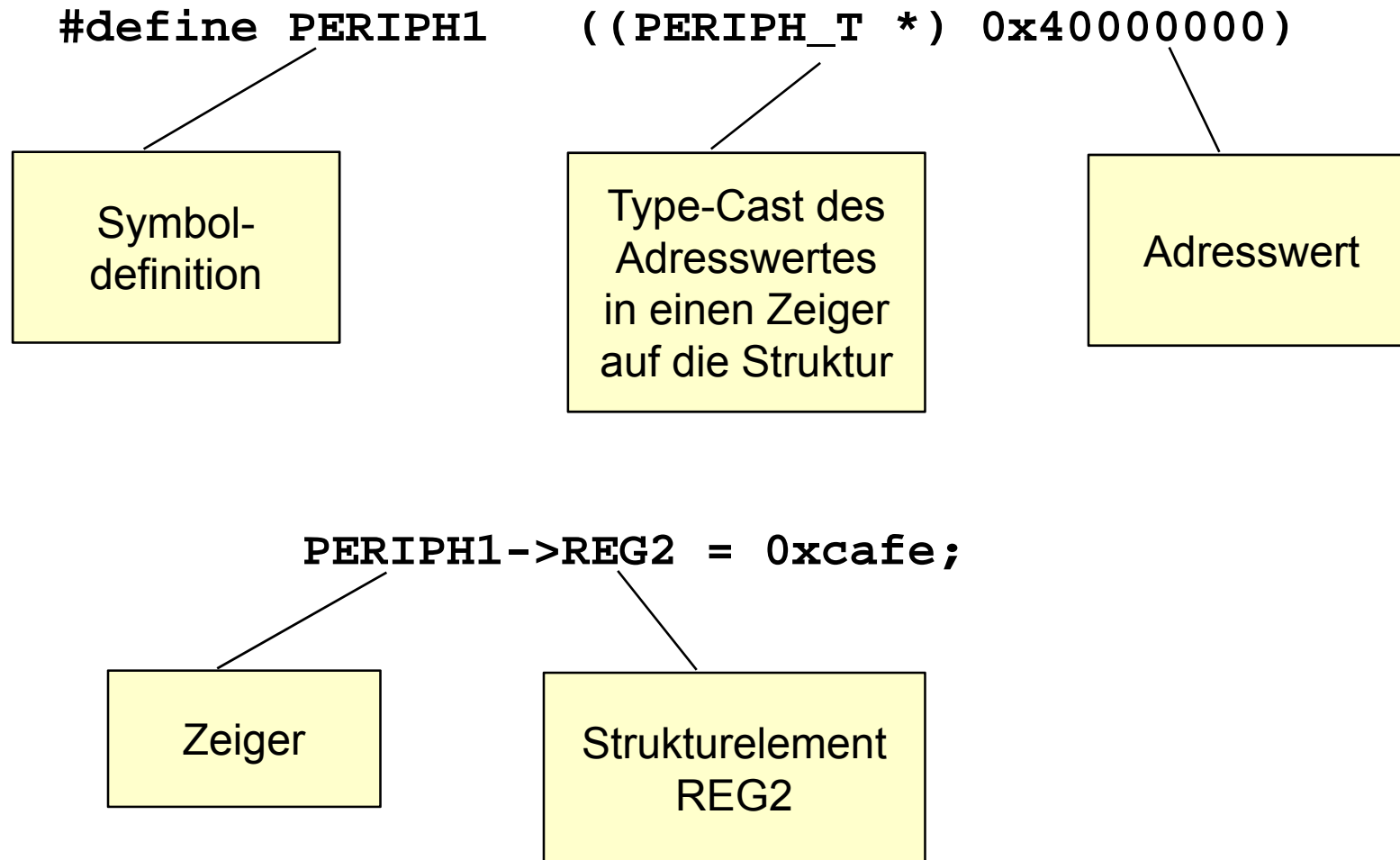
    PERIPH1->REG1 = 0xdeadbeef;
    PERIPH1->REG2 = 0xcafe;
    PERIPH1->REG3 = 0x55;

    a = PERIPH1->REG1;
    b = PERIPH1->REG2;
    c = PERIPH1->REG3;
}
```

typedef: Definition als
neuer Datentyp PERIPH_T
für nachfolgende Konversion in Zeiger

Adresse	Inhalt	Register
0x4000_0008	0x00	
0x4000_0007	0x00	
0x4000_0006	0x55	REG3
0x4000_0005	0xCA	REG2
0x4000_0004	0xFE	REG2
0x4000_0003	0xDE	REG1
0x4000_0002	0xAD	REG1
0x4000_0001	0xBE	REG1
0x4000_0000	0xEF	REG1

Zeiger auf die Struktur



Nutzung von Bitfeldern

- Durch Bitfelder können einzelne Bits in Registern gelesen und geschrieben werden
- Ein Bitfeld wird in C als Struktur definiert.
- Bitfelder können nur für 32-Bit Datentypen definiert werden, also hier „uint32_t“.
- Die Bestandteile (=Felder) der Struktur sind ebenfalls vom Typ „uint32_t“, es wird aber die Bitbreite nach dem Doppelpunkt spezifiziert.
 - Folge: Die Felder werden nacheinander mit der spezifizierten Bitbreite im Speicher abgelegt.
 - Ggf. werden mehrere 32-Bit Worte belegt.
- Der Zugriff auf Bitfelder ist langsamer als auf die normalen C-Datentypen.

Beispiel

```
#include "stdint.h"

typedef struct {
    volatile uint32_t FIELD1:4;
    volatile uint32_t FIELD2:4;
    volatile uint32_t RESERVED:24;
} REG_T;

#define REG1    ((REG_T *) 0x40000000)

//Main called by startup_NUC1xx.s
int main (void){

    REG1->FIELD1 = 0x3;
    REG1->FIELD2 = 0x5;
    REG1->RESERVED = 0xFFFFFFFF;

    a = REG1->FIELD2;
}
```

a = 0x00000005

Adresse	Inhalt	Register
0x4000_0008	0x00	
0x4000_0007	0x00	
0x4000_0006	0x00	
0x4000_0005	0x00	
0x4000_0004	0x00	
0x4000_0003	0xFF	REG1
0x4000_0002	0xFF	REG1
0x4000_0001	0xFF	REG1
0x4000_0000	0x53	REG1

Überlagerung von Register und Feldern

- Mit Hilfe einer „union“ kann sowohl auf die Felder eines Registers als auch auf das Register als Ganzes zugegriffen werden.
- Bei einer „union“ (dt.: Vereinigung) werden die einzelnen Elemente überlagert und belegen (im Unterschied zur „struct“) den gleichen Speicherplatz.
- Verwendung in einer Register-Struktur (siehe Beispiel):
 - Die Union wird anonym deklariert (`#pragma anon_unions`), um sich die Referenzierung der Union beim Zugriff auf die Elemente innerhalb der Register-Struktur zu sparen.
 - Der Zugriff auf die Bitfeld-Elemente erfolgt dann über den „.“-Operator innerhalb der Register-Struktur!

Beispiel

Adresse	Inhalt	Register
0x4000_0008	0x00	
0x4000_0007	0xFF	REG2
0x4000_0006	0xFF	REG2
0x4000_0005	0xFF	REG2
0x4000_0004	0x53	REG2
0x4000_0003	0xDE	REG1
0x4000_0002	0xAD	REG1
0x4000_0001	0xBE	REG1
0x4000_0000	0xEF	REG1

```
#include "stdint.h"
#pragma anon_unions
```

„anonyme“ Unions ermöglichen

```
typedef struct {
    volatile uint32_t REG1;
    union {
        volatile uint32_t u32REG2;
        struct {
            volatile uint32_t FIELD1:4;
            volatile uint32_t FIELD2:4;
            volatile uint32_t RESERVED:24;
        } REG2;
    };
} PERIPH_T;
```

REG2 als union

