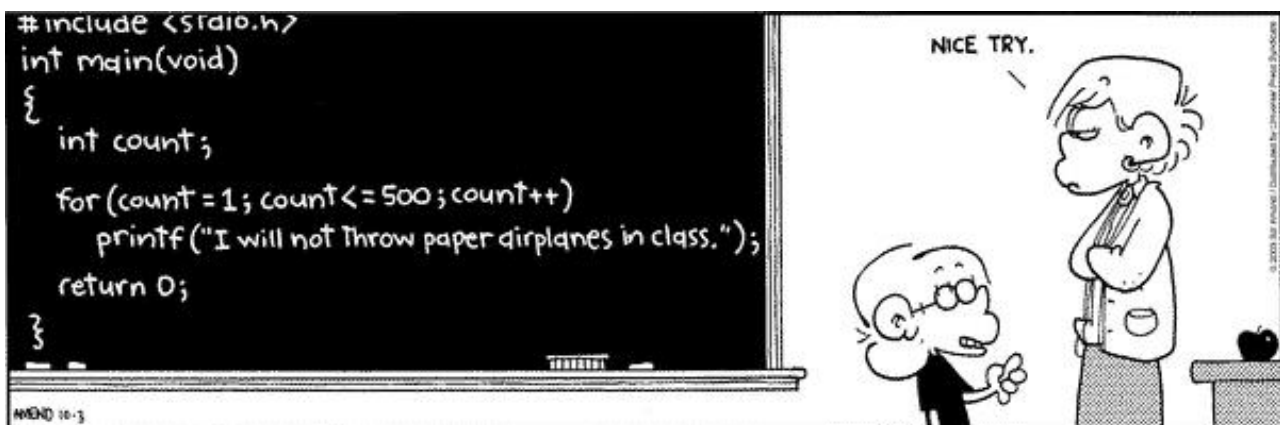
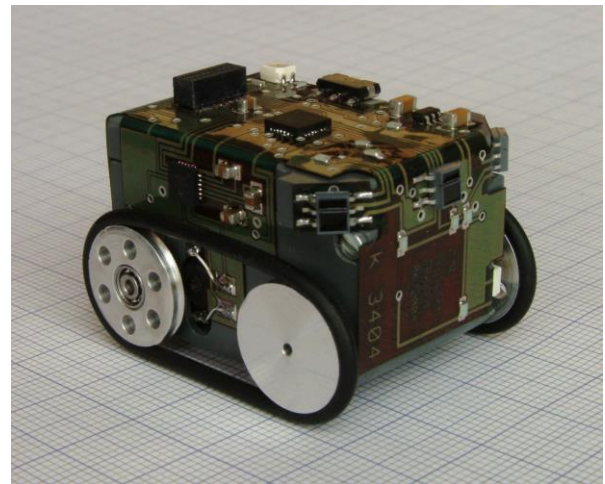
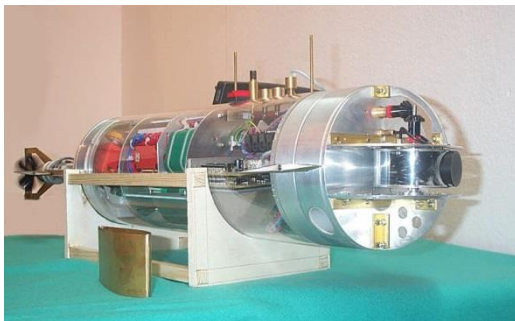


**Einführung in die C Programmierung für Mikrocontroller (Kapitel 1 – 9)**

**Schwerpunkt STM32 (ab Kapitel 10)**

**Prof. Dr. -Ing. Otto Parzhuber**

**Hochschule München FK 06**



Version vom 15.11.2018

## Inhaltsverzeichnis

1.	Erste Schritte in C .....	5
1.1	Prinzipieller Grundaufbau eines einfachen C-Programmes.....	5
2	Übersicht: zulässige Datentypen in C-Programmen .....	7
2.1	Besonderheiten bei ganzzahligen Datentypen .....	8
2.2	An welcher Stelle werden die Variablen deklariert? .....	8
3	Die Zuweisung von Werten bei Variablen.....	9
3.1	Wann werden Variablen im Dezimalformat, wann im Hexadezimalformat angegeben?....	9
4	Operatoren.....	10
4.1	Arithmetische Operatoren .....	10
4.2	Vergleichsoperatoren .....	10
4.3	Verschiebe Operatoren.....	11
4.4	Logische Operatoren .....	11
4.5	Bit Operatoren.....	11
4.6	Beispiel Binär zu BCD Wandlung .....	13
5	Schleifen.....	15
5.1	Die "for" - Schleife .....	15
5.2	Die "while"-Schleife .....	16
5.3	Die "do while" - Schleife .....	18
5.4	Bitverknüpfungen innerhalb von Schleifenabfragen .....	18
5.5	Negierung innerhalb von Schleifenabfragen.....	19
6	Auswahl .....	20
6.1	Die "if" - Anweisung .....	20
6.2	Die "if - else " - Anweisung.....	20
6.3	Die "switch" - Anweisung .....	21
7	C – Anwendungsbeispiel (Alarmanlage) .....	22
8	C – Funktionen.....	23
9	Zusammengesetzte Datentypen.....	24
9.1	Felder .....	24
9.2	Strukturen.....	25
9.3	Schreiben von Gleitkommazahlen in Strings.....	26
10	STM32 spezifische Programmierung.....	27
10.1	Zugriff auf Register des STM32 Microcontrollers .....	27
10.2	Wie kann in der Sprache C auf diese Register zugegriffen werden?.....	27
10.2.1	Ein Zeiger, was ist das?.....	28
10.3	Registerzugriff in der Sprache C in der Praxis .....	29
10.3.1	Ein Zeiger auf eine Struktur – CMSIS Standard- .....	29
10.3.1.1	Beispiel1: Toggeln des 1. Bits von PortA:.....	30
10.3.1.2	Beispiel 2: Ausgabe des Prozessortaktes an PortA Pin 8 (GPIOA.8):.....	30
10.3.1.3	Beispiel 3: Einstellen des Timer 6 .....	31
10.4	Hardware Abstraction Layer (CubeMX-HAL).....	32
10.4.1	Prinzip Software OpenSTM.....	33
10.4.2	Beispiel Takteinstellung.....	34
10.5	Interrupts .....	36
10.5.1	Beispiel Interrupt bei Timer3 Überlauf.....	38
10.5.2	Externe Interrupts.....	41

10.5.3	Polled bzw. Vectored Interrupts.....	42
10.6	GPIO-Ports.....	43
10.6.1	Dateneingabe mit Schalter .....	44
10.6.2	Datenausgabe mit LED .....	46
10.6.3	Treiberschaltungen für LEDs.....	49
10.6.4	Vergleich Open-Drain oder Push-Pull Ausgang.....	50
10.7	Timer.....	51
10.7.1	SysTickTimer.....	51
10.7.1.1	Wartefunktion ohne Cube-HAL.....	52
10.7.1.2	Wartefunktion mit Cube-HAL .....	52
10.7.2	General Purpose Timer .....	53
10.7.2.1	Taktquelle.....	53
10.7.2.2	Basis .....	53
10.7.2.3	Ohne Cube-HAL .....	55
10.7.2.4	Mit Cube-HAL.....	55
10.7.2.5	Input-Capture .....	57
10.7.2.6	Output-Compare (PWM) .....	61
10.7.2.7	Encoder-Signale auswerten.....	64
10.8	UART.....	66
10.8.1	Senden mit Cube-HAL.....	67
10.8.2	Empfangen mit Cube-HAL, interruptgesteuert.....	69
10.9	I2C-Bus .....	71
10.9.1	I2C-Master mit der HAL-Bibliothek .....	71
10.9.1.1	Senden.....	74
10.9.1.2	Empfangen .....	74
10.10	SPI-Bus .....	76
10.10.1	SPI-Master mit der HAL-Bibliothek.....	76
10.10.1.1	Pins einstellen .....	77
10.10.1.2	SPI-Bus konfigurieren.....	78
10.10.1.3	Senden und Empfangen .....	79
10.11	CAN-Bus.....	80
10.11.1	Konfigurieren .....	80
10.11.2	Schreiben von Nachrichten .....	82
10.11.3	Akzeptanzfilterung von Nachrichten .....	84
10.11.3.1	Mask-Mode .....	85
10.11.3.2	IDList-Mode.....	86
10.11.4	Empfangen von Nachrichten - polling .....	88
10.11.5	Empfangen von Nachrichten – interruptgesteuert .....	89
10.12	Analoge Schnittstellen .....	90
10.12.1	ADC-Grundlagen .....	90
10.12.2	ADC- Implementierung im Mikrocontroller.....	92
10.12.3	DAC-Grundlagrn .....	94
10.12.4	DAC-Implementierung im Mikrocontroller.....	94
10.13	RTOS am Beispiel FreeRTOS .....	96
10.13.1	Headerdateien für die Konfiguration .....	98
10.13.1.1	FreeRTOSConfig.h .....	98
10.13.1.2	Hook Funktionen.....	99
10.13.1.3	RunTime Statistik .....	100
10.13.1.4	Queue .....	101
10.14	Debuggingverfahren.....	104
10.14.1	Crossdebugger.....	104

10.14.2	Simulator.....	104
10.15	Profiling .....	104
11	State Machine.....	107
11.1.1	Beispiel Ampelsteuerung .....	107
11.1.2	Realisierung in C.....	110
12	IoT (Internet of Things) .....	113
12.1	MQTT Protokoll .....	113
12.1.1	CC3100-MQTT Projekt .....	113
13	Anhang.....	116
13.1	Wieso Murphy die Sprache C liebt .....	116
13.1.1	Speicherüberlauf von Arrays .....	116
13.1.2	Stacküberlauf .....	116
13.1.2.1	Wie man einen Stacküberlauf feststellt:.....	116
13.1.3	Nullpointer .....	117
13.1.4	Interrupts .....	117
13.1.5	Dynamisch Speicher allozieren.....	117
13.1.6	Race Conditions .....	118
13.1.7	Reentrant Code.....	118
13.1.8	Defensive Programmierung .....	119
13.1.9	assert().....	119
13.1.10	Optimierter C-Code.....	119
13.2	Symbole für Programmabläufe .....	121
13.3	Anschlussplan Nucleo-F072RB .....	122

## 1. Erste Schritte in C

In dieser Anleitung werden diese Grundlagen am Beispiel der Programmierung mit dem Mikrocontroller STM32 erklärt.

### 1.1 Prinzipieller Grundaufbau eines einfachen C-Programmes

Den prinzipiellen Aufbau eines C –Programms zeigt das folgende Codeschnippel:

```
/** (1)
 * \file      test.c
 * \brief     Gerüst eines C Programms
 * \author    Otto Parzhuber
 * \date      12.12.2012
 *
 */
#include "stm32f0xx.h" (2)
#include "stm32f0xx_nucleo.h" (2)

#define PI 3.1415 (3)

uint32_t wichtige_globale_variable; (4)

int main(void) (5)
{ (6)
    Init(); // Initialisierungen, z.B. GPIO-Ports (7)
    while(1) (8)
    {
        //hier stehen Ihre Anweisungen und Funktionen
    }
    return 0;
}
```

(1)

Am Anfang aller C-Programme steht ein Kommentarkopf mit wichtigen Infos zur der Quelldatei (Autor, Version, kurze Beschreibung etc.).

Anmerkung:

Mit den Bezeichnern „\file, \brief, ...“ können Sie später für eine leistungsfähige html-Dokumentation des Quellcodes das weit verbreitete Dokumentations-Tool Doxygen<sup>1</sup> verwenden.

(2)

Dann folgen die "Header-Dateien". Sie sorgen dafür, dass die verwendeten Variablen und Funktionen dem Compiler bekannt gemacht werden. Dafür stehen in diesen Dateien die Deklarationen und eventuelle Funktionsprototypen. Headerdateien werden mit dieser Syntax am Anfang der Quelldatei eingefügt:

```
#include "stm32f0xx.h" // " . . . " für Dateien im Projektverzeichnis
```

Diese Headerdateien werden für den Mikrocontroller im Praktikum benötigt und ermöglichen den Zugriff auf die Register des Mikrocontrollers.

Wenn Sie Funktionen aus der Standardbibliothek der Sprache C verwenden müssen Sie die Datei **stdio.h** einfügen:

---

<sup>1</sup> Open-Source Dokumentations-Tool für Software, [www.doxygen.org](http://www.doxygen.org)

## Einführung in C für STM32

```
#include <stdio.h>    //< . . . > für Dateien im Systemverzeichnis der
                     // Programmierungsumgebung
```

(3)

Global benötigte Konstanten werden ganz oben in der Datei mit „`#define`“ deklariert. Zur Unterscheidung zu „normalen“ Variablen sollten diese globalen Konstanten mit Großbuchstaben vereinbart werden.

(4)

Von Zeit zu Zeit werden auch globale Variablen benötigt. Diese werden vor der `main` Funktion deklariert.

(5)

Die Hauptfunktion in C lautet ***main***, die Parameter sind üblicherweise:

```
int main(void)
{
    Anweisung;
    Anweisung;
    return 0;
}
```

```
void main(void)
{
    Anweisung;
    Anweisung;
}
```

Die links gezeigte Variante gibt einen Wert mit dem Datentyp ***int*** zurück, die rechte Variante gibt nichts zurück. Abgeschlossen wird eine Anweisung immer mit dem Semikolon! Mehr zu Funktionen später. . .

(6)

Ein Anweisungsblock in C beginnt mit der öffnenden geschweiften Klammer und endet mit der schließenden geschweiften Klammer:

```
{
    Anweisung;
    Anweisung;
}
```

Für die Formatierung können Sie eine der beiden Arten wählen:

```
while(1)
{
    Anweisung;
    Anweisung;
}
```

```
while(1){
    Anweisung;
    Anweisung;
}
```

Die schließende geschweifte Klammer muss unbedingt in derselben Spalte wie die öffnende Klammer sein bzw. in derselben Spalte wie in diesem Beispiel die Kontrollstruktur `while()`. Welche der beiden Formatierung Sie wählen bleibt Ihnen überlassen, aber Sie dürfen diese innerhalb der Datei nicht mischen!

(7)

In der Regel müssen einmalige Initialisierungen vorgenommen werden, diese erfolgen üblicherweise in einer Funktion, in dem Beispiel `Init()` genannt. Mehr zu Funktionen später. . .

**Jede Anweisung in der Sprache C muss mit einem Semikolon beendet werden!!!!**

(8)

Hier wird eine Endlosschleife mit der Kontrollstruktur *while* programmiert (siehe 4.3). Ein Programm ohne ein darunterliegendes Betriebssystem darf niemals beendet werden, deshalb diese Endlosschleife.

Anmerkung:

Das Programm führt stur im Programmspeicher Anweisung für Anweisung aus, wenn der Programmcode zu Ende ist wird der nächste Inhalt im Programmspeicher abgearbeitet. Im besten Fall ist der Speicher gelöscht, dann passiert nichts schlimmes, aber es könnten ja auch unsinnige Werte im Programmspeicher stehen....

## 2 Übersicht: zulässige Datentypen in C-Programmen

Für die allgemeine C-Programmierung gilt der folgende "Vorrat" an Zahlentypen.

Verständliche Bezeichnung	Offizieller Name	Wertebereich	Belegter Speicherplatz
Einzelnes Bit	In C kein Datentyp	0 oder 1	1 Bit
Positive ganze Zahlen von Null bis 255	<b>unsigned char</b>	0 bis 255	1 Byte
Ganze Zahlen von -128 bis +127	<b>(signed) char</b>	-128 bis +127	1 Byte
Positive ganze Zahlen zwischen Null und 65535	<ul style="list-style-type: none"> <li>• <b>unsigned short int</b> oder</li> <li>• <b>unsigned int</b> bei 8 bit Prozessoren</li> </ul>	0 bis 65535	2 Byte
Ganze Zahlen zwischen -32768 und +32767	<ul style="list-style-type: none"> <li>• <b>(signed) short int</b> oder</li> <li>• <b>int</b> bei 8 bit Prozessoren</li> </ul>	-32768 bis +32767	2 Byte
Positive ganze Zahlen zwischen Null und 4 294 967 295	<ul style="list-style-type: none"> <li>• <b>unsigned int</b> oder</li> <li>• <b>unsigned long int</b> bei 8 bit Prozessoren</li> </ul>	0 bis 4 294 967 295	4 Byte
Ganze Zahlen zwischen -2 147 483 648 und +2 147 483 647	<ul style="list-style-type: none"> <li>• <b>(signed) int</b> oder</li> <li>• <b>(signed) long int</b> bei 8 bit Prozessoren</li> </ul>	-2 147 483 648 bis +2 147 483 647	4 Byte
Gleitkommazahlen im Bereich von $10^{-37}$ bis $10^{+37}$ mit 6 Nachkommastellen	<b>float</b>	$10^{-37}$ bis $10^{+37}$	4 Byte
Gleitkommazahlen im Bereich von $10^{-308}$ bis $10^{+308}$ mit 16 Nachkommastellen	<b>double</b>	$10^{-308}$ bis $10^{+308}$	8 Byte

## 2.1 Besonderheiten bei ganzzahligen Datentypen

Da der Wertebereich der ganzzahligen Datentypen abhängig von der Datenbusbreite des Prozessors ist, sollten Sie bei der Programmierung von Mikrocontrollern unbedingt die folgenden Ersatzdatentypen verwenden:

Original	Ersatz
unsigned char	uint8_t
signed char oder char	int8_t
unsigned short	uint16_t
signed short oder short	int16_t
unsigned int	uint32_t
signed int oder int	int32_t
unsigned long int	uint64_t
signed long int oder long int	int64_t

Diese Datentypen sind in der Datei **stdint.h** deklariert. Da diese Datei ein Bestandteil des Compilers für den Mikrocontroller ist, kann durch den Austausch der passenden Datei **stdint.h** problemlos von einem 8-bit Prozessor zu einem 32-bit Prozessor gewechselt werden.

Die beiden Deklarationen sind also identisch:

```
int8_t var1;  
char var1;
```

Die beiden folgenden Deklarationen sind nur bei 8 Bit Prozessoren identisch:

```
uint16_t var2;  
unsigned int var2;
```

Bei 8 Bit Prozessoren ist die Wortbreite für den Datentyp int 16 Bit groß und bei einem 32 Bit Prozessor ist die Wortbreite 32 Bit!