```
#define PERIPH1 ((PERIPH_T *) 0x40000000)
```

```
//Main called by startup_NUC1xx.s
```

```
int main (void){
    uint32_t a;
```

```
    PERIPH1->REG1 = 0xdeadbeef;
```

```
    PERIPH1->u32REG2 = 0xFFFFFFFF;
```

```
    PERIPH1->REG2.FIELD1 = 0x3;
```

```
    PERIPH1->REG2.FIELD2 = 0x5;
```

```
    a = PERIPH1->u32REG2;
```

```
}
```

a = 0xFFFFFFFF53

Kapitelübersicht

- I. Vom Quellcode zum Programm
- II. Vom Reset zum Hauptprogramm
- III. Zugriff auf Peripherieregister in C
- IV. CMSIS**
- V. Programmiertechniken

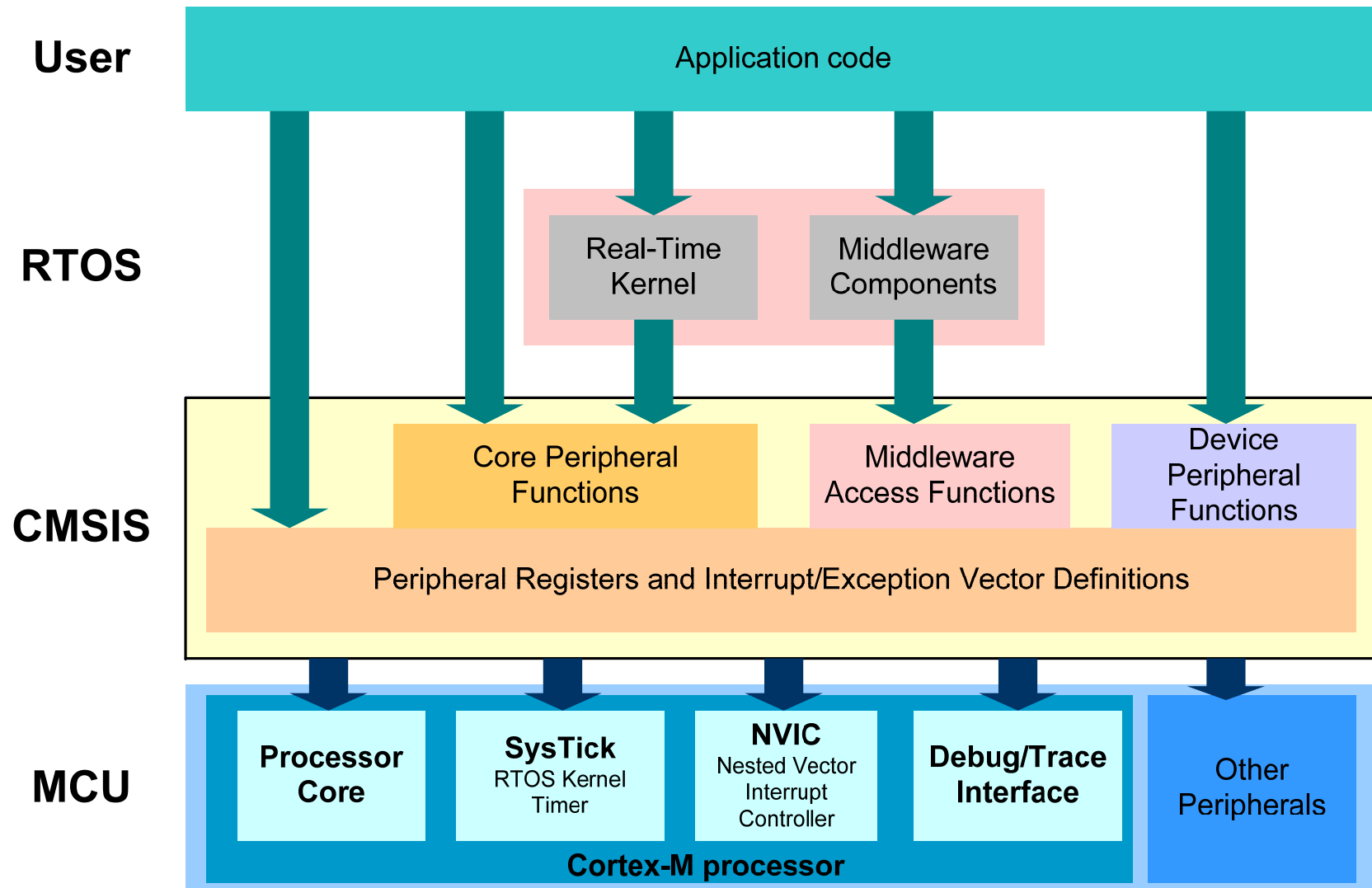
Was ist CMSIS?

- Steht für „Cortex Microcontroller Software Interface Standard“
- Bietet eine standardisierte (Programmier-) Schnittstelle zur Prozessor-Hardware, z.B. Interrupt Controller NVIC, System Control
- Soll die Wiederverwendbarkeit und Portierbarkeit der erstellten Software fördern
- Aktuelle Version ist 2.0

Was bietet CMSIS?

- Zugriffsfunktionen auf Peripherieeinheiten, die zum Core gehören (NVIC, System Control Block, System Tick Timer)
- Registerdefinitionen für die Core-Peripherie
- Zugriff auf spezielle Maschinenbefehle des Cortex
- Standardisierte Namen für die Exception Handler
- Standardisierter Namen für die System-Initialisierungsfunktion
- Standardisierte Variable für Informationen zur Taktfrequenz des Prozessors
- Vorgaben für die Peripherie-Treiber-Bibliothek der Hersteller (Peripheral Drivers)

Struktur von CMSIS

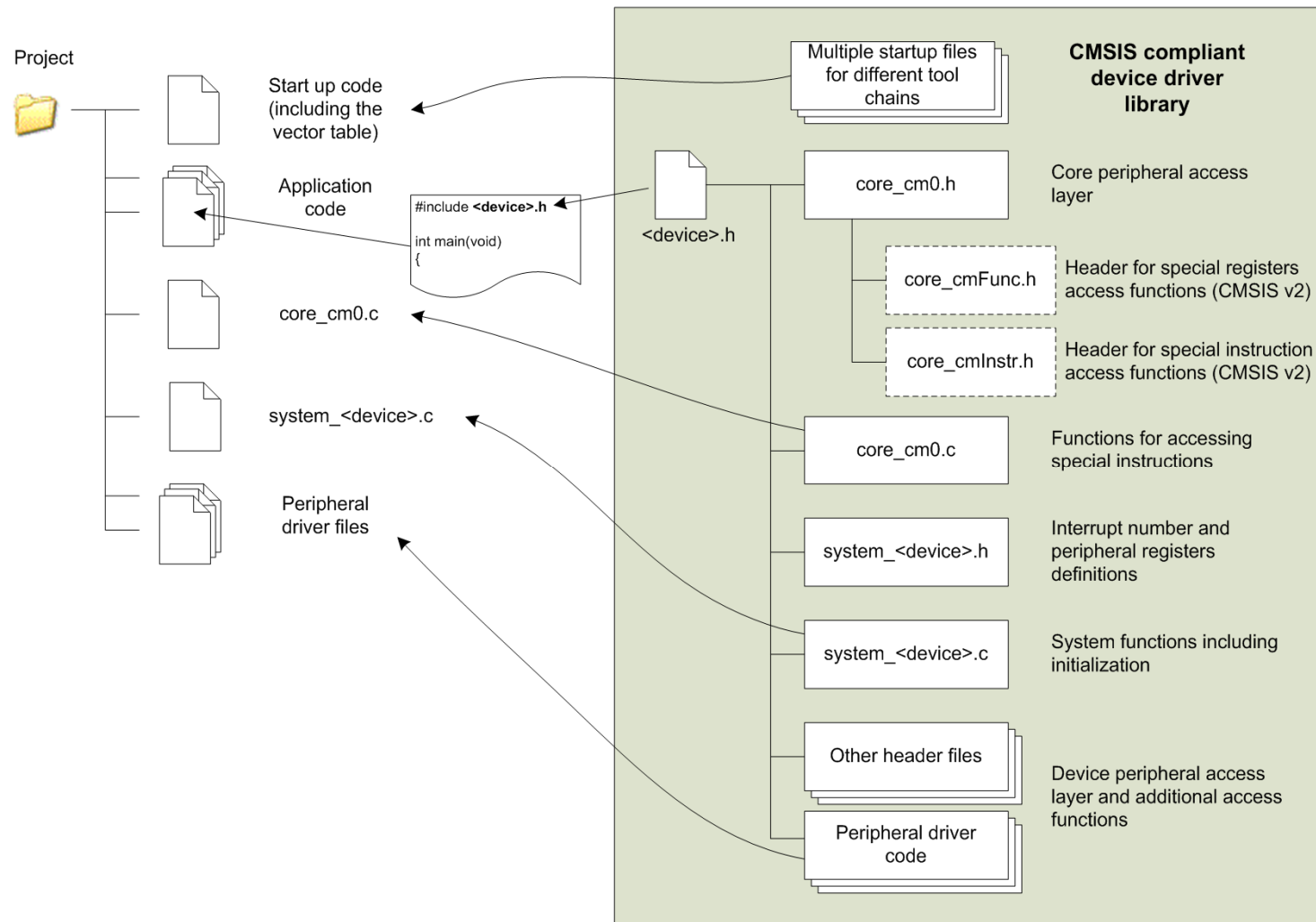


Bildquelle: Joseph Yiu, The Definitive Guide to the ARM Cortex-M0

Nutzung von CMSIS

- Die Chip-Hersteller bieten i.d.R. alle notwendigen Quellcodes an
 - Z.B. Nuvoton „Board Support Package“
 - Oder in den Installationsverzeichnissen der KEIL-SW
 - Im Labor werden die Dateien vorgegeben
- Die entsprechenden Dateien müssen dann zum Projekt hinzugebunden werden.
- Legt man mit der μ Vision Projekte an, so sind einige Dateien schon eingebunden.

Dateistruktur von CMSIS-Projekten



Bildquelle: Joseph Yiu, The Definitive Guide to the ARM Cortex-M0

Inhalte der CMSIS-Dateien

- Startup-Code (im Labor „startup_NUC1xx.s“)
- <device.h> (im Labor „NUC1xx.h“):
 - Referenziert die anderen Header-Dateien
 - Definiert u.a. die Register-Strukturen des Chips
- „core_cm0.h“ und „core_cm0.c“: Implementiert die Definitionen und Funktionen für den Cortex-Kern („Core Peripheral Access Layer“)
 - Ist normalerweise durch Anlegen des Projektes eingebunden
 - Im Labor: Bestandteil des Verzeichnisses „_Driver“

Vereinfachungen für das Labor

- „NUC1xx.h“:
 - Referenziert „core_cm0.h“
- „Driver_M_Dongle.h“:
 - Referenziert „NUC1xx.h“
 - Makros und Funktionen für wichtige Peripherieeinheiten
- „BoardConfig.h“:
 - Referenziert „Driver_M_Dongle.h“
 - Definitionen von Port-Pins

Beispiel: GPIO-Programmierung

- Der NUC130 verfügt über 80 so genannte „General Purpose Input/Output“-Pins (GPIO). Dies sind bidirektionale Pins, die als Input oder Output benutzt werden können.
- Die 80 Pins sind in 5 Gruppen zu je 16 Pins geordnet: GPIOA, GPIOB, GPIOC, GPIOD und GPIOE.
- Die fünf GPIOs können über jeweilige Strukturen, die in „NUC1xx.h“ definiert sind, angesprochen werden (Mit Zeigern darauf) oder über Funktionen und Makros aus „Driver_M_Dongle“.
- Informationen zu den GPIOs: Technical Reference Manual NUC130 „UM_NUC130_140.pdf“, Seite 175 ff.

Registerübersicht GPIOE

GPIOE_PMD	GP_BA+0x100	R/W	GPIO Port E Pin I/O Mode Control	0xFFFF_FFFF
GPIOE_OFFD	GP_BA+0x104	R/W	GPIO Port E Pin OFF Digital Enable	0x0000_0000
GPIOE_DOUT	GP_BA+0x108	R/W	GPIO Port E Data Output Value	0x0000_FFFF
GPIOE_DMASK	GP_BA+0x10C	R/W	GPIO Port E Data Output Write Mask	0x0000_0000
GPIOE_PIN	GP_BA+0x110	R	GPIO Port E Pin Value	0x0000_XXXX
GPIOE_DBEN	GP_BA+0x114	R/W	GPIO Port E De-bounce Enable	0x0000_0000
GPIOE_IMD	GP_BA+0x118	R/W	GPIO Port E Interrupt Mode Control	0x0000_0000
GPIOE_IEN	GP_BA+0x11C	R/W	GPIO Port E Interrupt Enable	0x0000_0000
GPIOE_ISRC	GP_BA+0x120	R/W	GPIO Port E Interrupt Source Flag	0XXXXX_XXXX

Quelle: Technical Reference Manual NUC130

Einstellen des GPIO-Modes

- Jeder der 16 Pins verfügt über 4 Modi, welche über das Register GPIOx_PMD programmiert werden können. Für jeden Pin sind 2 Bit darin vorhanden.
- Für GPIOE (LEDs und Joystick des Labor-Boards) wird jede 2-Bit-Gruppe über „GPIOE->PMD.PMDx“ angesprochen.
- Modi:
 - 00: Input
 - 01: Output