Da die Speicherressourcen eines Mikrocontrollers in der Regel begrenzt sind, sollten immer die kleinstmöglichen Datentypen verwendet werden. (Der im Praktikum verwendete µC LPC17xx z.B. hat 64 KB SRAM, ein kleinerer Bruder LPC3xx hat lediglich 4kB).

## 2.2 An welcher Stelle werden die Variablen deklariert?

Die Deklaration einer Variablen muss immer innerhalb derjenigen Funktion geschehen, in der diese Variable verwendet werden soll. Globale Variable werden nur sehr selten benötigt und erschweren die Fehlersuche ungemein.

Bei Variablennamen dürfen keine Sonderzeichen und Umlaute verwendet werden

Beispiele:

```
int main(void)  
{  
    uint16_t ui;    // Deklaration von ui als Zahl zwischen      0 . . . +65535  
    int16_t si;     // Deklaration von si als Zahl zwischen -35768 . . . +37767  
    uint8_t c;      // Deklaration von c als Zahl zwischen      0 . . .   +255  
    . . .  
}
```



## 3 Die Zuweisung von Werten bei Variablen

### 1. Beispiel: **Zuweisung eines bestimmten Zahlenwertes beim Programmstart ("Initialisierung")**

<code>int8_t strom = 5;</code>	bedeutet, dass für die Variable "strom" der Startwert 5 gespeichert wird (und "int8_t" schreibt die "Ganzzahl-Darstellung mit 8 Bit und positivem oder negativem Vorzeichen" vor).
--------------------------------	--

### 2. Beispiel: **Hochzählen einer Zahl**

<code>summe = summe + 5; oder: summe +=5;</code>	bewirkt, dass zum vorhandenen "summe-Ausgangswert" die Zahl 5 addiert und das Ergebnis wieder unter "summe" gespeichert wird.
--	---

### 3. Beispiel: **Zuweisung eines Rechenergebnisses**

<code>widerstand = spannung / strom;  strom = spannung / widerstand;  spannung = strom * widerstand;</code>	Dies sind Anwendungen des Ohm'schen Gesetzes, bei denen das Rechenergebnis der rechten Seite anschließend in die Variable der linken Seite geschoben wird.
---	--

### 3.1 Wann werden Variablen im Dezimalformat, wann im Hexadezimalformat angegeben?

Zahlen werden in der Regel entweder als dezimal oder als Hexadezimalzahl angegeben. Für die Auswertung im Rechner ist es nicht von Bedeutung, da alle Zahlen in das Binärformat umgewandelt werden.

Die Hexadezimalschreibweise bietet sich an, wenn einzelne Bits in der Variablen manipuliert also entweder gesetzt oder gelöscht werden sollen. In C bedeutet eine Zahl mit **0x** am Anfang eine Hexadezimalzahl. (Dazu später mehr. . .).

Beispiel für Dezimalzahl:

```
strom = 523; // Die Variable strom bekommt den Dezimalwert 523 zugewiesen  
            // Es könnte 523 mA bedeuten, Hexadezimal macht keinen Sinn.
```

Beispiel für Hexadezimalzahl:

```
reg = 0xAA; // Die Variable reg bekommt den Hexadezimalwert 0xAA zugewiesen, dies  
            // entspricht dem Dezimalwert 170  
            // aus der Hexadezimalzahl kann die Bitkombination 1010 1010 leicht  
            // abgelesen werden, aus der Dezimalzahl nicht!
```

## 4 Operatoren

### 4.1 Arithmetische Operatoren

Die Verwendung einiger arithmetischer Operatoren wurde bereits in dem Kapitel 3 gezeigt:

+	bedeutet " <b>Addition</b> "
-	bedeutet " <b>Subtraktion</b> "
*	bedeutet " <b>Multiplikation</b> "
/	bedeutet " <b>Division</b> "
%	bedeutet " <b>Modulo</b> "

### 4.2 Vergleichsoperatoren

In den Prüfbedingungen unserer Schleifen werden immer Vergleiche angestellt. Deshalb werden sie auch "**Vergleichsoperatoren**" genannt.

Folgende Vergleichsoperatoren stehen uns zur Verfügung:

<	bedeutet " <b>kleiner als</b> "
>	bedeutet " <b>größer als</b> "
<=	bedeutet " <b>kleiner oder gleich</b> "
>=	bedeutet " <b>größer oder gleich</b> "
!=	bedeutet " <b>ungleich</b> "
==	bedeutet " <b>gleich</b> "

Aufpassen muss man bei besonders bei der Prüfung auf Gleichheit, also dem Operator „==“ !!

```
if (taster == 1)    // korrekte Anweisung
{
    tu_was();
}

// aber die folgende Abfrage ist eine (Programmier-) Katastrophe

if (taster = 1)     // zuerst wird taster auf 1 gesetzt und anschließend abgefragt
{
    tu_was();
}
```

In Vergleichsabfragen werden die folgenden Operatoren auch sehr häufig verwendet:

## 4.3 Verschiebe Operatoren

>>	bedeutet " <b>bitweise Rechtsverschiebung</b> "
<<	bedeutet " <b>bitweise Linksverschiebung</b> "

## 4.4 Logische Operatoren

&&	bedeutet " <b>logische UND Verknüpfung</b> "
	bedeutet " <b>logische ODER Verknüpfung</b> "
!	bedeutet " <b>Negierung</b> "

## 4.5 Bit Operatoren

&	bedeutet " <b>bitweise UND Verknüpfung</b> "
	bedeutet " <b>bitweise ODER Verknüpfung</b> "
~	bedeutet " <b>bitweise Invertieren</b> "
^	bedeutet " <b>bitweise EXOR Verknüpfung</b> "

Bitoperationen sind insbesondere bei der Mikrocontrollerprogrammierung wichtig. Für die Initialisierung von Peripheriekomponente nwie z.B. eine serielle Schnittstelle oder ein Timer müssen Bits in einem Register gesetzt oder gelöscht werden. Das Beispiel zeigt das Setzen und löschen der Bits 0, 6 und 7 in einem 8-Bit Datenwort:

7	6	5	4	3	2	1	0	Bit
		0	0	0	0	0		

```
uint8_t ui = 0x00;

ui = ui | 0x01;    // setzt das Bit 0, ui = 0000 0001 binär
ui = ui | 0x80;    // setzt das Bit 7, ui = 1000 0001 binär
ui = ui | 0x44;    // setzt die Bits 6 und 2, ui = 1100 0101 binär

ui = ui & 0xFE;    // löscht das Bit 0, ui = 1100 0100 binär
ui = ui & 0x7F;    // löscht das Bit 7, ui = 0100 0100 binär
ui = ui & 0xBB;    // löscht die Bits 6 und 2, ui = 0000 0000 binär

// oder auch in kürzerer Schreibweise, aber ansonsten gleich
ui = 0x00;
ui |= 0x01;    // setzt das Bit 0, ui = 0000 0001 binär
ui |= 0x80;    // setzt das Bit 7, ui = 1000 0001 binär
ui |= 0x44;    // setzt die Bits 6 und 2, ui = 1100 0101 binär

ui &= 0xFE;    // löscht das Bit 0, ui = 1100 0100 binär
```

## Einführung in C für STM32

```
ui &= 0x7F; // löscht das Bit 7, ui = 0100 0100 binär
ui &= 0xBB; // löscht die Bits 6 und 2, ui = 0000 0000 binär

// wenn die zu setzenden und zu löschenden Bits identisch sind ist die folgende
// Schreibweise mit bitweisem invertieren leichter lesbar

ui &= ~0x01; // löscht das Bit 0, ui = 1100 0100 binär
ui &= ~0x80; // löscht das Bit 7, ui = 0100 0100 binär
ui &= ~0x44; // löscht die Bits 6 und 2, ui = 0000 0000 binär
```

In Verbindung mit den beiden Verschiebeoperatoren << und >> können Bitmuster einfach und leichter lesbar erzeugt werden:

```
uint8_t ui = 0x00;

ui = (1<<1); // setzt das Bit 1, ui = 0000 0010 binär
ui |= (1<<7); // setzt das Bit 7, ui = 1000 0001 binär

// diese beiden Zeilen können in eine Zeile geschrieben werden:
ui = (1<<1) | (1<<7);
```

Es können auch komplexere Bitmuster problemlos erzeugt werden. Hier werden in einem 16 bit breiten Datenwort zwei bzw. 3 Bits ab Bit 1 bzw. Bit 9 gesetzt:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Bit
0	0	0	0	1	1	1	0	0	0	0	0	0	1	1	0	

```
uint16_t ui = 0x0000;

ui = (0x3<<1); // setzt ab Bit 1 zwei Bits, ui = 0000 0000 0000 0110 binär
ui |= (0x7<<9); // setzt ab Bit 9 drei Bits, ui = 0000 1110 0000 0110 binär

//bzw.:

ui = (0x3<<1) | (0x7<<9);
```

#### 4.6 Beispiel Binär zu BCD Wandlung

Für verschiedenste Aufgaben kann es notwendig sein, Variablen vom Binärformat in eine BCD<sup>2</sup> Darstellung umzuwandeln. Ein Beispiel ist eine Echtzeituhr bei der die Sekunden, Minuten etc. im BCD Format in bestimmte Register geschrieben werden müssen.

Die Tabelle zeigt den Ausschnitt aus einem Datenblatt der Echtzeituhr RTC 8253 von NXP:

**Table 13. SECONDS coded in BCD format**

Seconds value in decimal	Upper-digit (ten's place)			Digit (unit place)			
	Bit			Bit			
	6	5	4	3	2	1	0
00	0	0	0	0	0	0	0
01	0	0	0	0	0	0	1
02	0	0	0	0	0	1	0
:	:	:	:	:	:	:	:
09	0	0	0	1	0	0	1
10	0	0	1	0	0	0	0
:	:	:	:	:	:	:	:
58	1	0	1	1	0	0	0

Als Beispiel soll die Zahl 58 in das BCD Format gewandelt werden. Das binäre Muster der Zahl lautet 0011 1010 und soll in das binäre Muster 0101 1000 gewandelt werden. Den Schlüssel dafür liefert die Dezimaldarstellung. Jede Dezimalstelle wird bei der BCD Codierung in eine Dualzahl umgewandelt.

Die Zahl dividiert durch 10 liefert die Zehnerstelle, in unserem Fall 5, der Rest ist 8. Den Rest erhält man in C durch den Modulo-Operator (%).

Für die Umwandlung werden diese beiden arithmetischen Operatoren / und % benötigt. Der Code zeigt diese Umwandlung in einer Codezeile zusammengefasst:

```
sec_in_bcd = (((sec_binary / 10) << 4) & 0xF0) | ((sec_binary % 10) & 0x0F);
```

Einfacher zu verstehen ist diese Umwandlung, wenn sie auf mehrere Zeilen aufgeteilt wird:

```
sec_in_bcd_tens = (sec_binary / 10); //!Berechnung Zehnerstelle
```

Nach dem extrahieren der Zehnerstelle durch dividieren durch 10 muss lt. der obigen Tabelle die Zahl um 4 Bit nach links geschoben werden:

```
sec_in_bcd_tens <<= 4;
//oder gleichbedeutend:
sec_in_bcd_tens = sec_in_bcd_tens << 4;
```

<sup>2</sup> Binary Coded Decimals

## Einführung in C für STM32

Es sind jetzt nur die oberen 4 Bits des 8 Bit breiten Wortes relevant, also werden die unteren 4 Bits gelöscht. Dafür kann die Bit weise Konjunktion (&) verwendet werden:

```
sec_in_bcd_tens &= 0xF0; //!oder sec_in_bcd_tens = sec_in_bcd_tens & 0xF0
```

Analog wird mit der Einerstelle verfahren:

```
sec_in_bcd_units = (sec_binary % 10); //!Berechnung der Einerstelle
```

Der Modulo-Operator liefert den ganzzahligen Rest, in dem Beispiel also die Zahl 8. Es sind nur die unteren 4 Bits relevant, also werden die oberen 4 Bits gelöscht:

```
sec_in_bcd_units &= 0xF0; //!oder sec_in_bcd_units = sec_in_bcd_units & 0x0F
```

Der Inhalt der Variablen lautet nun:

sec_in_bcd_tens	0	1	0	1	0	0	0	0
sec_in_bcd_units	0	0	0	0	1	0	0	0

Jetzt müssen die beiden Variablen nur mehr addiert werden. Bei Dualzahlen ohne Übertrag kann das entweder mit der bitweisen Disjunktion erfolgen, oder auch mit dem Additionsoperator.

```
sec_in_bcd = (sec_in_bcd_tens + sec_in_bcd_units);  
// oder:  
sec_in_bcd = (sec_in_bcd_tens | sec_in_bcd_units);
```

Übungsaufgabe:

Umwandlung von BCD in eine binäre Darstellung.

#todo

## 5 Schleifen

Sie dienen zur Wiederholung von Programmteilen. Hierbei gibt es **drei verschiedene Möglichkeiten**:

- a) Eine Bedingung am **Schleifenanfang** erzwingt eine (ganz exakt und genau berechenbare) **Zahl der Wiederholungen**:

for-Schleife

- b) Es wird **nur solange** wiederholt, wie eine am **Schleifenanfang** stehende Bedingung **erfüllt** ist:

while-Schleife

- c) Die Schleife wird prinzipiell jedenfalls einmal durchlaufen. **Am Ende** des **ersten Durchganges** steht eine Prüfbedingung. Durch sie wird die **Zahl der Wiederholungen** festgelegt:

do-while-Schleife

### 5.1 Die "for" - Schleife

Sie steht immer **am Anfang** einer Schleife und **erzwingt solange eine Wiederholung**, bis die in der Klammer auf "for" folgende **Bedingung nicht mehr wahr** ist.

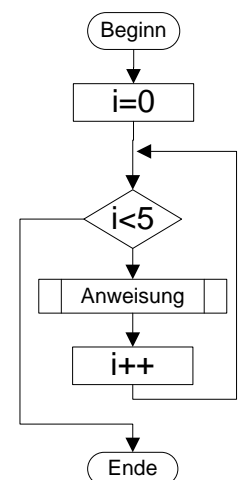
(In der Klammer steht üblicherweise eine Vorschrift, mit der eine Zahl x bis zu einem bestimmten Höchstwert inkrementiert wird).

Das kann für folgende Anwendungen genutzt werden:

- a) Exakt vorgeschriebene Anzahl von Wiederholungen eines Programmteiles.

Beispiel: Die Variable **ausgang** soll 5mal invertiert werden.

```
{
    uint8_t i, ausgang = 0xAA;
    .....
    for(i=0; i<5; i++)    // kein Komma verwenden, sondern
                        // Semikolon!!!!
    {
        ausgang = ~ausgang;
    }
    .....
}
```



b) Programmierung von Warte- und Zeitschleifen.

Beispiel: Programmierung einer Zeitschleife. Der Bezeichner **volatile** ist eine Aufforderung an den Compiler, die Variable nicht zu „weg zu optimieren“.

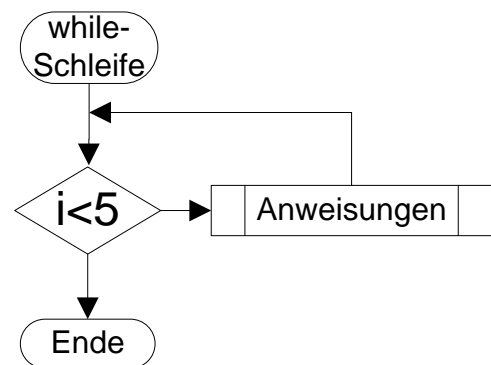
```
{
    volatile uint16_t i;
    .....
    for(i=0; i<=50000; i++);    // Achtung auf Überlauf: uint8_t geht in diesem
                                //Beispiel nicht
    .....
}
```

## 5.2 Die “while”-Schleife

Prinzip

- Wenn die am Schleifenanfang stehende Bedingung nicht gilt, dann wird die gesamte Schleife übersprungen .
- Solange die am Schleifenanfang stehende Bedingung gilt, wird die Schleife wiederholt.

Das ist z.B. eine ideale Methode zur Abfrage von Schaltern, Tasten, Sensoren, . . . , um anschließend einen bestimmten Vorgang auszulösen.



Hinweis: ein wichtiger Sonderfall ist die Formulierung:

```
{
    while(1)                // Endlosschleife Beginn
    {
        Anweisungen;
    }                       // Endlosschleife Ende
    Anweisungen;            // dieser Code wird nie erreicht!!!
}
```

Da die Bedingung in der Klammer immer erfüllt ist, erhält man eine Endlosschleife. Diese Endlosschleifen können mit **break** auch jederzeit wieder verlassen werden:

```
void main (void)
{
    uint8_t i = 0;          // fange mit i=0 an
    while(1)                // Endlosschleife
    {
        if(i<250)
            i++;            // inkrementiere x bis 250
        else
            break;          // verlasse die Schleife
    }
    tu_was_wichtiges();
}
```



1. Beispiel: Solange die Variable “**taste1**” an LOW-Pegel liegt, wird die Variable “**ausgang**” invertiert:

```
{
    .....
    while (taste1==0)
    {
        ausgang =~ ausgang;
    }
    .....
}
```

Schreiben Sie niemals anstelle == nur =  
Die Anweisung (taste=0) würde zuerst zugewiesen und erst nachher die Schleife while() ausgewertet

2. Beispiel: Es wird die Invertierung des Ausgangs um eine Zeitschleife ergänzt und alles in einer Endlosschleife wiederholt:

```
void main (void)
{
    while(1)
    {
        uint8_t i;

        for(i=0; i<250; i++){;;}
        ausgang = ~ausgang;
    }
}
```

3. Beispiel: Das lässt sich auch ausschließlich mit “while”-Schleifen realisieren:

```
void main (void)
{
    uint8_t i;
    while(1)                // Endlosschleife
    {
        i = 0;               // fange mit x=0 an
        while(i<250)         // inkrementiere x bis 250
        {
            i++;
        }
        ausgang = ~ausgang;
    }
}
```

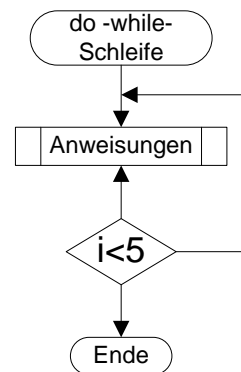
## 5.3 Die “do while” - Schleife

Bei der gerade besprochenen “while” - Schleife stand die Prüfbedingung **vor** den Anweisungen. Sie heißt deshalb “kopfgesteuerte Schleife”.