 - 10: Open-Drain (wird im Labor nicht benötigt)
 - 11: „Quasi-Bidirektional“ (Zustand nach Reset, nicht benötigt)

PMD-Bits

31	30	29	28	27	26	25	24
PMD15		PMD14		PMD13		PMD12	
23	22	21	20	19	18	17	16
PMD11		PMD10		PMD9		PMD8	
15	14	13	12	11	10	9	8
PMD7		PMD6		PMD5		PMD4	
7	6	5	4	3	2	1	0
PMD3		PMD2		PMD1		PMD0	

Quelle: Technical Reference Manual NUC130

Ausgabe von Daten auf GPIO-Pins

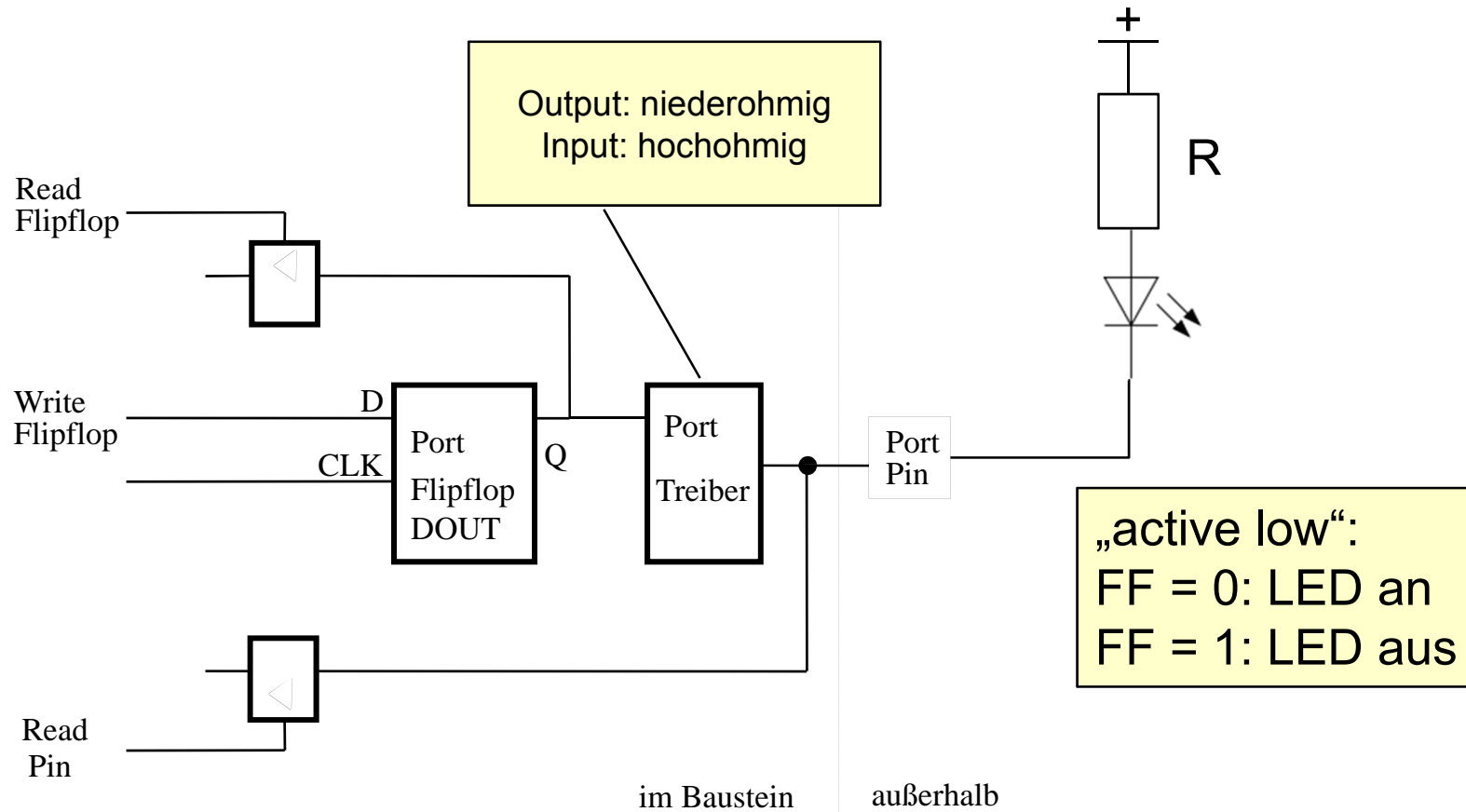
- Die Datenausgabe erfolgt über das Register GPIOx_DOUT, bzw. in C über die Datenstruktur (für GPIOE) „GPIOE->DOUT“
- Die unteren 16 Bit des Registers werden entsprechend auf den Pins 0 bis 15 ausgegeben. Die oberen 16 Bit sind ohne Funktion („Reserved“).
- Mit „GPIOE->DOUT_BYTE0“ und „GPIOE->DOUT_BYTE1“ können die beiden Bytes einzeln ausgegeben werden.
- Die Register-Bits können auch gelesen werden, dabei wird allerdings NICHT der Wert des Pins gelesen!

DOUT-Bits

31	30	29	28	27	26	25	24
Reserved							
23	22	21	20	19	18	17	16
Reserved							
15	14	13	12	11	10	9	8
DOUT[15:8]							
7	6	5	4	3	2	1	0
DOUT[7:0]							

Quelle: Technical Reference Manual NUC130

Schreiben und Lesen des Port-Flipflops



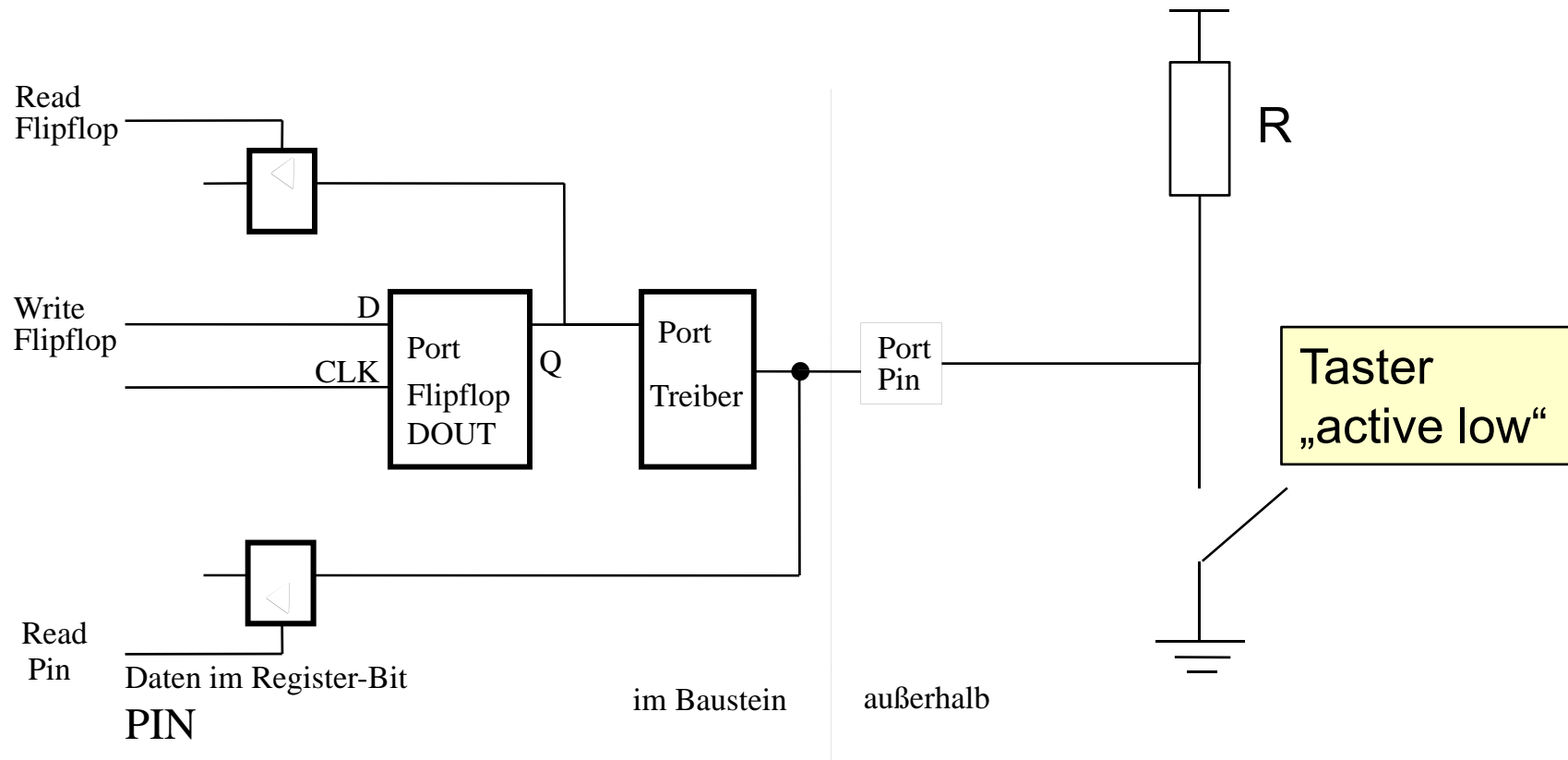
Einlesen von Daten von GPIO-Pins

- Das Einlesen von an den Pins anliegenden Daten erfolgt über das Register GPIOx_PIN, d.h. „GPIOE->PIN“.
- Nur die unteren 16 Bit sind wieder relevant.
- Mit „GPIOE->PIN_BYTE0“ und „GPIOE->PIN_BYTE1“ können die Bytes einzeln eingelesen werden.

PIN-Bits

31	30	29	28	27	26	25	24
Reserved							
23	22	21	20	19	18	17	16
Reserved							
15	14	13	12	11	10	9	8
PIN[15:8]							
7	6	5	4	3	2	1	0
PIN[7:0]							

Lesen des Pins











Übersicht GPIOE-Beschaltung

Bits	15	14	13	12	11	10	9	8
LED-Zeile	LED7	LED6	LED5	LED4	LED3	LED2	LED1	LED0
Mode	PMD15	PMD14	PMD13	PMD12	PMD11	PMD10	PMD9	PMD8
Wert	01	01	01	01	01	01	01	01
DOUT / PIN BYTE1	15	14	13	12	11	10	9	8

Bits	7	6	5	4	3	2	1	0
Joystick*	DOWN	TAST	-	L	R	UP	-	-
Mode	PMD7	PMD6	PMD5	PMD4	PMD3	PMD2	PMD1	PMD0
Wert	00	00	00	00	00	00	00	00
DOUT / PIN BYTE0	7	6	5	4	3	2	1	0

*: Bei Orientierung des Boards mit LED-Zeile links unten

Beispiel

LED-Zeile	i
	1
	2
	4
	8
	16
	32
	64
	128

Achtung: LEDs
sind „low-aktiv“!