Im Gegensatz dazu wird bei der “do while” - Schleife die Prüfbedingung an das **Ende der Anweisung** gesetzt (= fußgesteuerte Schleife). Dadurch erzwingt man **mindestens einen Durchgang** durch das Schleifenprogramm, der anschließend -- solange die Bedingung erfüllt ist - beliebig oft wiederholt werden kann.

Auch damit lässt sich sehr elegant eine Tastenabfrage durchführen.  
Beispiel: Die Schaltung macht dann weiter, wenn die Taste 1 auf LOW gelegt wird. Das kann z.B. der Kontakt einer Alarmanlage sein.

```
{
    .....
    do
    {;;}           // mache gar nix
    while(taste1==1); //solange taste1 geöffnet ist
    .....
}
```



## 5.4 Bitverknüpfungen innerhalb von Schleifenabfragen

In Mikrocontrollerprogrammen stoßen Sie häufig auf folgenden Programmcode:

```
while(irgendeineVariable & 0x40){;}    // (1)
while(irgendeineVariable & (1<<6){;}    // (2)
```

(1)

Die Schleife wird in diesem Beispiel solange nicht verlassen, wie die bitweise Verknüpfung der einzelnen Bits der Variable `irgendeineVariable` mit der Bitkombination `0100 0000` ungleich Null ergibt, also das Ergebnis TRUE liefert! Dies ist nur der Fall, wenn das Bit 6 der Variablen gesetzt ist.

(2)

Alternative Schreibweise, aber gleiches Ergebnis und besser lesbar!

### Beispiel:

Die folgenden Tabellen zeigen zwei typische Beispiele für diese Abfrage mit einer bitweisen konjunktiven Verknüpfung. Die abgefragte Variable hat in der linken Tabelle als Beispiel den Wert `0x97` und in der rechten den Wert `0xC3` (`irgendeineVariable = 0x97` und `irgendeineVariable = 0xC3`). Da nur das Bit 6 „abgefragt“ wird, liefert `0x97` ein FALSE und `0xC3` ein TRUE als Ergebnis der Abfrage:

`irgendeineVariable`  
`& 0x40`  
 Ergebnis

1	0	0	1	0	1	1	1
0	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0
= 0, also FALSE							

`irgendeineVariable`  
`& 0x40`  
 Ergebnis

1	1	0	0	0	0	1	1
0	1	0	0	0	0	0	0
0	1	0	0	0	0	0	0
= 0x40 ≠ 0, also TRUE							

## 5.5 Negierung innerhalb von Schleifenabfragen

Für die folgende Abfrage

```
.....
while (taste1==0){
.....
```

kann mit dem **Negierungsoperator** auch geschrieben werden:

```
.....
while (!taste1){
.....
```

Häufig wird dieser Operator verwendet, um auf ein bestimmtes gesetztes Bit in einem Register zu warten. Das Beispiel zeigt Funktionen für das Senden und Empfangen von Daten über die UART Schnittstelle. Die #define Deklarationen repräsentieren die Stellung der einzelnen Bits in dem Statusregister ISR der USART Schnittstelle<sup>3</sup>:

```
//Anordnung der Bits im Register ISR der UART Schnittstelle
#define USART_ISR_RXNE_Pos    (5U)
#define USART_ISR_RXNE_Msk    (0x1U << USART_ISR_RXNE_Pos)    /*!< 0x00000020 */
#define USART_ISR_RXNE        USART_ISR_RXNE_Msk    /*!< Read Data Register Not Empty */

#define USART_ISR_TXE_Pos     (7U)
#define USART_ISR_TXE_Msk     (0x1U << USART_ISR_TXE_Pos)     /*!< 0x00000080 */
#define USART_ISR_TXE        USART_ISR_TXE_Msk    /*!< Transmit Data Register Empty */
. . . . .

void UART1_Sendchar(char c)
{
    while(!(USART1->ISR & USART_ISR_TXE)); // Block until tx empty
    USART1->TDR = c;
}

char UART1_Getchar()
{
    char c;
    while(!(USART1->ISR & USART_ISR_RXNE)); // Nothing received so just block
    c = USART1->RDR; // Read Receiver buffer register
    return c;
}
```

Der Zugriff auf Register wird in den nächsten Kapiteln erklärt

In der Sendefunktion wird darauf gewartet, dass das Bit USART\_ISR\_TXE gesetzt ist und in der Empfangsfunktion wird auf das setzen des Bits LSR\_RDR gewartet.

(Funktionen werden in Kapitel 7 erklärt).

<sup>3</sup> User Manual STM32 Mikrocontroller

## 6 Auswahl

Bisher haben wir folgende Programmstrukturen kennen gelernt:

- In den vorhergehenden Kapiteln die **sequentielle Struktur** (= ein Schritt nach dem anderen wird abgearbeitet, bis alles erledigt ist).
- Speziell im letzten Kapitel die **Wiederholstruktur** (for, do, while).

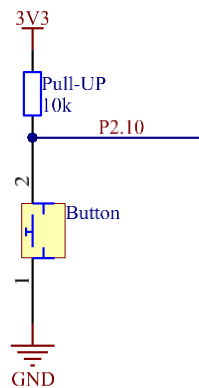
Deshalb wollen wir uns jetzt die **Auswahl** (also die Wahl zwischen verschiedenen Alternativen) ansehen.

### 6.1 Die “if” - Anweisung

Bei der “if” - Anweisung werden die folgende Anweisung (oder ein ganzer Anwendungsblock) nur dann ausgeführt, wenn die hinter der ”if” stehenden Bedingung wahr ist. Auch das eignet sich vorzüglich für irgendwelche Entscheidungen oder Abfragen.

Beispiel: Wenn “Button\_1” gedrückt ist, soll die Variable “ausgang” gesetzt werden. Drückt man dagegen “Button\_2”, wird “ausgang” wieder gelöscht. Drückt man “Button\_3”, so wird geblinkt.

```
{
    .....
    if (Button_1 == 0){
        ausgang = 1;
    }
    if (Button_2 == 0){
        ausgang = 0;
    }
    if (Button_3 == 0){
        blink();
    }
    .....
}
```

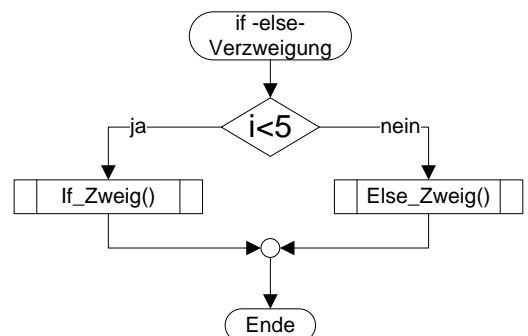


### 6.2 Die “if - else “ - Anweisung

Mit diesem Zusatz kann nun zwischen **zwei Alternativen** gewählt werden.

Beispiel: Wenn der Wert der Variable i kleiner als fünf ist, soll die Variable “ausgang” auf 1 gesetzt werden. Im anderen Fall soll “ausgang” gleich Null sein.

```
{
    .....
    if (i<5){
        ausgang = 1;
    }
    else{
        ausgang = 0;
    }
    .....
}
```

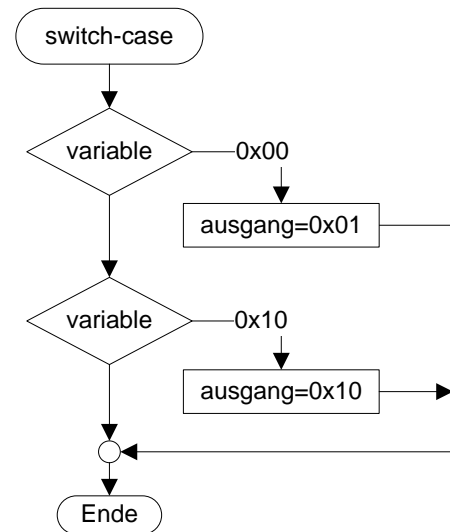


## 6.3 Die “switch” - Anweisung

Mit Hilfe der “switch”- Anweisung ist es möglich, aus einer **ganzen Reihe von Alternativen** ( in Form der “case” - Liste) den gewünschten Fall herauszufischen und eine Reaktion auszulösen.

Beispiel:

```
{
    .....
    switch (variable)
    {
        case 0x00:
            ausgang = 0x01;
            break;
        case 0x10:
            ausgang = 0x10;
            break;
        default:
            ausgang = ~ausgang;
            break;
    }
    .....
}
```



Achtung

- Es ist zulässig, dass mehrere Möglichkeiten gültig sind und dieselbe Wirkung haben. Sie werden einfach nacheinander aufgelistet und dann mit der zutreffenden Vorschrift versehen.
- Passt **keine der Möglichkeiten**, dann wird die “**default**” - Einstellung genommen und ausgeführt

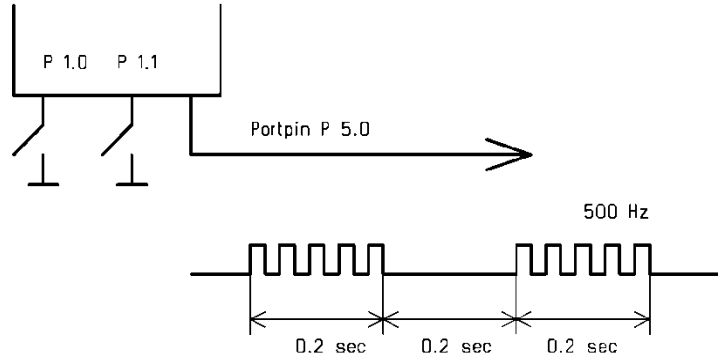
Beispiel: In einem Register mit dem Namen “ergebnis” ist ein Messergebnis oder eine Zahl gespeichert. Abhängig vom genauen Wert soll die Variable „ausgang“ unterschiedliche Werte annehmen.

```
{
    .....
    switch (ergebnis)
    {
        case 0x00:
        case 0x10:
        case 0x20:
            ausgang = 1;
            break;
        case 0x40:
        case 0x50:
        case 0x60:
            ausgang = 0;
            break;
        default:
            ausgang = ~ausgang;
            break;;
    }
    .....
}
```

## 7 C – Anwendungsbeispiel (Alarmanlage)

Aufgabe: es ist eine Alarmanlage aufzubauen, die folgende Eigenschaften aufweist:

- Am Portpin P 1.0 ist die Taste T1 angeschlossen. Wird sie betätigt, dann erzeugt der Controller am Portpin P 1.2 ein Alarmsignal mit dem angegebenen Verlauf.
- Das Alarmsignal kann nur durch Schließen einer weiteren Taste T2 am Portpin P1.1 wieder abgeschaltet werden. Anschließend soll die Alarmanlage sofort wieder "scharf" sein.



Lösung:

```
#include <stdio.h>
#include "stm32f0xx.h"

#define ALARM (1<<0)           //Portpin P 1.0 als Alarm-Auslöser      (GPIOA.0)
#define RESET (1<<1)          //Portpin P 1.1 als Reset-Taste       (GPIOA.1)
#define PIEPS (1<<2)          //Portpin P 1.2 als Audio Ausgang   (GPIOA.2)

void main(void)
{
    uint16_t i,j;               //i und j als 16 - Bit - Zahlen
    init();                     //Init für GPIOs und sonsties
    while (1)                   //Endlosschleife
    {
        do                      //Abfrage der Taste P1.0
        {;;}
        while (GPIOA->IDR & ALARM) //Endlosschleife für Alarmsignal

        while (1)
        {
            for (j=0; j<=2000; j++) //ca. 200 Millisekunden piepsen
            {
                for (i=0; i<=1000; i++){;;} //Intervall
                GPIOA->ODR ^= PIEPS;         //Piepsen
            }
            if (GPIOA->IDR & RESET) //Reset nicht gedrückt?
            {
                GPIOA->ODR &= ~PIEPS;        // dann ca. 200 Millisekunden Ruhe
                for(i=0; i<=2000; i++){;;} //Wartezeit
            }
            else //sonst: Alarm aus, weil RESET gedrückt
            {
                break; //Rückspr. aus der Alarmschl. zur Tastenabfrage
            }
        } //while(1)
    } //while(1)
}
```

Der Zugriff auf Register wird in den nächsten Kapiteln erklärt

## 8 C – Funktionen

Mit Funktionen können C Programme modular aufgebaut werden. Bei längeren Programmen erleichtern Funktionen das Verständnis und damit auch die Fehlersuche.

Im folgenden Code sind drei Funktionen mit unterschiedlichen Parametern eingesetzt:

```

. . .
#define ON 1
#define OFF 0

#include "stm32f0xx.h"

//Funktionsprototypen //5)
void Init(void);
uint8_t Key(void);
uint8_t Led(uint8_t);

int main(void)
{
    uint8_t key;
    Init(); //1)

    while(1)
    {
        key = Tastaturabfrage(); //2)
        if (key)
            Led(ON); //3)
        else
            Led(OFF); //4)
    }
    return 0;
}
. . .

```

(1)

Eine Funktion ohne Parameter

```

/**
    Diese Funktion erhält keine Parameter und gibt auch keine Parameter zurück
    In dem Beispiel wird GPIOA.0 auf Ausgang gestellt und GPIOC.3 als Eingang
*/
void Init(void)
{
    GPIOA->MODER |= (uint32_t)4 (0b01<<0); // 0b01 an Bit 1 und 0
    GPIOC->MODER &= (uint32_t) ~(0b11<<6); // 0b00 an Bit 7 und 6
}

```

(2)

Eine Funktion mit Rückgabewert aber ohne Parameter

```

/**
    Diese Funktion erhält keine Parameter aber gibt einen Wert zurück
*/
uint8_t Key(void)
{
    uint8_t temp;

```

<sup>4</sup> Ohne den sogenannten „cast“ (uint32\_t) würde es auch funktionieren. Diese cast-Anweisung weist dem rechten Teil der Anweisung eine 32 Bit Variable zu.

## Einführung in C für STM32

```
temp = GPIOA->IDR & (uint32_t)(1<<3); // entweder wird 0 oder 0x8 zurückgegeben
return temp;
}
```

(3) und (4)

Eine Funktion ohne Rückgabewert aber mit Parameter

```
/**
    Diese Funktion erhält einen Parameter aber gibt keinen Wert zurück
*/
uint8_t Led(uint8_t led)
{
    if (led==ON)
        GPIOA->ODR |= (uint32_t)(1<<10); // GPIOA.10 high
    else
        GPIOA->ODR &= ~(uint32_t)(1<<10); // GPIOA.10 low
}
```

(5)

In der Sprache C können Namen von Variablen oder Funktionen erst verwendet werden, wenn sie vorher deklariert, also bekannt gemacht worden sind.

Bei Funktionen spricht man von Funktionsprototypen. Dazu wird der Name der Funktion mit den jeweiligen Parametern und Rückgabetyphen hingeschrieben und mit einem Semikolon abgeschlossen. Unter den Ziffern (1), (2), (3) und (4) sind die dazugehörigen Funktionsdefinitionen gezeigt.

```
//Funktionsprototypen //5)
void Init(void);
uint8_t Key(void);
uint8_t Led(uint8_t);
```

Vereinfacht gilt also:

Deklaration	= Namen der Variablen oder Funktion dem Compiler bekanntgeben
Definition	= Maschinencode erzeugen bzw. Speicher reservieren

## 9 Zusammengesetzte Datentypen

### 9.1 Felder

Mehrere Elemente desselben Datentyps können in einem Array (Feld) zusammengefasst werden. Das Beispiel zeigt die Deklaration zweier Felder mit den Datentypen *int32\_t* und *uint8\_t* mit 100 bzw. 20 Werten. Die 100 Werte des *uint8\_t* Feldes werden mit 255 initialisiert:

```
uint8_t feld1[100]; // (1)
int32_t feld2[20]; // (2)

for (uint8_t i=0;i<100;i++) // (3)
    feld1[i] = 0xFF;

feld1[100] = 23; // (4) Fehler, da Index mit 0 beginnt und nur 100
                // Elemente!!!
```



(1)

Deklaration des Feldes **feld1** mit 100 Werten des Datentyps *unsigned char*

(2)

Deklaration des Feldes **feld2** mit 20 Werten des Datentyps *int*.

(3)

Das Feld **feld1** wird mit der Dezimalzahl 255 initialisiert. Der Deklaration des Feldes folgt hiermit eine Definition bzw. Initialisierung, es wird also Speicherplatz reserviert bzw. Maschinencode erzeugt.

(4)

Dies führt zu sehr schwer zu erkennenden Fehlern, der Compiler findet diesen Fehler nicht!!!!

(Der Fehler tritt dann zur Laufzeit des Programms auf und wird auch als *runtime error* bezeichnet)

## 9.2 Strukturen

In einer Struktur können im Gegensatz zu einem Array (Feld) auch unterschiedliche Datentypen zusammengefasst werden.

```
// In der Regel wird die Struktur global deklariert!!!
struct feld {                                //(1)
    uint8_t var1;
    double var2;
    uint32_t var3;
    int16_t var4[100];                        //(2)
};

// Die Variablen sollten natürlich soweit als möglich lokal deklariert werden!!!
int Irgendeine_Funktion_oder_main(void)
{
    struct feld var_mit_struct;              //(3)

    var_mit_struct.var1 = 255;                //(4)
    var_mit_struct.var2 = 3.1415;
    var_mit_struct.var3 = 3456789;
    for (uint8_t i=0;i<100;i++)
        var_mit_struct.var4[i] = -1;         //(5)

    . . .
```

In C können eigene neue Datentypen mit dem Bezeichner **typedef** erzeugt werden. Die Deklaration wird damit kürzer und leichter lesbar:

```
typedef struct feld {                        //(6)
    uint8_t var1;
    double var2;
    uint32_t var3;
} feld_t;

feld_t var_mit_struct;                       //(7)
```

(1)

Deklaration der Struktur. In der **struct** können beliebige C Datentypen enthalten sein, aber keine Funktionen (Funktionen in Strukturen sind nur in C++ erlaubt, aus der Struktur wird damit die Klasse).

(2)

Hier wird in einer **struct** ein Feld mit 100 Werten deklariert, auch das ist möglich.

(3)

Um eine **struct** verwenden zu können, muss eine Variable dieses Typs deklariert werden.

(4)

Zugriff auf die Elemente der Struktur mit dem Punkt Operator (.):

```
var_mit_struct.var1 = 28;
```

(5)

Alle 100 Elemente des Feldes var4 innerhalb der **struct** werden mit -1 initialisiert.

(6 und 7)

Die Deklaration einer Variablen mit dem Datentyp **struct** geht kürzer, wenn die Struktur mit dem Schlüsselwort **typedef** deklariert wurde. Mit **typedef** können eigene Bezeichner erzeugt werden, in diesem Beispiel wird die Struktur **struct feld** mit den drei Elementen var1, var2 und var3 durch den neuen Bezeichner **feld\_t** ersetzt. Für die Deklaration der Variablen var\_mit\_struct muss nun lediglich der Name **feld\_t** davor geschrieben werden und **var\_mit\_struct** wird zu einer Variablen des Typs **struct feld**.

### 9.3 Schreiben von Gleitkommazahlen in Strings

Sehr oft möchte man Gleitkommazahlen ausgeben, aber viele Mikrocontroller bieten in der „abgespeckten“ Version der „stdio“ Bibliothek den Formatierungsparameter „%f“ nicht zur Verfügung. Dieses Problem kann aber mit der Verwendung des „%d“ Parameters und wenigen Rechenoperationen einfach umgangen werden.

Die Variable **frq\_Hz** sei vom Datentyp **double**, also ein Wert z.B. **frq\_Hz = 5,012 Hz**.

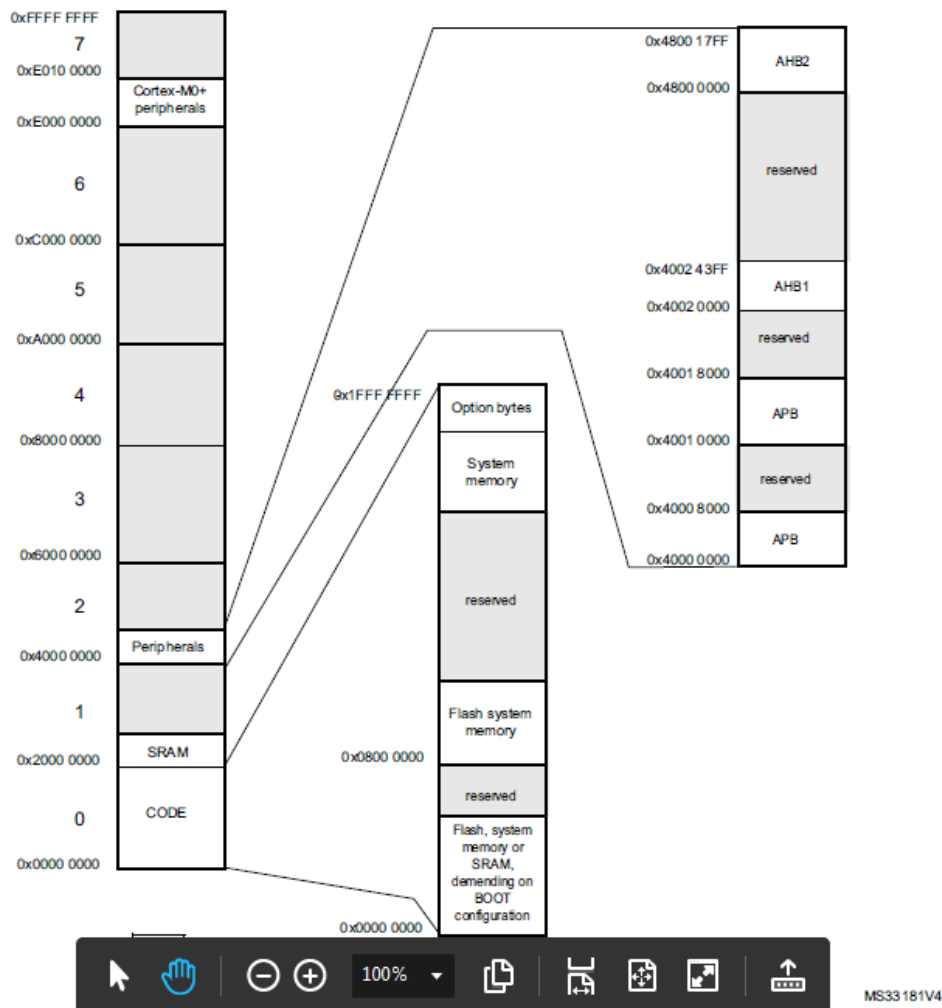
- Die Berechnung  $((\text{frq\_Hz} * 1000) / 1000)$  ergibt als Ergebnis 5, da das Rechenergebnis als Integervariable interpretiert wird.
- Die Berechnung  $((\text{frq\_Hz} * 1000) \% 1000)$  ergibt als Ergebnis 12, da das Rechenergebnis als Integervariable interpretiert wird ergibt der Rest der Berechnung die Zahl 12.
- In dem String steht nach dem Aufruf von **sprintf**: **"Frequenz = 5.012 Hz\r\n"**

```
snprintf(str, sizeof(str), "Frequenz = %d.%d Hz\r\n", (uint32_t)(frq_Hz * 1000) / 1000, (uint32_t)(frq_Hz * 1000) % 1000);
```

## 10 STM32 spezifische Programmierung

### 10.1 Zugriff auf Register des STM32 Microcontrollers

Die Peripheriemodule des  $\mu$ C werden durch das setzen bzw. löschen von vorgegebenen Bits in einer großen Zahl von 32-Bit Registern eingestellt. Die folgenden Bilder aus dem Datenbuch zeigen die Adressen dieser Register bzw. einen Ausschnitt aus dem Adressraum des 32-Bit Mikrocontrollers



### 10.2 Wie kann in der Sprache C auf diese Register zugegriffen werden?

Die Lösung sind Zeiger. Im Gegensatz zu der üblichen Verwendung **müssen** Sie diese auf die vorgegebenen Adressen der jeweiligen  $\mu$ Controller -Register zeigen lassen.

In Betriebssystemen wie Windows oder Linux ist das nicht möglich, da das Betriebssystem die Speicherbereiche verwaltet. Wenn Sie in einem Windows- oder Linuxprogramm einen Zeiger auf x-beliebige Adressen zeigen lassen, erhalten Sie einen *Runtime-Error* mit einer Meldung „Nicht erlaubter Speicherzugriff“ o.ä.

### 10.2.1 Ein Zeiger, was ist das?

Zeiger werden in der Sprache C für den Zugriff auf bestimmte Adressen verwendet und diese Fähigkeit ist sicherlich eine der Gründe für die Verbreitung von C bei der Programmierung von Mikrocontrollern.

```
uint8_t *pvar;           // 1) Zeiger deklariert, zeigt auf nix
uint8_t var;             // Variable deklariert

var = 7;                 // Variable hat den Wert 7
pvar = &var;             // 2) Variable

*pvar = 8;               // 3)
```

1)

Um einen Zeiger zu deklarieren muss ein \* vor einen Variablennamen gesetzt werden. Der Zeiger darf erst verwendet werden, wenn ihm eine Adresse zugewiesen wird, er also auf einen bestimmten Speicherplatz zeigt. Zur Unterscheidung zu „normalen“ Variablen sollte der Name mit einem p beginnen (= Pointer), also **\*pvar**, **\*irgendwas**, etc..

2)

Der Zeiger **pvar** zeigt mit dem & Operator nun auf die Adresse an der die Variable **var** liegt. Dieser & Operator auf eine normale Variable angewendet liefert die Adresse, an der diese Variable abgelegt ist

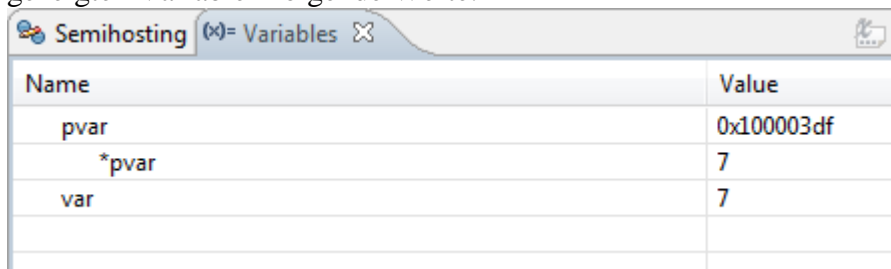
3)

Der Wert in dieser Adresse ist nun 8.

In Verbindung mit Zeigern werden also zwei neue Operatoren verwendet:

- **& Referenzierungsoperator** : Mit diesem Operator erhält man die Adresse einer Variable
- **\* Dereferenzierungsoperator** : Mit diesem Operator kann auf den Wert einer Adresse zugegriffen werden. Dieser Operator kann nur auf einen Zeiger angewendet werden (z.B. würde der Compiler die Anweisung **\*var = 8;** als Fehler melden)

Nach dem Ablaufen des gezeigten Codes einschließlich der Zeile **pvar = &var;** besitzen die gezeigten Variablen folgende Werte:



Name	Value
pvar	0x100003df
*pvar	7
var	7

Der Zeiger **pvar** zeigt auf die Adresse 0x100003df. Der Inhalt dieser Adresse ist der Wert 7.

Nach der letzten Codezeile wird in die Adresse 0x100003df der Wert 8 geschrieben.

**Mit diesen Zeigern ist es möglich auf die Adressen der einzelnen Register des µControllers zuzugreifen. Es werden in der Praxis zwei Verfahren angewendet:**

## 10.3 Registerzugriff in der Sprache C in der Praxis

### 10.3.1 Ein Zeiger auf eine Struktur – CMSIS Standard-

Da die Cortex Mikrocontroller über eine Vielzahl von Peripheriemodulen verfügen, z.B. viele Timer und viele serielle Schnittstellen sind auch eine Vielzahl von Registern für das Einstellen nötig. Deshalb wurden zusammengehörige Register jeweils in einer Struktur (struct) zusammengefasst. Um auf diese Strukturen zugreifen zu können, müssen Sie für den STM32 die folgende Headerdatei einfügen:

```
#include "stm32f0xx.h"
```

In der Regel müssen Sie sich nicht um diese „Kleinigkeiten“ kümmern, da Sie ja eine integrierte Programmierumgebung einsetzen und diese kümmert sich um diese Dinge.

Diese Include-Dateien stammen aus der CMSIS<sup>5</sup>- Bibliothek. Sie können damit anstelle von endlos vielen Zeigern auf einzelne Register einen wesentlich übersichtlicheren Zugriff über den Datentyp *struct* verwenden.

Der folgende Codeausschnitt stammt aus dieser CMSIS Headerdatei für die Peripherieeinheit des DAC<sup>6</sup>:

```
/**
 * @brief Digital to Analog Converter
 */

typedef struct
{
    __IO uint32_t CR;           /*!< DAC control register, Address offset: 0x00 */
    __IO uint32_t SWTRIGR;     /*!< DAC software trigger register,Address offset: 0x04 */
    . . . . .
    __IO uint32_t DOR1;
    __IO uint32_t DOR2;
    __IO uint32_t SR;
} DAC_TypeDef;                //(1)

. . . . .

#define PERIPH_BASE            ((uint32_t)0x40000000U)
. . . . .
#define APBPERIPH_BASE        PERIPH_BASE
. . . . .
#define DAC_BASE               (APBPERIPH_BASE + 0x00007400)    //(2)
. . . . .
#define DAC1                   ((DAC_TypeDef *) DAC_BASE)        //(3)
```

(1)

Für die Struktur wird der neue Bezeichner `DAC_TypeDef` deklariert.

<sup>5</sup> Cortex Microcontroller Software Interface Standard

<sup>6</sup> DAC = Digital Analog Converter

(2)

Der DAC beginnt bei der Adresse 0x40007400

(3)

Hier wird der Bezeichner **DAC1** definiert, der einen Zeiger auf die oben genannte Adresse darstellt.

Auf die DAC-Register kann nun sehr einfach zugegriffen werden

```
(*DAC1).CR = (1<<3) | (1<<2) | (1<<1);           // (1)

// oder gleichbedeutend

DAC1->CR = (1<<3) | (1<<2) | (1<<1);             // (2)
```

(1)

Hier werden die Bits 1 bis 3 des DACCTRL- Registers gesetzt. Der Zugriff auf das Register erfolgt nun über einen Zeiger auf eine Struktur.

(2)

Da Zeiger auf Strukturen in C häufig genutzt werden gibt es dafür einen eigenen Operator, den (->) Operator.

10.3.1.1 Beispiel1: Toggeln des 1. Bits von PortA:

```
while(1){
    if (GPIOA->ODR & (1 << 0))
        GPIOA->ODR |= (1 << 0);
    else
        GPIOA->ODR &= ~(1 << 0);
}
```

Das funktioniert natürlich auch viel einfacher mit dem folgenden Code, hier wird eine EXOR Verknüpfung benutzt:

```
while(1){
    GPIOA->ODR ^= (1 << 0);
}
```

10.3.1.2 Beispiel 2: Ausgabe des Prozessortaktes an PortA Pin 8 (GPIOA.8):

```
// Takt ausgeben

GPIOA->MODER &= ~(0b11 <<16);           // Alternate GPIOA.8
GPIOA->MODER |= (0b10 <<16);           // Alternate GPIOA.8
RCC->CFGR |= (0x4<<24);                 // SystemClock Output 4
GPIOA->AFR[0] |= (0x04 << 0);           // PA8 als MCO Alt.Funktion 4
```

### 10.3.1.3 Beispiel 3: Einstellen des Timer 6

Die STM32 Mikrocontroller besitzen Timer mit vielen Einstellmöglichkeiten und Betriebsarten. Als einfachstes Beispiel ist hier der Timer6 gewählt. Das Vorgehen für die Einstellung der Timer zeigt der folgende Abschnitt aus dem Benutzerhandbuch (S. 490)

```
// Wartefunktion mit dem Timer 6

void delay_us_Timer6(uint16_t warte_in_us)
{
    TIM6->SR = 0;                // 1)
    TIM6->PSC = 8;                // 2)
    TIM6->ARR = warte_in_us;      // 3)
    TIM6->CR1 |= TIM_CR1_CEN;     // 4)
    while (!(TIM6->SR & TIM_SR_UIF)); // 5)
}
```

- Zu 1)  
Das Statusregister wird gelöscht, in diesem Register ist nur das unterste Bit relevant, es wird gesetzt sobald Änderungen an den Registern erfolgt sind (UIF = Update Interrupt Flag).
- Zu 2)  
In diesem Register wird der Prescaler eingestellt, daraus ergibt sich die Frequenz des Timers. Bei einer CPU-Frequenz von 8 MHz ergibt sich mit dem eingestellten Wert eine Frequenz von 1 MHz, der Timer läuft alle Mikrosekunden um einen Zähler nach oben.
- Zu 3)  
Hier wird der Autoreloadwert geschrieben, der Timer läuft von 0 bis zu dem Wert im Autoreloadregister. ARR = 10000 würde hier bedeuten der Timer läuft 10 Millisekunden.
- Zu 4)  
Hier wird der Timer gestartet. Der Bezeichner TIM\_CR1\_CEN ist deklariert als (1<<0) und setzt das unterste Bit des Kontrollregisters, damit beginnt der Timer zu laufen.
- Zu 5)  
Das Programm bleibt solange in der Schleife bis das UIF Bit gesetzt ist, siehe Punkt 1

## 10.4 Hardware Abstraction Layer (CubeMX-HAL)

In der Praxis ist mühselig die Bits in den unendlich vielen Registern der Mikrocontroller einzustellen. Der von ARM entwickelte „Cortex Microcontroller Software Interface Standard“ (CMSIS) stellt einen standardisierten Hardware Abstraction Layer (HAL) für Cortex-M3-Mikrocontrollersysteme dar und definiert einheitliche Standards für:

- Zugriff auf die Peripherieregister sowie die Definition von Ausnahme- (Interrupt-) vektoren.
- Registerdefinitionen für Core, den Nested Vector Interrupt Controller (NVIC), die Memory Protection Unit (MPU) sowie für diverse andere Hardwarekomponenten.
- Systeminitialisierung
- Organisation der Header-Dateien
- Zugriff auf den Systemtakt
- Und vieles mehr.....

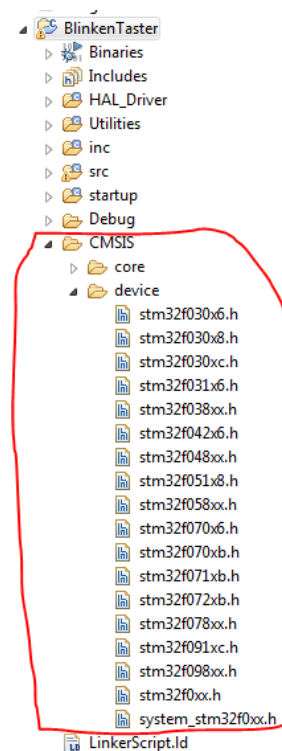
Der aktuelle HAL für STM32 Mikrocontroller ist STM32Cube.

### CMSIS und CubeMX-HAL

Ein Projekt für den STM32 Mikrocontroller, das Sie mit der OpenSTM IDE erzeugen enthält ein Programmgerüst in dem bereits vom Hersteller ARM bereitgestellten CMSIS<sup>7</sup> Headerfiles integriert sind (siehe rechts).

CMSIS ist ein von den Chipherstellern unabhängiger „Hardware Abstraction Layer“ für die Cortex-M Mikrocontroller Serie. Es wird damit eine konsistente und relativ einfache Software Schnittstelle für die Programmierung der Peripherie, Echtzeitbetriebssysteme und Middleware zur Verfügung gestellt. Die Wiederverwendung bestehender Software wird erleichtert und Einarbeitung wird vereinfacht.