```
#include "BoardConfig.h"
#include "Driver_M_Dongle.h"
#include "init.h"

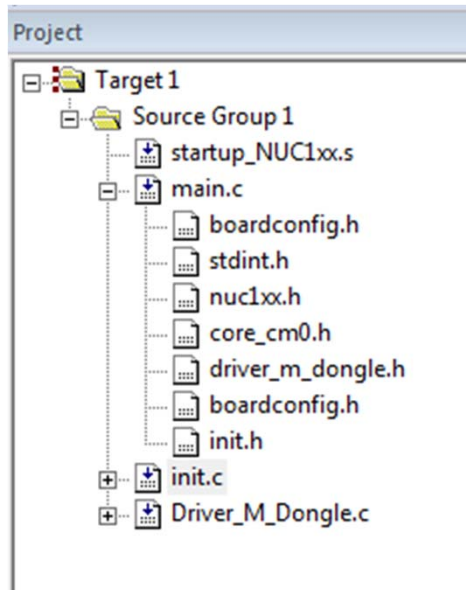
void delay(uint32_t delayCnt){
    while(delayCnt--){}
}

//Main called by startup_NUC1xx.s
int main (void){
    uint8_t i = 1;

    DrvSystem_ClkInit();    //Setup clk system
    Board_Init();           //Initialize GPIOs

    while(1) {
        GPIOE->DOUT_BYTE1 = ~i;    //Next LED on
        delay(1000000);             //wait
        GPIOE->DOUT_BYTE1 = 0xFF;  //switch off
        delay(1000000);             //wait
        if(i == 128) i = 1;
        else i = i*2;
    }
}
```

µVision-Projekt für das Beispiel



„Driver_M_Dongle.c“ wird zum Projekt hinzugefügt. Diese Datei und die Header-Dateien sind in einem Verzeichnis (`_Driver`) abgelegt, welches für jedes Projekt wiederverwendet wird. Dieses Verzeichnis kann in den Compiler-Einstellungen bekannt gemacht werden (Include-Paths), dann müssen die Pfade der Header-Dateien nicht angegeben werden.

Ferner ist noch die Datei „init.c“ aus dem Projektverzeichnis hinzuzufügen. Darin befindet sich die Initialisierungsfunktion „Board_Init()“.

Ausschnitt BoardConfig.h

```
// Definitionen für Ports, benutzen Zuordnungen aus dem ENUM E_DRVGPIO_PORT
// Notwendig für die DrvXXX_ Funktionen, die Ports und Bits bearbeiten
#define PORT_LEDS          E_GPE           // Port E
#define PORT_JOYSTICK      E_GPE           // Port E
#define PORT_BACKLIGHT     E_GPC           // Port C
#define PORT_LCD            E_GPD           // Port D
#define P_TASTER_SW        E_GPA           // Port A

// Definitionen, die einen Portpin innerhalb eines Ports festlegen
// Notwendig für die DrvXXX_ Funktionen, die Ports und Bits bearbeiten

// LED-Zeile
#define LED0                8
#define LED1                9
#define LED2               10
#define LED3               11
#define LED4               12
#define LED5               13
#define LED6               14
#define LED7               15
...
// Pins des Joysticks auf Port E
#define JOY_UP              2             // 0x04
#define JOY_DOWN            7             // 0x80
#define JOY_L               4             // 0x10
#define JOY_R               3             // 0x08
#define JOY_TAST            6             // 0x40
```

Achtung: Wenn der Joystick bewegt wird, wird eine 0 am jeweiligen Pin erzeugt, ansonsten liegt an den Pins jeweils eine 1 an.

Funktion „Board_Init“

```
void Board_Init(void)
{

    //Switch LED GPIO pins to output mode
    DrvGPIO_PortOpen(PORT_LEDS, LED0, E_IO_OUTPUT);
    DrvGPIO_PortOpen(PORT_LEDS, LED1, E_IO_OUTPUT);
    DrvGPIO_PortOpen(PORT_LEDS, LED2, E_IO_OUTPUT);
    DrvGPIO_PortOpen(PORT_LEDS, LED3, E_IO_OUTPUT);
    DrvGPIO_PortOpen(PORT_LEDS, LED4, E_IO_OUTPUT);
    DrvGPIO_PortOpen(PORT_LEDS, LED5, E_IO_OUTPUT);
    DrvGPIO_PortOpen(PORT_LEDS, LED6, E_IO_OUTPUT);
    DrvGPIO_PortOpen(PORT_LEDS, LED7, E_IO_OUTPUT);
    GPIOE->DOUT_BYTE1 = 0xFF; //switch all off (active low)

    //Switch Joystick pins to input mode
    DrvGPIO_PortOpen(PORT_JOYSTICK, JOY_UP, E_IO_INPUT);
    DrvGPIO_PortOpen(PORT_JOYSTICK, JOY_DOWN, E_IO_INPUT);
    DrvGPIO_PortOpen(PORT_JOYSTICK, JOY_L, E_IO_INPUT);
    DrvGPIO_PortOpen(PORT_JOYSTICK, JOY_R, E_IO_INPUT);
    DrvGPIO_PortOpen(PORT_JOYSTICK, JOY_TAST, E_IO_INPUT);

}
```

Nutzung von Treiberfunktionen

- Um die Programmierung zu erleichtern, bieten die Hersteller i.d.R. Treiberfunktionen (engl.: Driver) für die Peripherieeinheiten und das System an.
- Bei Nuvoton im „Board Support Package“. Die Funktionen sind in „NuMicro NUC100 Series Driver Reference Guide.pdf“ dokumentiert.
- Eine Auswahl von Treiberfunktionen und Makros sind in „Driver_M_Dongle.c/.h“ vorhanden.

Beispiel:

```
#include "BoardConfig.h"
#include "Driver_M_Dongle.h"
#include "init.h"