Für das Programmieren mit dem STM32 Chip wird aber noch eine Hersteller spezifische HAL benötigt. Diese wird durch STM32Cube Framework bereitgestellt



die

die

Das

oben gezeigte Bild ist eine vereinfachte Darstellung der Zusammenhänge bei der Erstellung der Firmware..

CMSIS ist die universelle Sammlung der Headerdateien, wurde von ARM entwickelt und ist für alle Cortex-M Hersteller identisch (ST, NXP, Atmel, ...).

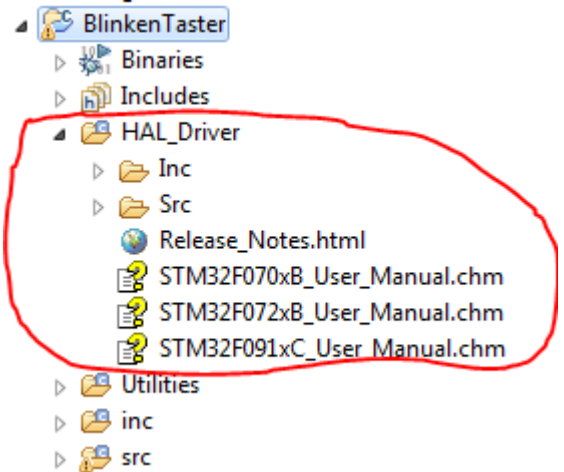
<sup>7</sup> CMSIS = Cortex Mikrocontroller Software Interface Standard



## Einführung in C für STM32

ST-HAL ist der „Hardware Abstraction Layer“ von ST für die spezifischen Derivate (F0, F4, F7, ...). Die Device-HAL kann als eine Art „Stecker“ angesehen werden womit eine Verbindung zwischen den Subsystemen CMSIS und ST-Hal hergestellt wird.

Für den Benutzer sind hauptsächlich die im Verzeichnis HAL\_Driver definierten Funktionen wichtig.



### 10.4.1 Prinzip Software OpenSTM

Der Aufbau Ihres Benutzerprogramms ist immer ähnlich. Hier ist ein Beispiel (Ausschnitt) für den CAN-Bus gezeigt.

```
#include "stm32f0xx.h" //1)
#include "stm32f0xx_nucleo.h" //2)

void SystemClock_Config(void);
. . .

CAN_HandleTypeDef C; //3)
static CanTxMsgTypeDef TxMessage;
. . .

int main(void)
{
    HAL_Init(); //4)
    SystemClock_Config(); //5)

    __HAL_RCC_GPIOA_CLK_ENABLE(); //6)

    GPIO_InitTypeDef GPIO_InitStruct; //7)

    GPIO_InitStruct.Pin = GPIO_PIN_11 | GPIO_PIN_12;
    GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
    GPIO_InitStruct.Pull = GPIO_NOPULL;
    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_HIGH;
    GPIO_InitStruct.Alternate = GPIO_AF4_CAN;
    HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);

    __HAL_RCC_CAN1_CLK_ENABLE(); //8)
    C.Instance = CAN;
    C.Init.Mode = CAN_MODE_NORMAL;
    C.Init.Prescaler = 24;
    . . .
    HAL_CAN_Init(&C); //9)

    for(;;){
        HAL_CAN_Transmit(&C, timeout_ms);
        HAL_Delay(1000);
    }
}
```

1)

Als erstes kommen die include Anweisungen für den Präprozessor

2)

Alle Funktionen die unterhalb von main **definiert** sind müssen hier **deklariert** werden. Ganz einfach den Funktionsnamen schreiben mit den Übergabeparametern und das Semikolon nicht vergessen! (Funktionsdeklaration = Funktionsprototyp)

3)

Hier wird für das Senden des CAN-Bus eine Variable des Datentyps `CanTxMsgTypeDef` deklariert. Das Schlüsselwort *static* bewirkt, dass diese Variable auf einer festen Speicheradresse abgelegt wird (Praktisch eine Variable mit Gedächtnis). Ohne *static* geht's in diesem Kontext auch.

4)

Diese Funktion muss aufgerufen werden sonst funktioniert die Cube-HAL Library nicht fehlerfrei.

5)

Hier wird der Takt eingestellt, wenn diese Funktion nicht aufgerufen wird läuft der uC im Beispiel mit 8 MHz anstelle 48 MHz (stm32F072RB).

6)

Takt einschalten bevor ein Modul konfiguriert wird, im Beispiel GPIO

7)

Konfiguration des Moduls, im Beispiel die GPIOs für den CAN-Bus.

8)

Takt einschalten bevor ein Modul konfiguriert wird, im Beispiel CAN-Bus, anschließend wird der CAN-Bus konfiguriert-

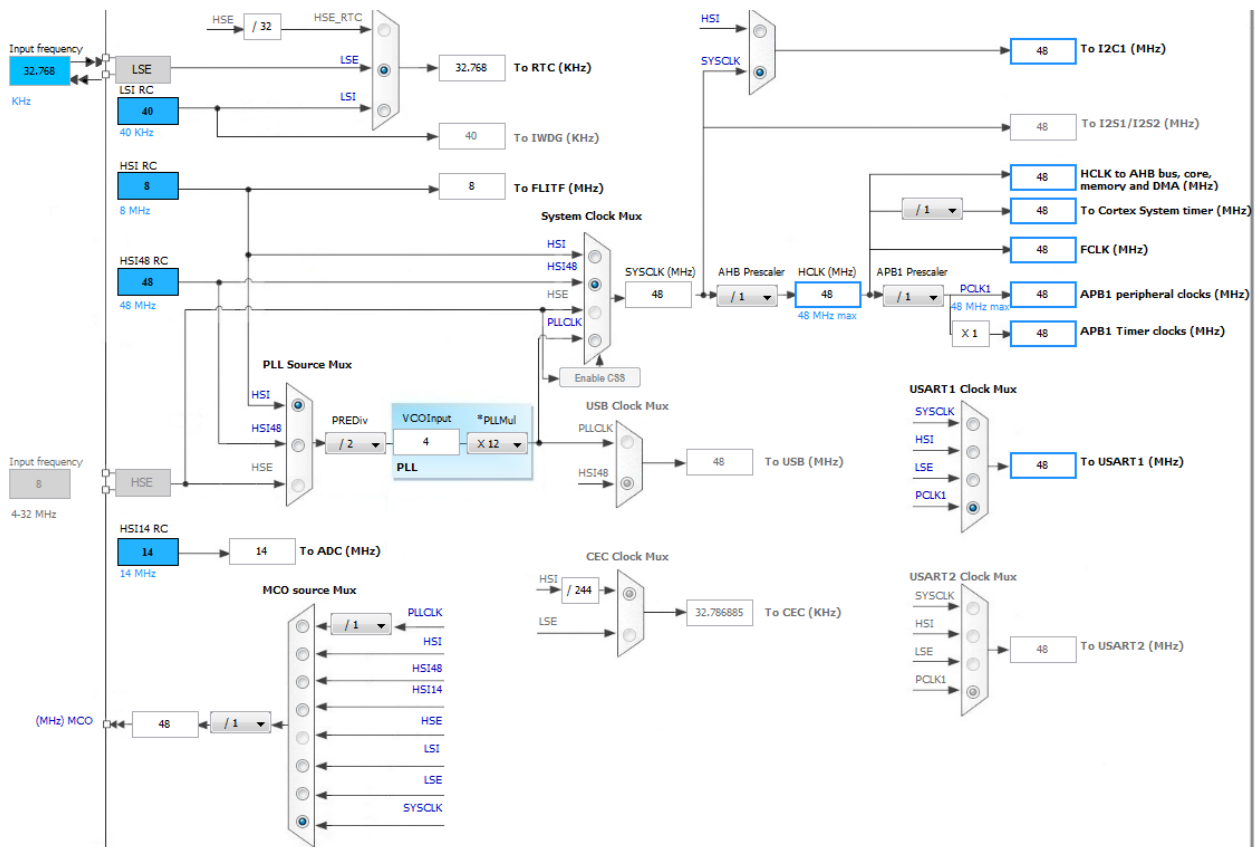
9)

Das ist die entscheidende Funktion die die Werte der struct in die Register schreibt.

### 10.4.2 Beispiel Takteinstellung

Mit Cube-MX-HAL können die einzelnen Peripherieelemente graphisch eingestellt werden, das macht das Vorbereiten eines Projekts noch einfacher. Das Bild zeigt exemplarisch die Einstellung der verschiedenen Takte für die Peripherieelemente des Praktikumsboards:

## Einführung in C für STM32



Der mit dieser Software erzeugte Code kann durch den Aufruf dieser Funktion sofort in Ihr Projekt eingebunden werden. Zuvor müssen Sie diese Funktion selbstverständlich in Ihr Projekt kopieren:

```
/** System Clock Configuration
 */
void SystemClock_Config(void)
{
    RCC_OscInitTypeDef RCC_OscInitStruct;
    RCC_ClkInitTypeDef RCC_ClkInitStruct;
    RCC_PeriphCLKInitTypeDef PeriphClkInit;

    /**Initializes the CPU, AHB and APB busses clocks
     */
    RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI|RCC_OSCILLATORTYPE_LSE;
    RCC_OscInitStruct.LSEState = RCC_LSE_ON;
    RCC_OscInitStruct.HSIState = RCC_HSI_ON;
    RCC_OscInitStruct.HSICalibrationValue = 16;
    RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
    RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
    RCC_OscInitStruct.PLL.PLLMUL = RCC_PLL_MUL12;
    RCC_OscInitStruct.PLL.PREDIV = RCC_PREDIV_DIV2;
    if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
    {
        Error_Handler();
    }
}
```

```

}

/**Initializes the CPU, AHB and APB busses clocks
 */
RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
                             |RCC_CLOCKTYPE_PCLK1;
RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;

if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_1) != HAL_OK)
{
    Error_Handler();
}

PeriphClkInit.PeriphClockSelection = RCC_PERIPHCLK_USART1|RCC_PERIPHCLK_I2C1
                             |RCC_PERIPHCLK_RTC;
PeriphClkInit.Usart1ClockSelection = RCC_USART1CLKSOURCE_PCLK1;
PeriphClkInit.I2c1ClockSelection = RCC_I2C1CLKSOURCE_HSI;
PeriphClkInit.RTCClockSelection = RCC_RTCCLKSOURCE_LSE;
if (HAL_RCCEx_PeriphCLKConfig(&PeriphClkInit) != HAL_OK) {
    Error_Handler();
}

/**Configure LSE Drive Capability
 */
__HAL_RCC_LSEDRIVE_CONFIG(RCC_LSEDRIVE_LOW);

HAL_RCC_MCOConfig(RCC_MCO, RCC_MCO1SOURCE_HSI, RCC_MCODIV_1);

/**Configure the SysTick interrupt time
 */
HAL_SYSTICK_Config(HAL_RCC_GetHCLKFreq()/1000);

/**Configure the SysTick
 */
HAL_SYSTICK_CLKSourceConfig(SYSTICK_CLKSOURCE_HCLK);

/* SysTick_IRQn interrupt configuration */
HAL_NVIC_SetPriority(SysTick_IRQn, 0, 0);
}

```

## 10.5 Interrupts

Der Cortex M3 Mikroprozessor enthält einen mächtigen InterruptController (NVIC<sup>8</sup>) der Resets, Software Interrupts und Hardware Interrupts ermöglicht. Im folgenden Kapitel wird das Vorgehen für Hardware Interrupts näher behandelt.

Jeder der Interrupts hat einen zugehörigen 32-Bit Adressvektor, der auf eine Adresse zeigt an der die Interruptroutine (ISR<sup>9</sup>)beginnt. Diese Vektoren sind im ROM gespeichert und beginnen bei der Adresse 0.

Einen Ausschnitt dieser Interruptvektoren zeigt die Tabelle aus dem Referenzmanual:

<sup>8</sup> Nested Vector Interrupt Controller

<sup>9</sup> Interrupt Service Routine

**Table 36. Vector table (continued)**

Position	Priority	Type of priority	Acronym	Description	Address
3	10	settable	FLASH	Flash global interrupt	0x0000 004C
4	11	settable	RCC_CR	RCC and CRS global interrupts	0x0000 0050
5	12	settable	EXTI0_1	EXTI Line[1:0] interrupts	0x0000 0054
6	13	settable	EXTI2_3	EXTI Line[3:2] interrupts	0x0000 0058
7	14	settable	EXTI4_15	EXTI Line[15:4] interrupts	0x0000 005C
8	15	settable	TSC	Touch sensing interrupt	0x0000 0060
9	16	settable	DMA_CH1	DMA channel 1 interrupt	0x0000 0064
10	17	settable	DMA_CH2_3 DMA2_CH1_2	DMA channel 2 and 3 interrupts DMA2 channel 1 and 2 interrupts	0x0000 0068
11	18	settable	DMA_CH4_5_6_7 DMA2_CH3_4_5	DMA channel 4, 5, 6 and 7 interrupts DMA2 channel 3, 4 and 5 interrupts	0x0000 006C
12	19	settable	ADC_COMP	ADC and COMP interrupts (ADC interrupt combined with EXTI lines 21 and 22)	0x0000 0070

Der Abstand zwischen zwei Interruptadressen (Interruptvektoren) sind immer 4 Byte, also 32 Bit. Mehrere Ereignisse eines Moduls lösen denselben Interrupt aus, bzw. haben denselben Interruptvektor, z.B. werden vom ADC\_COMP mehrere Ereignisse zum selben Interruptvektor 0x70 geleitet. Durch die Abfrage bestimmter Bits in einem der Interruptregister kann dann die jeweilige Quelle zugeordnet werden. Man spricht hier von „*polled Interrupts*“ im Gegensatz zu den „*vectored Interrupts*“. Bei den „*vectored Interrupts*“ ist jeder Interruptquelle ein Interruptvektor zugeordnet.

Der C Programmierer muss sich um diese Interruptadressen nicht kümmern, dies wird von einem sogenannten *StartupCode* erledigt. Je nach der Programmierungsumgebung sind diese Dateien etwas unterschiedlich. Ein Ausschnitt aus der Datei `startup_stm32f072xb.s` der OpenSTM32 IDE ist hier abgebildet:

```

/*****
*
* Provide weak aliases for each Exception handler to the Default_Handler.
* As they are weak aliases, any function with the same name will override
* this definition.
*
*****/

.weak      NMI_Handler
.thumb_set NMI_Handler,Default_Handler

.weak      HardFault_Handler
.thumb_set HardFault_Handler,Default_Handler

.weak      SVC_Handler
.thumb_set SVC_Handler,Default_Handler

.weak      PendSV_Handler
.thumb_set PendSV_Handler,Default_Handler

.weak      SysTick_Handler
.thumb_set SysTick_Handler,Default_Handler

```

In dem Codeausschnitt können Sie einige Bezeichnungen für die Interrupts (PendSVHandler, SysTick\_Handler, ...) ablesen. Wichtig sind lediglich die Namen der Interruptroutinen, da diese nicht verändert werden dürfen, andernfalls müssten Sie die Startup-Datei ändern.

Falls Sie z.B. für den SysTick Timer eine Interruptroutine schreiben, muss diese so aussehen:

```
void SysTick_Handler(void)
{
    . . . . .
}
```

### 10.5.1 Beispiel Interrupt bei Timer3 Überlauf

Zunächst muss der Timer3 korrekt konfiguriert werden. In dem Beispiel soll der Timer3 mit einer Periode  $T=100$  ms überlaufen und einen Interrupt auslösen. Da es sich um einen 16-Bit Timer handelt ist der maximale Zählerwert bis zum Überlauf 65535!

```
TIM_HandleTypeDef htim3;

__HAL_RCC_TIM3_CLK_ENABLE();

TIM_ClockConfigTypeDef sClockSourceConfig;
TIM_MasterConfigTypeDef sMasterConfig;

htim3.Instance = TIM3;
htim3.Init.Prescaler = 479; // 1)
htim3.Init.CounterMode = TIM_COUNTERMODE_UP;
htim3.Init.Period = 9999; // 2)

htim3.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
htim3.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
HAL_TIM_Base_Init(&htim3);

sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
HAL_TIM_ConfigClockSource(&htim3, &sClockSourceConfig);

/* Die folgenden Default Zeilen können auch weggelassen werden */
sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET; // default
sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE; // default
HAL_TIMEx_MasterConfigSynchronization(&htim3, &sMasterConfig); // default

HAL_TIM_GenerateEvent(&htim3, TIM_EVENTSOURCE_UPDATE);

/* Peripheral interrupt init */
HAL_NVIC_SetPriority(TIM3_IRQn, 0, 0);
HAL_NVIC_EnableIRQ(TIM3_IRQn);

HAL_TIM_Base_Start_IT(&htim3); // 3)
```

1)

Der eingestellte Prescaler von 479 ergibt bei  $f = 48$  MHz eine Taktrate des Timers von 100kHz bzw. einen Timetick von  $T_{\text{timetick}} = 10$  Mikrosekunden:

## Einführung in C für STM32

$$f_{timetick} = \frac{f_{Prozessortakt}}{Prescaler + 1}$$

$$T_{timetick} = T_{Prozessortakt} * (Prescaler + 1)$$

2)

Hier wird die Periode des Timers eingestellt, also die Zeit bis zum Überlauf des Timers  
Für die Berechnung kann die folgende Formel verwendet werden:

$$f_{Timer} = \frac{f_{timetick}}{Reload + 1}$$

$$Reload = \frac{f_{timetick}}{f_{Timer}} - 1$$

Beispiel:

Eingestellt werden soll  $f_{Timetick}=10\text{Hz}$

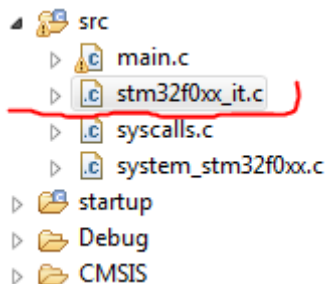
$f_{timetick} = 48 \text{ MHz}/480 = 100 \text{ kHz}$ . Siehe oben

Reloadwert =  $(100\text{kHz}/10\text{Hz}) - 1 = \text{htim3.Init.Period} = 9.999$

3)

Hier wird der Timer im Interruptmode gestartet

Der Ablauf vom Interruptereignis bis zu der vom Benutzer geschriebenen Interruptroutine ist zugegeben ein wenig umständlich da mehrstufig.