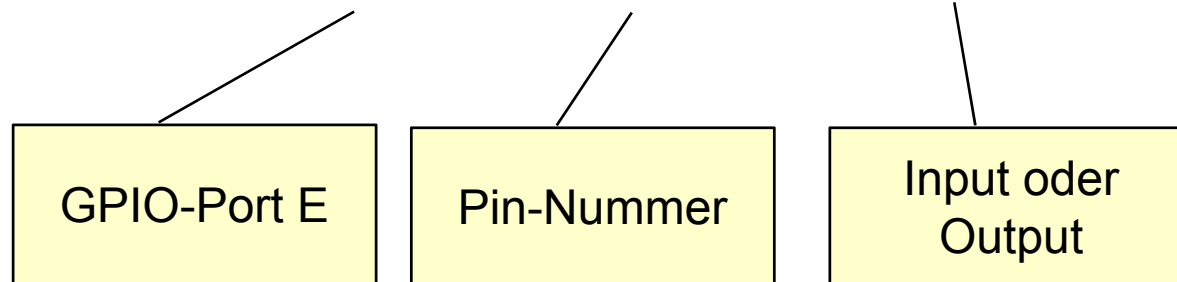
int main (void){
    uint32_t i;

    DrvSystem_ClkInit();    //Setup clk system
    Board_Init();           //Initialize GPIOs

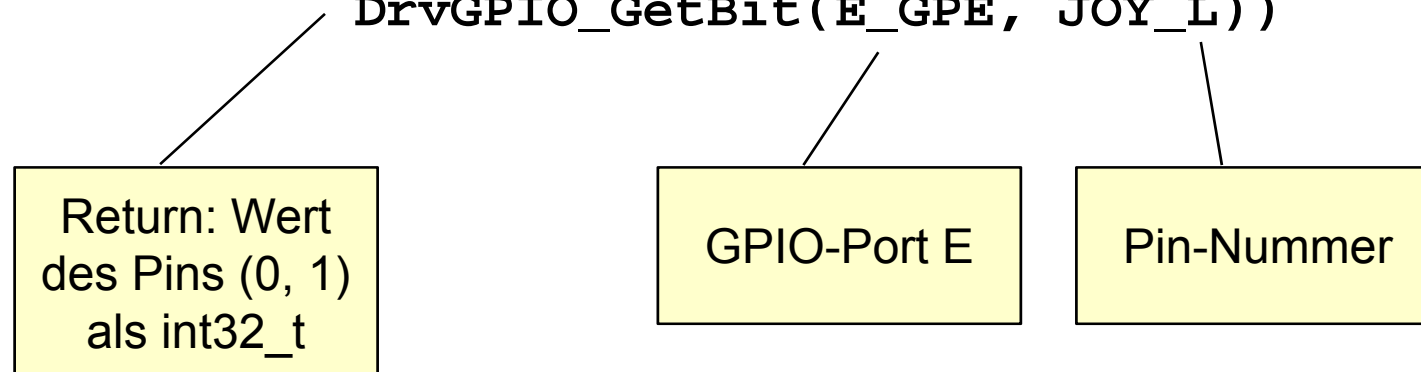
    while(1) {
        if(!DrvGPIO_GetBit(E_GPE, JOY_L)){
            GPIOE->DOUT_BYTE1 = 0xFF;        //all LEDs off
            for(i=LED0; i<LED7+1; i++){
                DrvGPIO_ClearBit(E_GPE, i);    //Clear Bit, LED i on
                DrvSystem_Wait_us(50000);       //wait
                DrvGPIO_SetBit(E_GPE, i);       //Set Bit, LED i off
                DrvSystem_Wait_us(50000);       //wait
            }
        }
        if(!DrvGPIO_GetBit(E_GPE, JOY_R)){
            GPIOE->DOUT_BYTE1 = 0xFF;        //all LEDs off
            for(i=LED7; i>LED0-1; i--){
                DrvGPIO_ClearBit(E_GPE, i);    //Clear Bit, LED i on
                DrvSystem_Wait_us(50000);       //wait
                DrvGPIO_SetBit(E_GPE, i);       //Set Bit, LED i off
                DrvSystem_Wait_us(50000);       //wait
            }
        }
        GPIOE->DOUT_BYTE1 = 0x00;            //all LEDs on
    }
}
```

Benutzte Treiberfunktionen

`DrvGPIO_PortOpen(PORT_JOYSTICK, JOY_UP, E_IO_INPUT)`

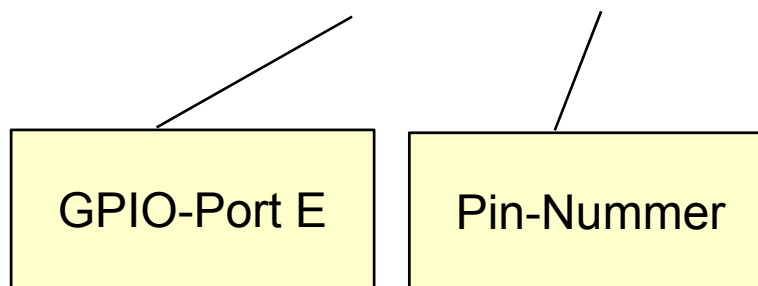


`DrvGPIO_GetBit(E_GPE, JOY_L)`

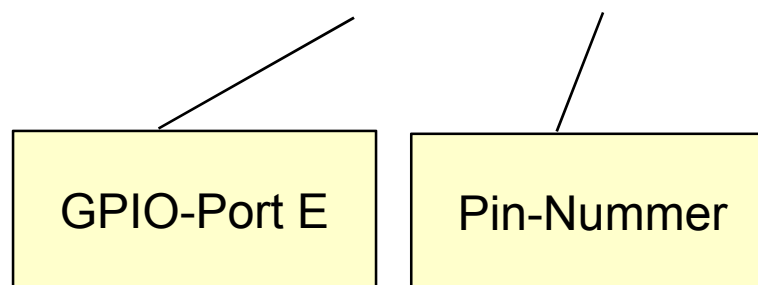


Benutzte Treiberfunktionen (2)