1) Für alle Interruptroutinen ist in der Projektumgebung die Datei `stm32f0xx.c` bereits vorbereitet wird bereits von der Entwicklungsumgebung bereitgestellt.

Die Interruptserviceroutine lautet **TIM3\_IRQHandler** und muss falls noch nicht vorhanden in diese Datei geschrieben werden.

```
void TIM3_IRQHandler(void)
{
    HAL_TIM_IRQHandler(&htim3);
}
```

2) In dieser Routine muss die Funktion `HAL_TIM_IRQHandler` aufgerufen werden. Diese ist Teil der Cube-HAL und in dieser Funktion werden je nach Interruptfreigabe die von der Applikation benötigten Routinen aufgerufen. Für den Interrupt bei Überlauf des Timers lautet diese Funktion `HAL_TIM_PeriodElapsedCallback(..)`. In dieser Funktion muss dann der Benutzecode stehen, z.B. ein Blinklicht.

```
void HAL_TIM_PeriodElapsedCallback(&htim3)
{
    Blinklicht();
}
```

```
}
```

Aufrufreihenfolge:





## 10.5.2 Externe Interrupts

An welchen Pins können externe Interrupts ausgelöst werden?

Das folgende Bild zeigt die Interrupts bei einem STM32 F0 Mikrocontroller.

Die 16 Interruptvektoren sind mit unterschiedlichen GPIOs verbunden. Im Bild ist EXTI0, EXTI1 und EXTI15 gezeigt.

Exemplarisch wird der Taster (PC13) des Evaluationboards als externer Interrupt verwendet.

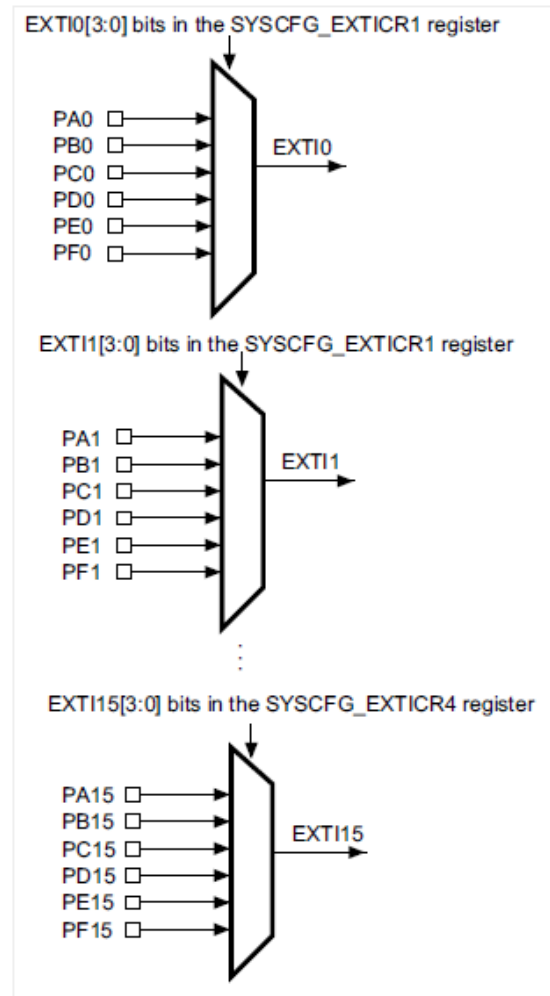
```
/* Test für einen externen interrupt am User
   Button (PC13)
*/

GPIO_InitTypeDef G;

G.Pin = GPIO_PIN_13;
G.Mode = GPIO_MODE_IT_FALLING; // 1)
G.Pull = GPIO_NOPULL;
HAL_GPIO_Init(GPIOC, &G);
```

Das Einstellen des Ports erfolgt nach demselben Schema wie es bereits bei dem Blinklichtbeispiel gezeigt wurde.

- 1) Im Element `G.Mode` der Struct muss der passende Typ eingestellt werden, für eine Interruptauslösung bei der fallenden Flanke muss hier `GPIO_MODE_IT_FALLING` stehen.



Nun muss der Interrupt freigegeben werden, dafür reicht ein einfacher Funktionsaufruf:

```
/* EXTI interrupt init*/
HAL_NVIC_SetPriority(EXTI4_15_IRQn, 0, 0); // 1)
HAL_NVIC_EnableIRQ(EXTI4_15_IRQn);        // 2)
```

- 1) Einstellen der Priorität, ist nur wichtig wenn mehrere Interrupts freigegeben sind.
- 2) Hier wird der Interrupt freigegeben.

## Einführung in C für STM32

Nun fehlt nur noch der Interrupthandler (auch ISR<sup>10</sup> bezeichnet). Die Bezeichnung findet man im Startup-Code in der Datei `startup_stm32f072xb.s`:

```
. . . . .
.word EXTI0_1_IRQHandler          /* EXTI Line 0 and 1      */
.word EXTI2_3_IRQHandler          /* EXTI Line 2 and 3      */
.word EXTI4_15_IRQHandler         /* EXTI Line 4 to 15     */
. . . . .
```

Also lautet die Funktionsdefinition:

```
/**
 * @brief This function handles EXTI line0 interrupt.
 */
void EXTI4_15_IRQHandler(void)
{
    HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_13);    //1)
    tu_was_wichtiges();                       //2)
}
```

- 1) Diese Funktion kümmert sich um das rücksetzen des Interruptflags
- 2) Hier ist dann der Benutzer-Code

### 10.5.3 Polled bzw. Vectored Interrupts

Mit Vectored Interrupts werden Interrupts bezeichnet, die bei Auslösung zu einem Sprung an eine vorgegebene Adresse im Programmspeicher führen.

Bei den gepollten Interrupts werden mehrere Interruptquellen zusammengefasst. Die Auswahl des Interrupts erfolgt über den Eintrag in einem Interruptstatusregister.

Die Peripherie der Cortex-M Familie verfügt über gepollte Interrupts. Beispielsweise löst die UART Schnittstelle 6 unterschiedliche Interrupts aus.

---

<sup>10</sup> Interrupt Service Routine

## 10.6 GPIO-Ports

Mit dem Begriff „GPIO-Port“<sup>11</sup> werden parallele digitale Ein- und Ausgabeschchnittstellen bezeichnet. Diese sind bereits im Chip integriert und meist bitweise ansprechbar. Je nach Controller bewegt sich die Anzahl der 32-Bit breiten Ports von einem bis zu mehr als 3 Ports.

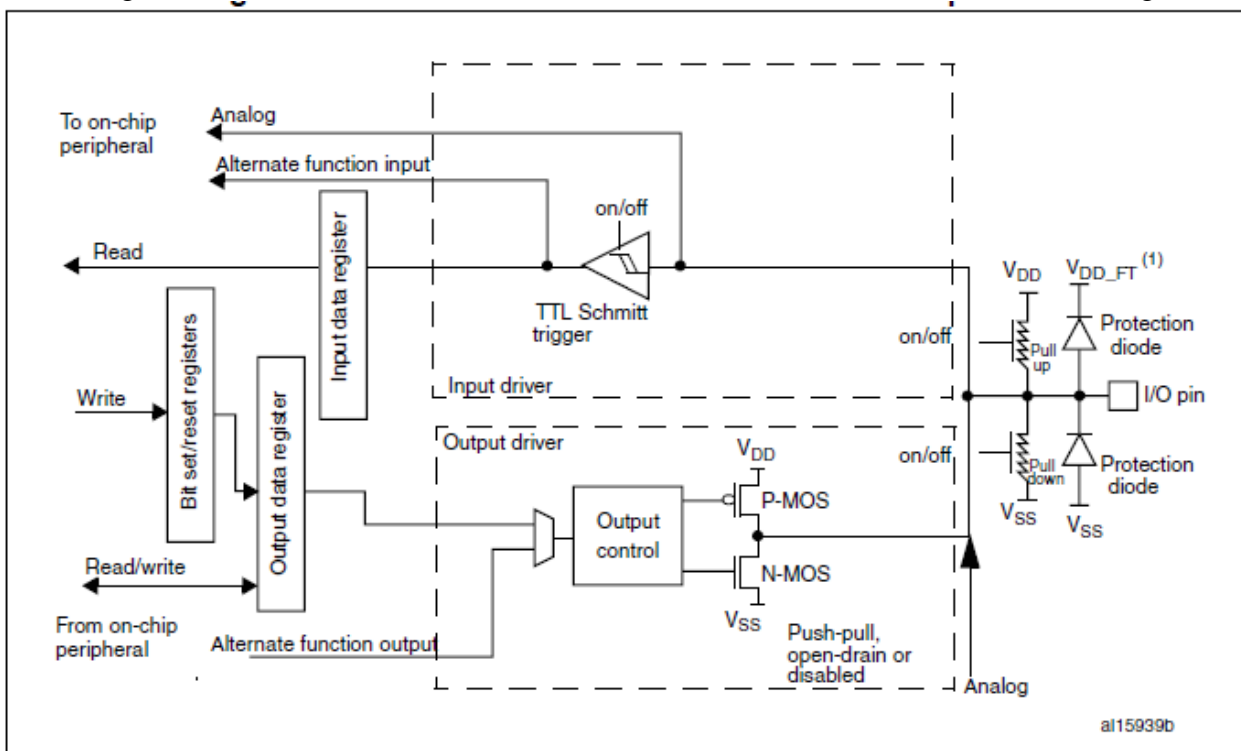
Die Prozessoren der Cortex- Controllerfamilie besitzen sehr flexibel einsetzbare Ports, die für viele Anwendungen benutzt werden können, z.B.

- Eingabe über Schalter und Taster
- Leuchtdiodenanschluss für Statusleuchten
- Transistoranschluss für Anwendungen, die höhere Ströme benötigen

Das folgende Bild zeigt den prinzipiellen Aufbau:

- GPIO Pins können als Ein- oder Ausgang konfiguriert werden
- GPIO Pins können auch “disabled” werden
- Eingabewerte können gelesen werden (0/1)
- Eingabewerte können auch als Interruptquelle genutzt werden
- GPIOs enthalten programmierbare Pull-Up oder Pull-Down Widerstände.

In der Regel sind GPIOs 5V tolerant, obwohl der uC nur mit 3,3V oder auch 1,8V versorgt wird



1. V<sub>DD\_FT</sub> is a potential specific to five-volt tolerant I/Os and different from V<sub>DD</sub>.

Mit dem integrierten Pull-Up Widerstand von etwa 50 kOhm wird dafür gesorgt, dass die Portpins nach RESET auf logisch "HIGH" liegen.

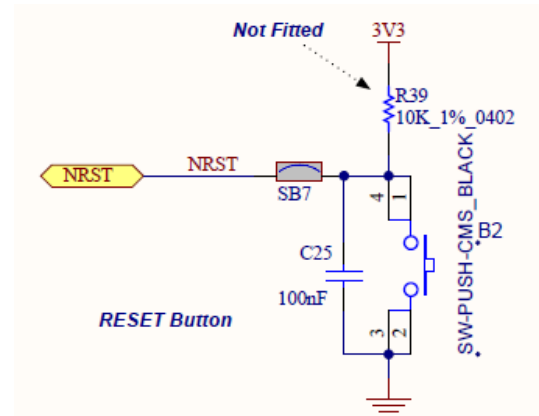
Eingabegeräte wie Taster, Schalter werden in der Regel zwischen dem Portpin und GND angeschlossen.

Bei der Ausgabe ist der FET im Low-Zustand durchgeschaltet. Der FET kann dabei etwa 20 mA Strom aufnehmen.

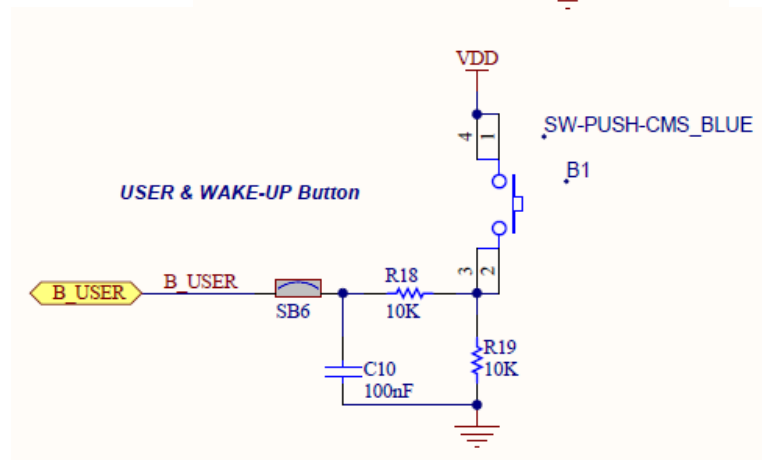
<sup>11</sup> General Purpose Input Output

## 10.6.1 Dateneingabe mit Schalter

Es gibt zwei Möglichkeiten einen Taster oder Schalter an den Mikrocontroller anzuschließen. Die GPIO Ports haben in der Regel nach dem Reset einen internen Pull-Up Widerstand eingeschaltet, deshalb werden Schalter häufig gegen Masse angeschlossen. Das Bild zeigt exemplarisch den RESET-Taster. Als User Button ist im Praktikumsboard das der Taster an PD.13 angeschlossen.



Es kann natürlich ein Taster auch gegen die Versorgungsspannung angeschlossen werden, dann ist aber ein Pull-Down Widerstand nötig und der interne Pull-Up Widerstand muss unbedingt abgeschaltet werden.



Das Register zur Einstellung des Pull-Up/Pull-Down Widerstands zeigt die folgende Tabelle des Referenzmanuals (STM32F0R072):

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
PUPDR15[1:0]	PUPDR14[1:0]	PUPDR13[1:0]	PUPDR12[1:0]	PUPDR11[1:0]	PUPDR10[1:0]	PUPDR9[1:0]	PUPDR8[1:0]								
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PUPDR7[1:0]	PUPDR6[1:0]	PUPDR5[1:0]	PUPDR4[1:0]	PUPDR3[1:0]	PUPDR2[1:0]	PUPDR1[1:0]	PUPDR0[1:0]								
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 2y+1:2y **PUPDRy[1:0]**: Port x configuration bits (y = 0..15)

These bits are written by software to configure the I/O pull-up or pull-down

00: No pull-up, pull-down

01: Pull-up

10: Pull-down

11: Reserved

In dem folgenden Code wird der Pull-down Widerstand für den Port PD.13 aktiviert:

```

GPIO->PUPDR &= ~(0x3 << 26);           // 0b11 Bits löschen ab Bit26
GPIO->PUPDR |= (0x2 << 26);             // 0b10 ab Bit 26, Pull-down aktivieren
    
```

Die folgenden Bezeichnungen sind üblich:

Active High	Taster gegen GND	Taster nicht betätigt = „0“ Taster betätigt = „1“
-------------	------------------	--

Active Low	Taster gegen VCC	Taster nicht betätigt = „1“ Taster betätigt = „0“
------------	------------------	--

// Beispiel Taster an Portpin PD.13 einlesen:

```
if (GPIO->IDR & (1<<13))
    Taster_nicht_gedrueckt();
else
    Taster_gedrueckt();
```

// oder dasselbe Ergebnis mit der negierten Abfrage:

```
if (!(GPIO->IDR & (1<<13)))
    Taster_gedrueckt();
else
    Taster_nicht_gedrueckt();
```

Häufig wird für das Einlesen von GPIO Pins auch eine Funktion programmiert. Die folgende Funktion gibt für den Fall eines „Schließers“ bei Betätigen des Tasters eine „1“ zurück, andernfalls eine „0“:

```
/*
 * \return:
 * 0      not pressed
 * 1      pressed
 */
uint8_t taster(void)
{
    return ((GPIO->IDR & (1<<13)) >> 13);
}
```

Einen typischen Aufruf der oben gezeigten Funktion taster() innerhalb einer FSM zeigt der folgende Code:

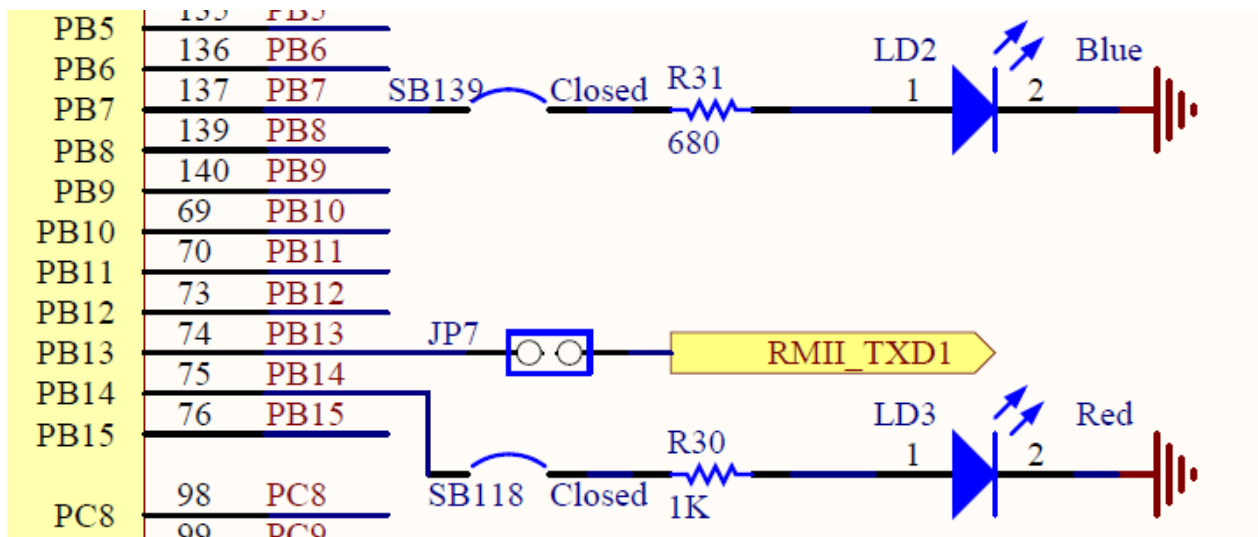
```
. . .
case START:
    if (taster())
        akt_state = RECHTSLAUF;           // wenn Taster gedrückt, dann....
    else
        akt_state = START;
    break;
. . .
```

Mit der STM32 CubeMX-HAL Bibliothek sieht dies einfacher aus:

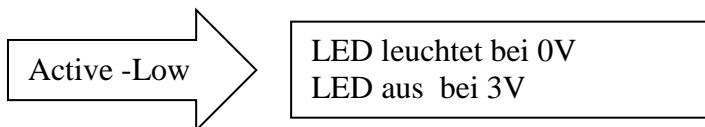
```
. . . .
__HAL_RCC_GPIO_CLK_ENABLE();           // ohne Takt geht nix!!!!
. . . .
if (HAL_GPIO_ReadPin(GPIO, GPIO_PIN_13) == GPIO_PIN_SET)
    Taster_nicht_gedrueckt();
else
    Taster_gedrueckt();
```

## 10.6.2 Datenausgabe mit LED

Das folgende Bild ist ein Ausschnitt aus einem Eval-Board (STM32 Nucleo-144 Pins) mit zwei LEDs:



Da ein Port-Pin im Low-Pegel (0V) wesentlich mehr Strom aufnehmen kann als er im High-Pegel liefern kann wird eine Last üblicherweise mit “low-active” betrieben.