```
DrvGPIO_SetBit(E_GPE, i);
```



```
DrvGPIO_ClearBit(E_GPE, i);
```



Kapitelübersicht

- I. Vom Quellcode zum Programm
- II. Vom Reset zum Hauptprogramm
- III. Zugriff auf Peripherieregister in C
- IV. CMSIS
- V. Programmieretechniken**

Der C-Präprozessor

- Der Präprozessor bearbeitet vor der eigentlichen Kompilation den Quelltext. Er hat im Wesentlichen folgende Aufgaben:
 - Entfernen von „\“ (Backslash-Zeichen) und verbinden der dadurch getrennten Zeilen im Quellcode.
 - Definition und Einsetzen von Makros
 - Bedingte Übersetzung
 - Einkopieren von Dateien (i.A. Header-Dateien)
- Die verschiedenen Funktionen werden über Direktiven gesteuert. Die Direktiven beginnen im Quelltext immer mit dem „#“-Zeichen.
- Der C-Präprozessor bearbeitet auch Assembler-Quellcode, so dass beispielsweise Header-Dateien auch für den ASM-Quellcode benutzt werden können.

Übersicht Präprozessor-Direktiven

#pragma	Einfügen von Compiler-Direktiven, die sonst auf der Kommandozeile eingegeben werden
#include	Fügt eine Datei an dieser Stelle ein
#define	Definition von Makros
#undef	Löschen einer bestehenden Makro-Definition
#ifdef	Abfrage, ob Bezeichner schon definiert ist
#ifndef	Abfrage, ob Bezeichner noch nicht definiert ist
#if	Abfrage, ob Ausdruck das Ergebnis TRUE (≠0) liefert
#elif	Abfrage, wenn vorangegangene if/elif FALSE (=0) waren.
#else	Abfrage ohne Bedingung, wenn vorangegangene if/elif FALSE (=0) waren.
#error #warning #message	Generieren von Fehlermeldungen und Warnungen
#line	Einfügen einer Zeilennummer

Includes

- An der Stelle der Direktive wird die angegebene Datei eingebunden. Dies sind typischerweise „Header-Dateien“, die Variablen, Bezeichner, Makros etc. definieren, die dann in verschiedenen Quelldateien benutzt werden können.
 - Achtung: Variablen dürfen nur einmal definiert werden, sie können aber beliebig oft deklariert werden („extern“). Daher in Header-Dateien, die für mehrere Quellcode-Dateien verwendet werden, Variablen nur deklarieren.
- Kann folgendermaßen angegeben werden:
 - `#include "C:\Design\project.h"`
 - `#include "project.h" //Projektverzeichnis`
 - `#include <global.h> //System- und Projektverzeichnis`
- Wenn keine Verzeichnispfade angegeben werden, sucht der Compiler im aktuellen Arbeitsverzeichnis und im Installationsverzeichnis der KEIL-Software. Weitere Verzeichnisse können in den Einstellungen des Compilers angegeben werden (include paths).
- Verschachtelte „Includes“ sind möglich.

Defines

- Im einfachsten Fall ersetzt ein Makro einen Bezeichner durch einen Text.
 - Für Bezeichner sollte Großschreibung verwendet werden.
 - Sollte beim Programmieren ausgiebig benutzt werden, wegen Änderungsfreundlichkeit (siehe voriges Projekt)
 - Angabe von numerischen Werten im Programm nicht sinnvoll, dafür sollte ein Bezeichner (auch: Symbol) benutzt werden
- Makros können auch mit einer Parameterliste angegeben werden, wobei dann beim Aufruf des Makros die aktuellen Parameter die formalen Parameter der Definition ersetzen.
 - Wir werden parametrisierte Makros hier nicht behandeln. Siehe Literatur.

Bedingte Übersetzung

- Mit Hilfe von `#ifdef`, `#ifndef`, `#if`, `#elif`, `#else` und `#endif` kann der Compiler Teile des Codes in Abhängigkeit von Bedingungen übersetzen.
 - Häufig sollen bestimmte Codeteile, z.B. Ausgaben auf Display oder UART, nur in einer Debug-Version ausgeführt werden.
- Bei `#ifdef` oder `#ifndef` wird abgefragt, ob der Bezeichner definiert ist.
- Bei `#if`, `#elif` wird ein Ausdruck (z.B. Vergleich) oder eine Funktion (z.B. `defined`) ausgewertet, die den Wert `TRUE` (`≠0`) oder `FALSE` (`=0`) liefert.

Bedingte Übersetzung: Beispiel

```
#define TRUE 1
#define FALSE 0
#define DEBUG FALSE

.....
while (1) {
    #if DEBUG == TRUE
        GPIOE->DOUT_B1 = 0x00;
    #else
        GPIOE->DOUT_B1 = GPIOE->DOUT_B0;;
    #endif
}
```

Verhindern von Mehrfacheinbindungen

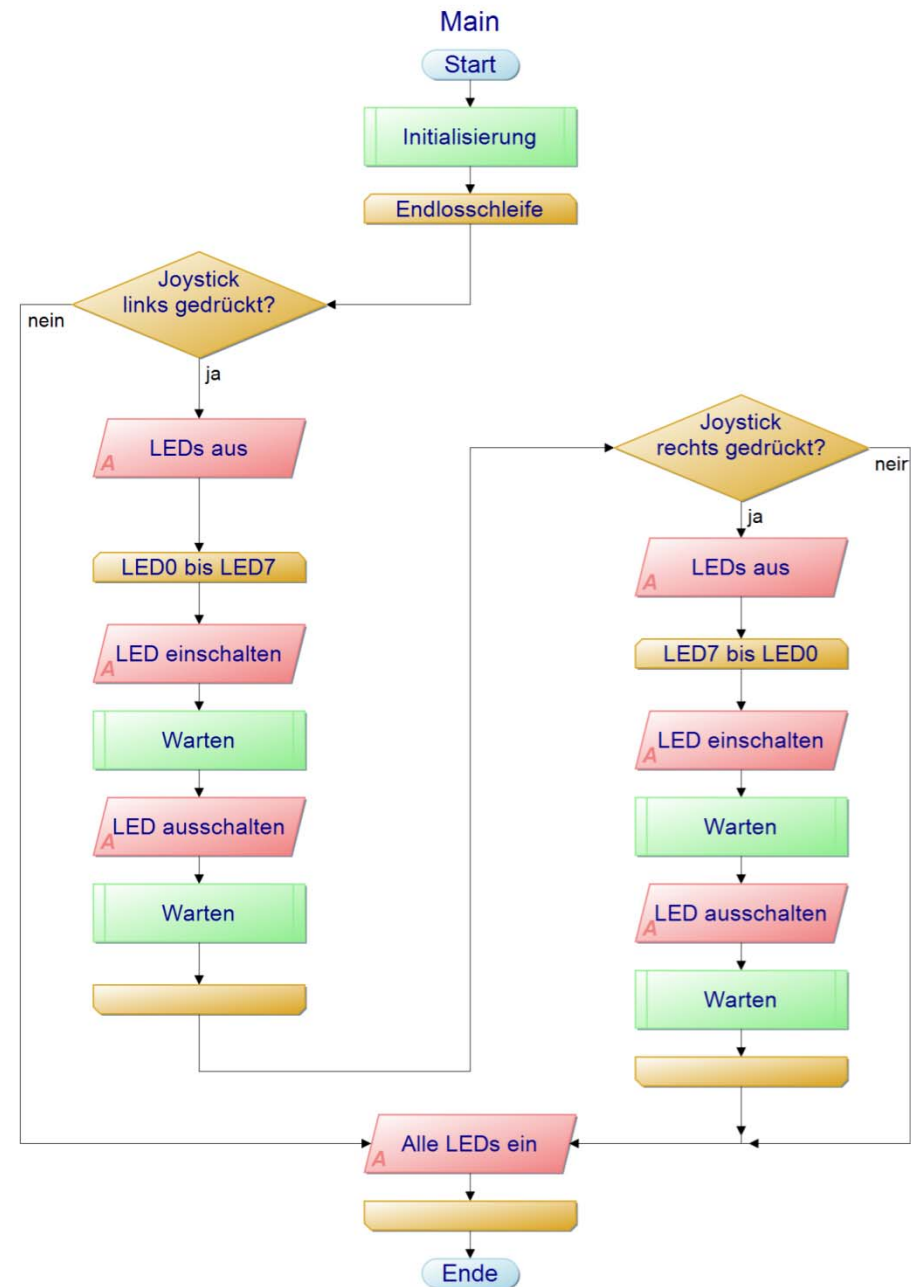
```
#if !defined boardconfig
#define boardconfig

...
// LED-Zeile
#define LED0      8
#define LED1      9
#define LED2     10
#define LED3     11
#define LED4     12
#define LED5     13
#define LED6     14
#define LED7     15