// Beispiel LED Portpin Port B Bit 7 toggeln:

```
while (1){
    GPIOB->ODR |= (1<<7);
    GPIOB->ODR &= ~(1<<7);
}
```

// oder:

```
while (1){
    GPIOB->ODR ^= (1<<7);
}
```

Mit der STM32 CubeMX-HAL Bibliothek sieht dies so aus:

```
while (1){
    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_7, GPIO_PIN_SET);
    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_7, GPIO_PIN_RESET);
}
```

// oder:

```
while (1){  
    HAL_GPIO_TogglePin(GPIOB, GPIO_PIN_7);  
}
```

Ein Blinklicht für das Praktikumsboard mit der LED an Port A.5 würde mit der Cube-HAL Bibliothek so aussehen:

```
. . . . .  
  
HAL_Init();                      //ohne diese Funktion  
SystemClock_Config();  
  
__HAL_RCC_GPIOA_CLK_ENABLE();    //1)  
  
GPIO_InitTypeDef GPIO_InitStructure; //2)  
  
GPIO_InitStructure.Pin = GPIO_PIN_5; //3)  
GPIO_InitStructure.Mode = GPIO_MODE_OUTPUT_PP;  
GPIO_InitStructure.Pull = GPIO_PULLUP;  
GPIO_InitStructure.Speed = GPIO_SPEED_FREQ_LOW;  
HAL_GPIO_Init(GPIOB, &GPIO_InitStructure); //4)  
  
. . . . .  
while (1)  
{  
    HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5 , GPIO_PIN_SET); //5)  
    HAL_Delay(500); //6)  
    HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5 , GPIO_PIN_RESET); //6)  
    HAL_Delay(500);  
}  
. . . . .
```

- 1)  
Zuerst muss der GPIO Port A eingeschaltet werden.
- 2)  
Anstelle der Register muss nun eine Struktur beschrieben werden, dafür muss aber zuerst eine variable mit dem Datentyp dieser Struktur deklariert werden.
- 3)  
Der entsprechende Pin mit der Nummer 5 wird als Ausgang mit eingeschaltetem Pull-Up eingestellt.
- 4)  
Diese Funktion schreibt die eingestellten Parameter in die entsprechenden Register.
- 5)  
Diese Funktion setzt den angegebenen Pin oder löscht ihn.

6)

Eine Wartefunktion in Millisekunden wird von der Cube-HAL auch bereitgestellt. In dem Fall werden 500 Millisekunden gewartet

Der große Vorteil der HAL-Bibliothek zeigt sich erst bei den komplexeren Peripheriemodulen wie I2C-Bus usw.

So geht's auch:

```
while (1)
{
    HAL_GPIO_TogglePin(GPIO, GPIO_PIN_5);
    HAL_Delay(500);
}
```

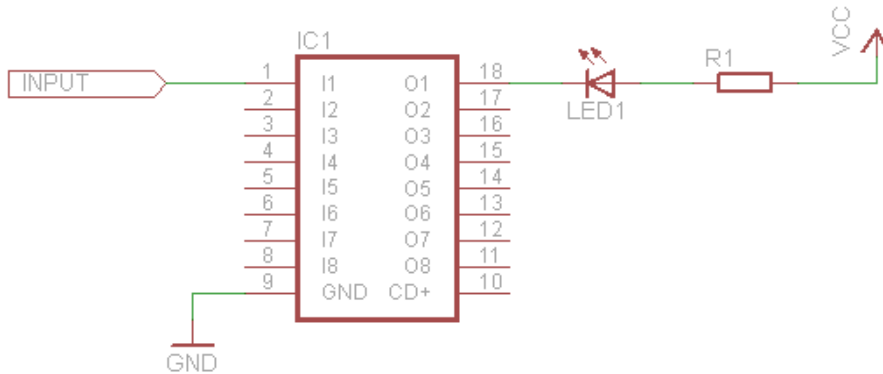
Diese Funktion toggelt den Pin zwischen 0 und 1

Als Richtlinie können low-current LEDs mit weniger als 8 mA direkt mit einem Vorwiderstand an den  $\mu$ Controller angeschlossen werden, für high-current LEDs muss ein Treibertransistor oder ein Treiber-IC verwendet werden.

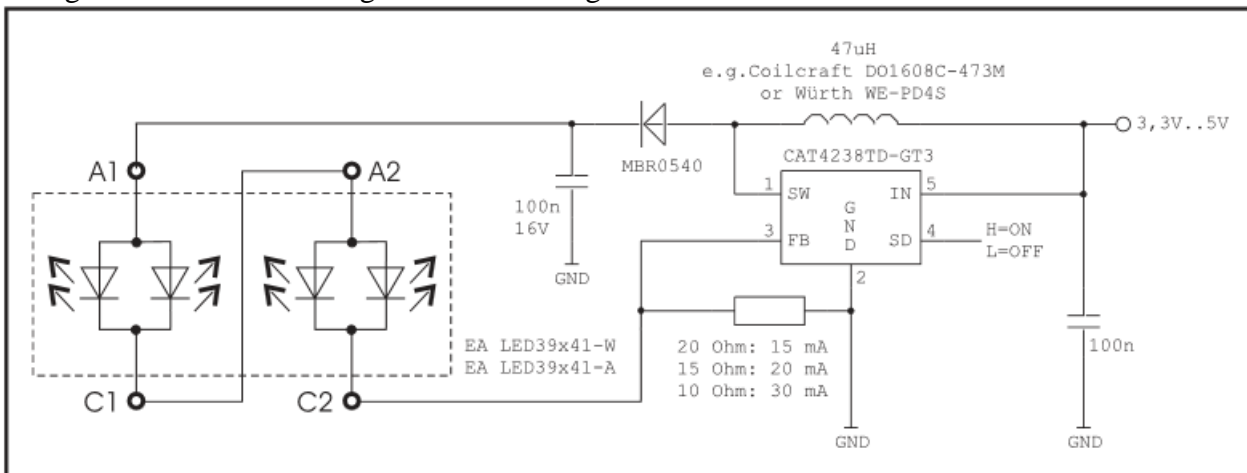


## 10.6.3 Treiberschaltungen für LEDs

Das Bild zeigt den Anschluss von LEDs über den Treiberbaustein ULN2803. Der ULN2803 ist ein NPN Darlington-Array bestehend aus 8 Darlington Transistorstufen samt nötigen Basiswiderständen. Dieser Baustein wird gerne genutzt um größere Lasten (bis ~500mA) an einem Mikrocontroller treiben zu können. Liegt an den Eingängen ein TTL Signal an, so schalten die Ausgänge nach Masse durch



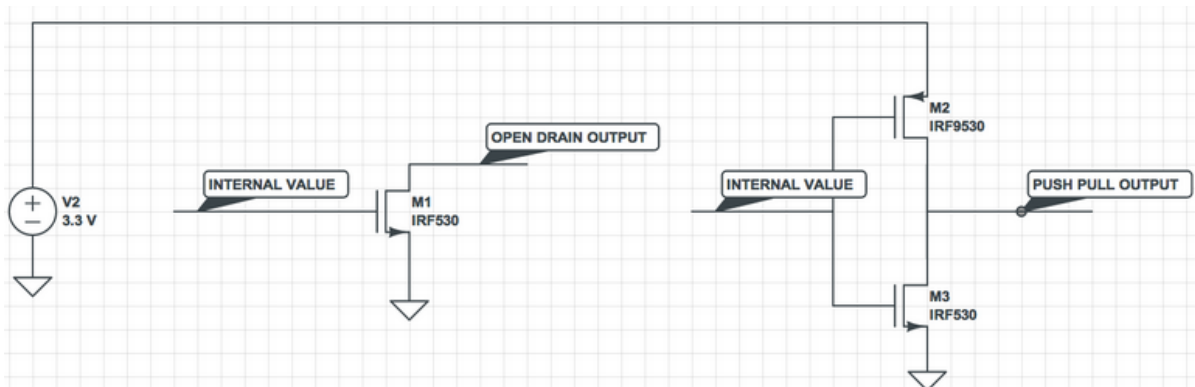
Für Hintergrundbeleuchtungen von LCD Anzeigen werden auch häufig folgende Treiberschaltungen eingesetzt. Das Bild zeigt die Schaltung für die Beleuchtung mit 4 LEDs<sup>12</sup>. Die LEDs werden hier mit Konstantstrom versorgt. Mit dem Pin 4 des Bausteins CAT4238 kann die Helligkeit über ein PWM Signal stufenlos eingestellt werden.



<sup>12</sup> Beispiel Hintergrundbeleuchtung für die EADOG Grafikdisplays von Electronic Assembly

## 10.6.4 Vergleich Open-Drain oder Push-Pull Ausgang

Die Ports können entweder im Push-Pull-Modus oder im Open-Drain-Modus betrieben werden.



Der „Open-Drain“-Ausgang besteht im Prinzip aus einem Transistorausgang mit fehlendem Drainwiderstand. Wenn der Drainwiderstand im Chip bereits integriert ist, entspricht dies prinzipiell einem Push-Pull-Ausgang

	Push-Pull	Open-Collector / Open-Drain
Setzen eines Port-Bits	am Port wird die Spannung 3V ausgegeben (am Port-Bit liegen 3V an)	Der Ausgang ist hochohmig geschaltet, mit einem Ohmmeter wird ein sehr hoher Widerstand gegen Masse gemessen (ca. 10 MOhm zwischen Port-Bit und GND)
Löschen eines Port-Bits	am Port wird die Spannung 0V ausgegeben (am Port-Bit liegen 0V an)	Der Ausgang ist kurzgeschlossen, mit einem Ohmmeter wird ein sehr kleiner Widerstand gegen Masse gemessen (ca. 10 mOhm zwischen Port-Bit und GND)

Der folgende Ausschnitt aus dem Referenzmanual zeigt das Register zur Einstellung des Open-Drain-Modus:

### 8.4.2 GPIO port output type register (GPIOx\_OTYPER) (x = A..F)

Address offset: 0x04

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OT15	OT14	OT13	OT12	OT11	OT10	OT9	OT8	OT7	OT6	OT5	OT4	OT3	OT2	OT1	OT0
r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w

Bits 31:16 Reserved, must be kept at reset value.

Bits 15:0 **OTy**: Port x configuration bits (y = 0..15)

These bits are written by software to configure the I/O output type.

0: Output push-pull (reset state)

1: Output open-drain

## Einführung in C für STM32

Um den Port Port-Bit PB.10 in den Open-Drain Mode zu schalten wird die folgende Anweisung benötigt:

```
GPIOB->OTYPER |= (1 << 10); // Open Drain Mode
```

Mit der Cube-HAL sieht die (vollständige) Einstellung des PB10 auf OpenDrain Ausgang so aus:

```
GPIO_InitTypeDef GPIO_InitStructure;

GPIO_InitStructure.Pin      = GPIO_PIN_10;
GPIO_InitStructure.Mode     = GPIO_MODE_OUTPUT_OD; // Open Drain Mode
GPIO_InitStructure.Pull     = GPIO_NOPULL;
HAL_GPIO_Init(GPIOB, &GPIO_InitStructure);
```

### 10.7 Timer

Alle Cortex Microcontroller besitzen in der Regel eine Vielzahl unterschiedlichster Timermodule, wobei alle Derivate von allen Herstellern den *SysTickTimer* und den *Repetitive Interrupt Timer (RIT)* auf dem Chip integriert haben. Der SysTickTimer ist hauptsächlich für den periodischen Interrupt eines Betriebssystems vorgesehen, während es der RIT Timer ermöglicht einen Timerinterrupt zu verwenden ohne die anderen Timer des Cortex zu verwenden.

#### 10.7.1 SysTickTimer

Der SysTick Timer ist ein einfacher Timer, der für Zeitverzögerungen und periodische Interrupts verwendet wird. Das Kernstück des Timers ist ein 24 Bit Abwärtszähler.

Hier wird die Verwendung des Timers mit der STM32CubeMX -HAL gezeigt. Die Initialisierung aller Taktgeber erfolgt wie bereits beschrieben in der Funktion `SystemClock_Config()`. Dies ist ein Ausschnitt aus dieser Funktion:

```
void SystemClock_Config(void){
    . . . . .

    /**Configure the SysTick interrupt time          */
    HAL_SYSTICK_Config(HAL_RCC_GetHCLKFreq()/1000); //1)

    /**Configure the SysTick                          */
    HAL_SYSTICK_CLKSourceConfig(SYSTICK_CLKSOURCE_HCLK); //2)
    . . . . .
}
```

- 1) Diese Funktion stellt die Anzahl an Ticks zwischen zwei Interrupts ein. In dem Beispiel wird der uC mit  $f=48\text{MHz}$  getaktet. Der Parameter ist der Reloadwert des Timers in Ticks:

$$\text{Reloadwert} = 48.000.000/1000 = 48.000$$

Ein Timetick dauert bei 48MHz exakt  $1/48\text{E}6$  Sekunden = 20,83 Mikrosekunden  
 $20,83 \text{ Mikrosekunden} * 48.000 = 0,001 \text{ Sekunden}$  bzw. eine Millisekunde.

Die Zeit zwischen 2 Interrupts wird also auf 1 Millisekunde eingestellt.

- 2) Diese Funktion stellt den Takt auf 48 MHz ein.

## 10.7.1.1 Wartefunktion ohne Cube-HAL

In der oben gezeigten Standardkonfiguration bietet sich der SysTicktimer für eine Wartefunktion mit Vielfachen von Millisekunden an:

```
//Globale Variablen für den SysTick-Handler

volatile uint32_t milliseconds = 0, seconds = 0;           //(1)
. . . . .

//Diese Funktion wird automatisch im SysTick-Handler aufgerufen
void HAL_SYSTICK_Callback(void){
    milliseconds++;                                         //(2)
    if (milliseconds%1000 == 999){                          //(3)
        seconds++;
    }
}

//Delay function for millisecond delay
void Delay_msec(uint32_t ms){
    volatile uint32_t MSStart = milliseconds;

    while((milliseconds - MSStart) < ms)
        asm volatile("__nop__");
}

//Delay function for second delay
void Delay_sec(uint32_t s){
    volatile uint32_t Ss = seconds;

    while((seconds - Ss) < s)
        asm volatile("__nop__");
}
```

- 1) Diese Variablen müssen global deklariert werden, da an eine ISR keine Variablen übergeben werden können.
- 2) Hier wird die Variable milliseconds inkrementiert.
- 3) Wenn 1000 Millisekunden vergangen sind wird die Variable seconds inkrementiert.

Die beiden Delayfunktionen sind ähnlich aufgebaut. In beiden Funktionen wird eine lokale Variable auf milliseconds bzw. seconds gesetzt. In einer while Schleife wird kontinuierlich abgefragt ob der aktuelle Wert von milliseconds bzw. seconds minus dem initialen Wert bei Eintritt in die Funktion kleiner ist als die vom Benutzer eingebene Verzögerungszeit.

## 10.7.1.2 Wartefunktion mit Cube-HAL

```
{
. . . . .
HAL_Delay(1000);           //Delay für 1 Sekunde bzw. 1000 Millisekunden
. . . . .
}
```

## 10.7.2 General Purpose Timer

Die STM32 Cortex Mikrocontroller verfügen über eine Vielzahl weiterer Timer, die aufgrund Ihrer vielfältigen Konfigurationsmöglichkeiten relativ kompliziert aufgebaut sind. Einen Überblick über die Timer des STM32F072 zeigt die folgende Tabelle:

**Table 7. Timer feature comparison**

Timer type	Timer	Counter resolution	Counter type	Prescaler factor	DMA request generation	Capture/compare channels	Complementary outputs
Advanced control	TIM1	16-bit	Up, down, up/down	integer from 1 to 65536	Yes	4	3
General purpose	TIM2	32-bit	Up, down, up/down	integer from 1 to 65536	Yes	4	-
	TIM3	16-bit	Up, down, up/down	integer from 1 to 65536	Yes	4	-
	TIM14	16-bit	Up	integer from 1 to 65536	No	1	-
	TIM15	16-bit	Up	integer from 1 to 65536	Yes	2	1
	TIM16 TIM17	16-bit	Up	integer from 1 to 65536	Yes	1	1
Basic	TIM6 TIM7	16-bit	Up	integer from 1 to 65536	Yes	-	-

Die Timer der weiteren STM32 Mikrocontroller sind ähnlich aufgebaut.

### 10.7.2.1 Taktquelle

Timer benötigen einen Takt, üblich sind zwei Taktquellen:

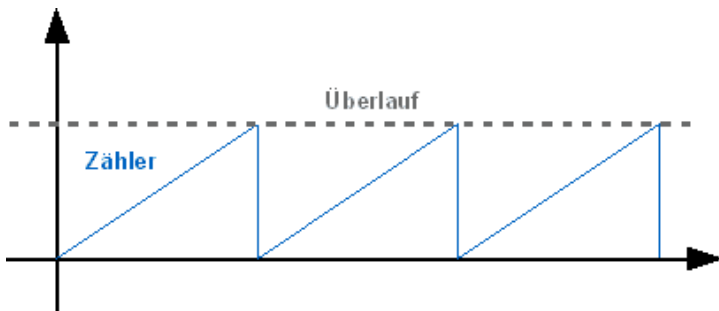
- Interner Takt, abgeleitet vom Prozessortakt
- Externer Takt, abgeleitet von der Ereignis an einem Pin

Dieser Takt wird üblicherweise durch einen Prescaler geteilt. Bei einem Takt des Mikro-Controllers mit 8 MHz ergibt ein Prescalerwert=7 einen Takt von 1MHz.