...
#endif
```

Programmablaufpläne

- Vor der Codierung sollte die Aufgabe mit Hilfe von Programmablaufplänen (auch: Flussdiagramm) analysiert werden.
- Tools wie „PapDesigner“ erleichtern die Erstellung.
- Im Labor bitte für jede Aufgabe zunächst einen PAP mit PapDesigner erstellen!

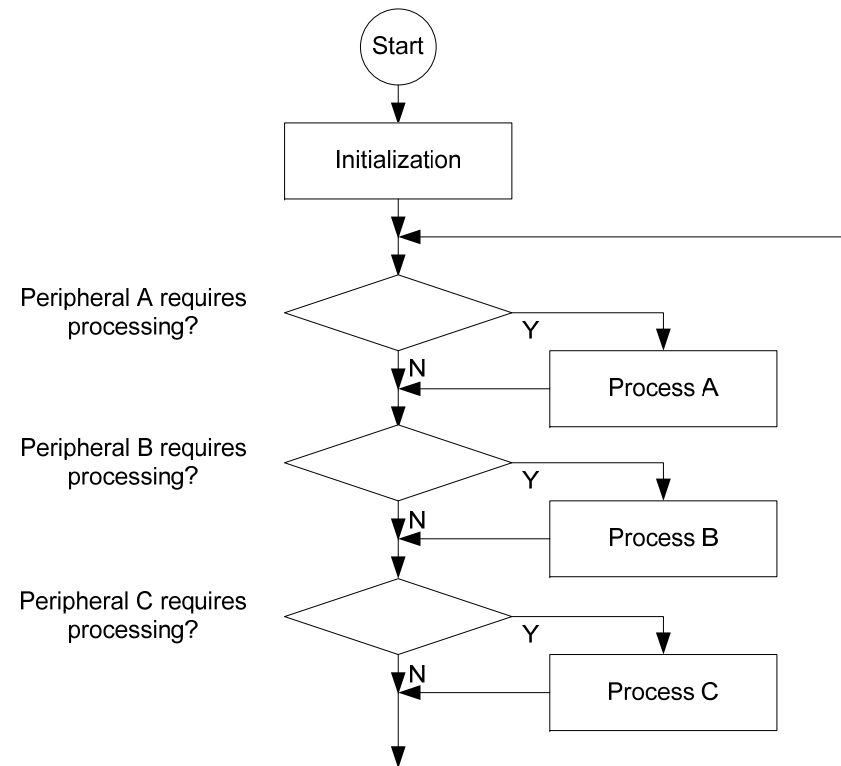


Strukturierung des Codes

- In C spielt die Strukturierung des Codes mit Hilfe von Funktionen eine wesentliche Rolle:
 - Wiederkehrende Abläufe
 - Gliederung des Codes
 - Aber: Keine zu starke Gliederung vornehmen (30-60 Zeilen pro Funktion)
 - Funktion „main“ muss immer vorhanden sein und wird nach Initialisierung ausgeführt.
- Wir verwenden kein Betriebssystem! Die Main-Funktion ist sozusagen das Betriebssystem.
- Wie kann man daher den Ablauf des Programms gestalten?

Polling (Super Loop)

- Bei μ Cs geht es in erster Linie darum, die Peripherieeinheiten zu bedienen oder auf Ereignisse in den PEs zu reagieren.
- Die einfachste Möglichkeit ist die zyklische Abfrage (= „Polling“) aller PEs in einer Endlosschleife im Main-Programm (so genannte „Super Loop“) und die Ausführung der nötigen Funktionen

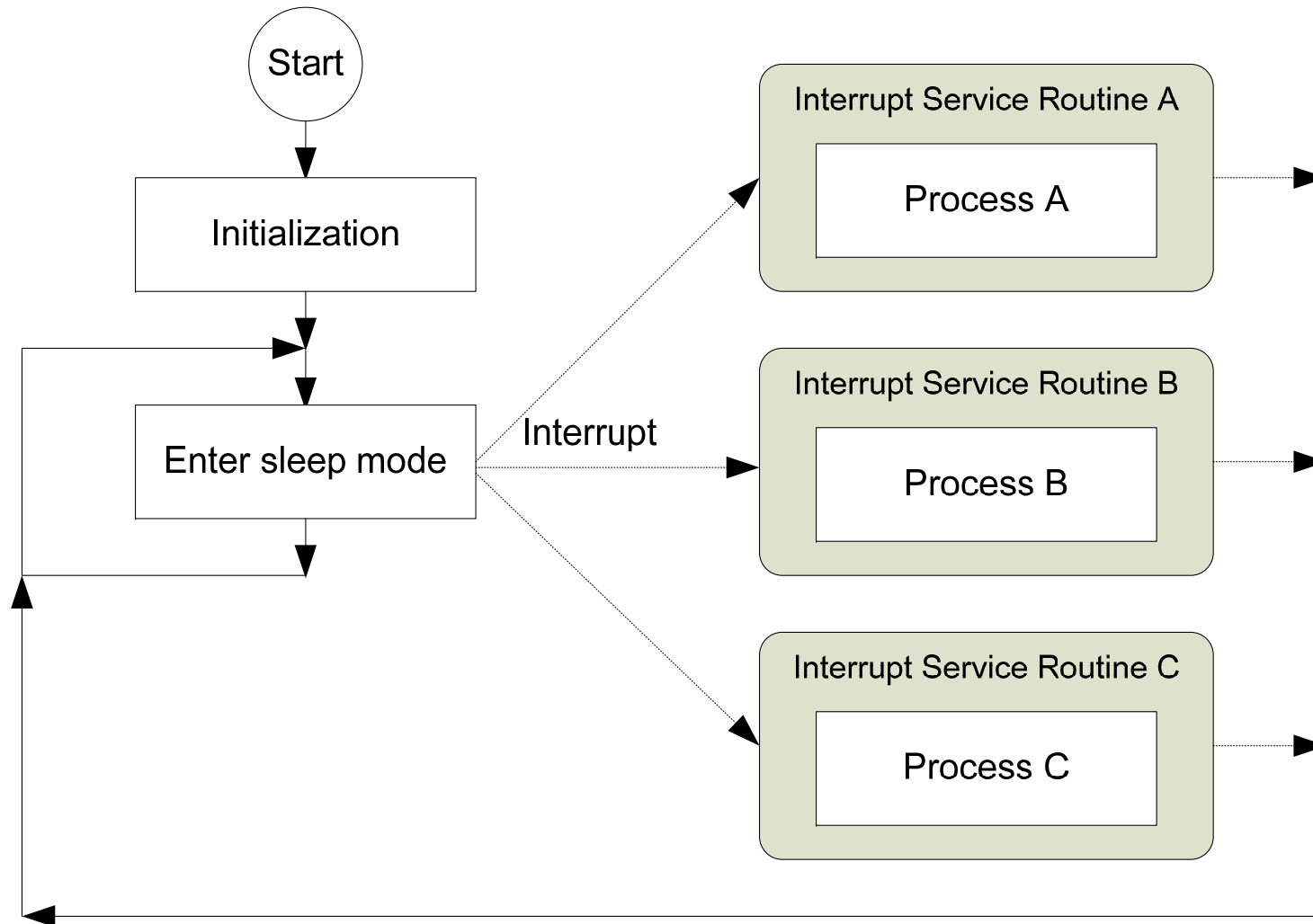


Bildquelle: Joseph Yiu, The Definitive Guide to the ARM Cortex-M0

Verwendung von Interrupts

- Bei μ C-Anwendungen gibt es zeitkritische Vorgänge (z.B. Zündung des Airbags), die sofort bearbeitet werden müssen.
- Ein so genannter Interrupt (oder Exception) kann von der PE selbst ausgelöst werden und führt, bei entsprechender Priorisierung, zu einer schnellen Ausführung des entsprechenden Interrupt Handlers (auch: Interrupt Service Routine)
- Auf der anderen Seite müssen gewisse Vorgänge in einer bestimmten Reihenfolge vorgenommen werden, was durch ausschließliche Verwendung von Interrupts nur schwer zu bestimmen ist.
- In der Praxis mischt man daher häufig Polling und Interrupts.

Verwendung von Interrupts



Bildquelle: Joseph Yiu, The Definitive Guide to the ARM Cortex-M0