### 10.7.2.2 Basis

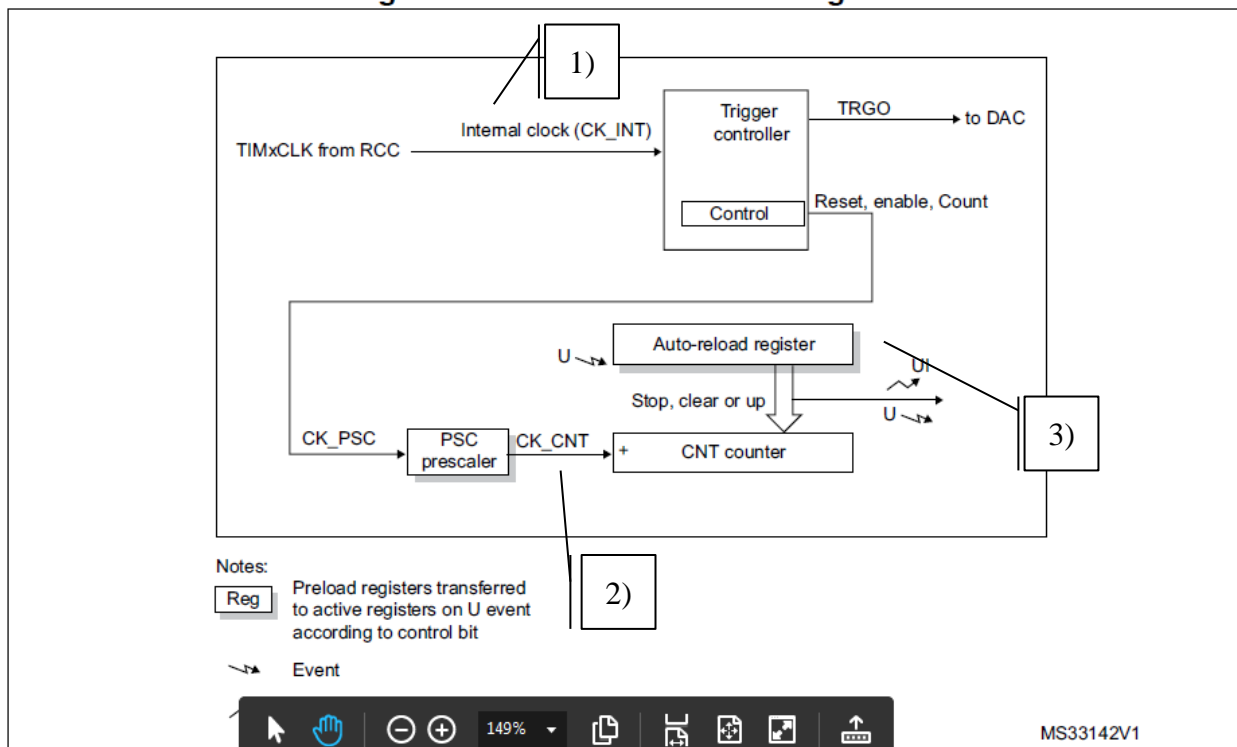
Timer sind prinzipiell Zähler mit 16 oder 32 Bit und zählen bis zu einem maximalen Wert. Üblich ist ein Reload oder ein Stoppen den Timers

Die einfachsten Timer sind die Timer 6 und 7, im Beispiel wird exemplarisch der Timer 6 betrachtet. Es handelt sich um einen 16-Bit Timer der ausgehend von Zählwert 0 aufwärts bis zu einem einstellbaren Maximalwert zählt:



Das Blockschaftbild zeigt die Komponenten dieses Timers:

**Figure 197. Basic timer block diagram**



Die rechteckigen Kästen repräsentieren die Register, z.B. *Control*, *PSC Prescaler* oder *Auto-reload Register* etc.

1)  
Der Timer wird gefüttert von CK\_INT, das ist der Takt der CPU, also entweder 8 oder 48 MHz.

2)  
Dieser Takt kann nun weiter geteilt werden. Mit dem resultierenden Timetick wird nun das Timer-Count-Register hochgezählt.

Beispiel:  $f_{cpu} = 48 \text{ MHz}$ , daraus folgt ein  $CK\_INT = 20,83\text{ns}$ .  
In das PSC Register wird nun der Wert  $48 - 1 = 47$  geschrieben.

$f_{cpu} / 48 = 1 \text{ MHz}$ , es ergibt sich also ein Timetick von 1 Mikrosekunde.

3)

## Einführung in C für STM32

Wenn das Timer-Count-Register den Wert des Autoreload-Registers erreicht, kann ein Interrupt ausgelöst werden oder der Timer neu geartet werden, je nach Konfiguration.

Beispiel:

Der maximale Wert ist  $2^{16} - 1 = 65535$ , also dauert es maximal  $65535 + 1 = 65,536$ ms bis zum Überlauf.

### 10.7.2.3 Ohne Cube-HAL

```
void delay_us(uint16_t us)
{
    TIM6->PSC = 47;          /* 48 MHz / 8 = 1 MHz */
    TIM6->ARR = us;           /* Autoreload Wert wird übergeben */
    TIM6->CR1 |= TIM_CR1_CEN; /* Timer starten */

    while (!(TIM6->SR & TIM_SR_UIF)); /* Warten auf Überlauf */
    /* Der Timer ist übergelaufen und Wartezeit abgelaufen */
}
```

### 10.7.2.4 Mit Cube-HAL

Eine Wartefunktion mit der STM32CubeMX-HAL sieht etwas anders aus. Sie müssen vorgegebene structs beschreiben. Hier ist die Wartefunktion für einen Systemtakt von  $f=8$  MHz.

```
void delay_us(uint32_t us) {
    __HAL_RCC_TIM6_CLK_ENABLE(); //1)

    htim6.Instance = TIM6;
    htim6.Init.Prescaler = 7; //2)
    htim6.Init.CounterMode = TIM_COUNTERMODE_UP;
    htim6.Init.Period = us; //3)
    htim6.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
    HAL_TIM_Base_Init(&htim6); //4)
    HAL_TIM_Base_Start(&htim6); //5)
    __HAL_TIM_CLEAR_FLAG(&htim6, TIM_FLAG_UPDATE); //6)
    while (__HAL_TIM_GET_FLAG(&htim6, TIM_FLAG_UPDATE) != SET); //7)
}
```

- 1) Takt einschalten ist der erste Schritt.
- 2) Prescaler = 7, also inkrementiert der Timer im Abstand von einer Mikrosekunde.

$$f_{timetick} = \frac{f_{Prozessortakt}}{Prescaler + 1}$$

$$T_{timetick} = T_{Prozessortakt} * (Prescaler + 1)$$

- 3) Wie viele Mikrosekunden soll der Timer laufen?
- 4) Die eingestellten Werte werden mit der Funktion HAL\_TIM\_Base\_Init in die Register geschrieben.
- 5) Der Timer wird gestartet.
- 6) Das Überlauf Flag muss gelöscht werden sonst ist es schon zu Beginn gesetzt!!!!

7) Hier wird solange gewartet bis der Timer übergelaufen ist.

Eine Wartefunktion in Millisekunden kann damit auch leicht programmiert werden:

```
void delay_ms(uint16_t time_ms)
{
    const uint16_t msec_in_usec = 1000;

    while (time_ms > 0)
    {
        delay_us(msec_in_usec);
        time_ms--;
    }
}
```

Beispiel Timer 6 Überlauf im Takt einer Millisekunde, der Timetick ist hierbei eine Mikrosekunde:

```
void Init_Timer6(void)
{
    htim6.Instance = TIM6;
    htim6.Init.Prescaler = 47;
    htim6.Init.CounterMode = TIM_COUNTERMODE_UP;
    htim6.Init.Period = 1000;
    if (HAL_TIM_Base_Init(&htim6) != HAL_OK){
        Error_Handler();
    }
}
```

Beispiel für längere Laufzeiten bis zum Überlauf eines 16 Bit Timers:

$$f_{\text{timetick}} = \frac{f_{\text{Prozessortakt}}}{\text{Prescaler} + 1} = \frac{47000 \text{ kHz}}{46999 + 1} = 1 \text{ kHz} .$$

Ein Prescaler von 46999 ergibt einen Timetick von einer Millisekunde.

$$\text{Timetick} = \frac{\text{Prescaler} + 1}{f_{\text{Prozessortakt}}} = \frac{46999 + 1}{47000 \text{ kHz}} = 1 \text{ ms}$$

Die Initialisierung des Timers 2 ist etwas komplizierter. Hier wird dieser 32 Bit Timer für eine Periode von 1 Sekunde konfiguriert:

```
TIM_HandleTypeDef htim2;

/* TIM2 init function */
static void TIMER2_Init(void)
{
    TIM_ClockConfigTypeDef sClockSourceConfig;
    TIM_MasterConfigTypeDef sMasterConfig;

    htim2.Instance = TIM2;
    htim2.Init.Prescaler = 47; // 1 us Timetick
    htim2.Init.CounterMode = TIM_COUNTERMODE_UP;
    htim2.Init.Period = 1E6; // Periode 1 Sekunde
    htim2.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
    htim2.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
```



```

if (HAL_TIM_Base_Init(&htim2) != HAL_OK)
{
    _Error_Handler(__FILE__, __LINE__);
}

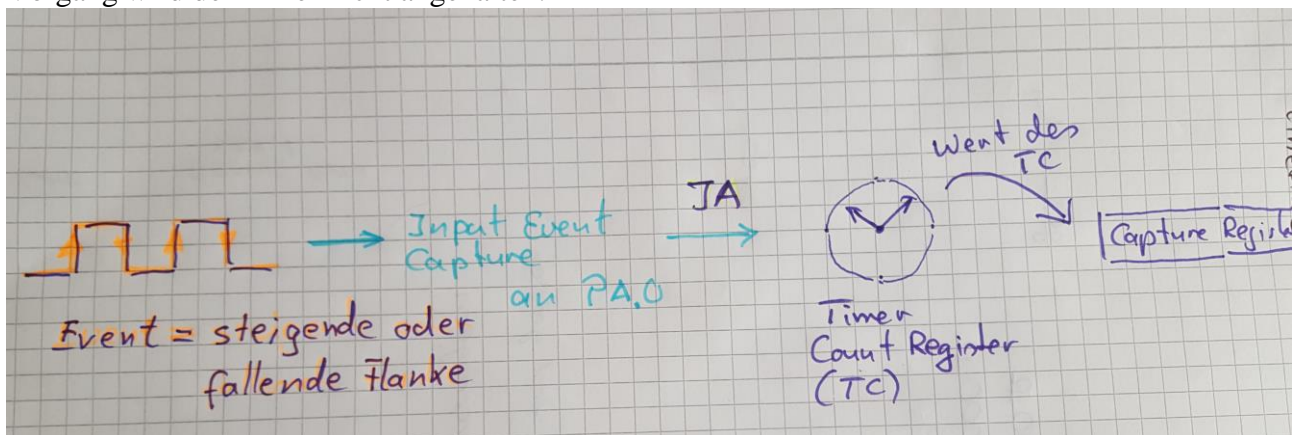
sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
if (HAL_TIM_ConfigClockSource(&htim2, &sClockSourceConfig) != HAL_OK)
{
    _Error_Handler(__FILE__, __LINE__);
}
}

```

## 10.7.2.5 Input-Capture

Die Timer werden häufig auch für Pulsweitenmessung, Frequenzmessung oder Drehzahlmessung eingesetzt. Für diese Aufgaben ist ein sogenannter Capture-Modus verfügbar.

In diesem Modus wird aufgrund eines externen Ereignisses, z.B. fallende Flanke an einem bestimmten Pins ein Schnappschuss des *TimerCounter-Registers* durchgeführt. Dabei werden die TimerCounter Register automatisch in sogenannte Capture Register geschrieben. Bei diesem Vorgang wird der Timer nicht angehalten.



Beispielhaft zeigt der folgende Code die Initialisierung für den Timer2 im Capturemodus und dem Captureeingang an dem Port PA.0 (Dies ist der Timer2 Kanal 1):

```

void TIMER2_IC_Init(void)
{
    GPIO_InitTypeDef G;
    TIM_HandleTypeDef htim2;
    TIM_ClockConfigTypeDef sClkConfig;
    TIM_IC_InitTypeDef sConfigIC;

    HAL_Init();
    SystemClock_Config();

    __HAL_RCC_GPIOA_CLK_ENABLE(); //Takt einschalten für die Peripherie
    /**TIM2 GPIO Configuration
    PA0 -----> TIM2_CH1
    */
    G.Pin = GPIO_PIN_0;
    G.Pull = GPIO_PULLUP;
    G.Mode = GPIO_MODE_AF_PP; //Alternativfunktion sonst geht nix
    G.Alternate = GPIO_AF2_TIM2; //1)
}

```

```
G.Speed = GPIO_SPEED_FREQ_HIGH;
HAL_GPIO_Init(GPIOA, &G);

//Initialisierung des Timers Nr 2 für den Input Capture Modus
__HAL_RCC_TIM2_CLK_ENABLE(); //Takt einschalten für die Peripherie

htim2.Instance = TIM2;
htim2.Init.Prescaler = 47; //2) ergibt f = 1MHz Timetick = 1us
htim2.Init.Period = 49999; //2) Periode des Timers 50ms = 20 Hz
htim2.Init.CounterMode = TIM_COUNTERMODE_UP;
HAL_TIM_IC_Init(&htim2); //3)

// Input Capture konfigurieren
sConfigIC.ICPolarity = TIM_ICPOLARITY_FALLING;
sConfigIC.ICSelection = TIM_ICSELECTION_DIRECTTI;
sConfigIC.ICPrescaler = TIM_ICPSC_DIV1;
sConfigIC.ICFilter = 0;
HAL_TIM_IC_ConfigChannel(&htim2, &sConfigIC, TIM_CHANNEL_1); // 5)

// Input Capture starten in Interrupt mode
HAL_TIM_IC_Start_IT(&htim2, TIM_CHANNEL_1); // 6)
HAL_TIM_Base_Start_IT(&htim2); // 7)
HAL_NVIC_EnableIRQ(TIM2_IRQn); // 8)
```

- 1) In den ersten Zeilen der Funktion wird der Pin für Input-Capture eingestellt. Im Datenbuch des uControllers findet man dazu die passende Alternativfunktion.

**Table 14. Alternate functions selected**

Pin name	AF0	AF1	AF2
PA0	-	USART2_CTS	TIM2_CH1_ETR
PA1	EVENTOUT	USART2_RTS	TIM2_CH2
PA2	TIM15_CH1	USART2_TX	TIM2_CH3
PA3	TIM15_CH2	USART2_RX	TIM2_CH4

- 2) Die meisten Einträge in diese struct sind selbsterklärend. Bei  $f_{CPU} = 48 \text{ MHz}$  erhält man mit `htim2.Init.Prescaler = 47` einen Timetick von 1 Mikrosekunde. Der Eintrag `htim2.Init.Period = 49999` stellt eine Periode des Timers von 50 Millisekunden ein.
- 3) Diese Funktion schreibt die Einstellungen in die Register. Damit ist der Timer2 als aufwärtslaufender Timer für Input-Capture mit Interruptverwendung und definierten Tick und Überlaufzeiten konfiguriert.
- 4) Die Taktversorgung des Timers kann intern oder extern erfolgen, hier wird der interne Systemtakt verwendet.
- 5) Hier wird nun der Input Capture Modus konfiguriert, der Timer soll den Capture auf die fallende Flanke auslösen.
- 6) Der Timer muss noch im IC Mode gestartet werden
- 7) Der Aufruf ist etwas eigenartig, aber notwendig um einen Überlauf-Interrupt des Timers auszulösen.

8) Zuletzt muss natürlich auch der Timer2 Interrupt eingeschaltet werden.

Die Capturewerte müssen nun in der Timerinterruptroutine in zwei Benutzervariablen geschrieben werden. Die Interruptfunktion für den Timer2 muss so aussehen:

```
void TIM2_IRQHandler(void) {  
    HAL_TIM_IRQHandler(&htim2);           //1)  
}
```

1)

Diese Funktion ist die Schlüsselfunktion für alle Ereignisse des Timers2. Für uns ist nur das Input Capture Ereignis relevant, also definieren wir die folgende Funktion, die exakt mit diesem Funktionsprototypen im HAL\_TIM\_IRQHandler aufgerufen wird:

```
void HAL_TIM_IC_CaptureCallback(TIM_HandleTypeDef *htim) {  
    switch (fmessState) {  
        case START:  
            inputCapture_A = HAL_TIM_ReadCapturedValue(htim, TIM_CHANNEL_1); //1)  
            overflow_cnt = 0;  
            update_Display = false;  
            fmessState = STOP;  
            break;  
        case STOP:  
            inputCapture_B = HAL_TIM_ReadCapturedValue(htim, TIM_CHANNEL_1); //1)  
            update_Display = true;                                           //2)  
            fmessState = START;  
            break;  
        default:  
            break;  
    }  
}
```

- 1) Diese Funktion schreibt die InputCapture-Werte in die Benutzervariablen
- 2) Die Berechnung soll im Hauptprogramm nur nach der vollständigen Messung aktualisiert werden.

Die Überläufe müssen natürlich auch gezählt werden:

```
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim) {  
    overflow_cnt++;  
}
```

Im Hauptprogramm werden anschließend die Ergebnisse aus den ISR ausgewertet und ausgegeben:

```
while (1) {  
    int32_t zw_Erg;  
    uint32_t pw_us;  
    double pw_s, frq_Hz;  
    char str[80];  
  
    if (update_Display == true) {  
        update_Display = false;  
        zw_Erg = inputCapture_B - inputCapture_A;    // 1)  
    }
```

```
if (zw_Erg >= 0) {
    pw_us = zw_Erg;                // erg = (B-A)
}
else{
    pw_us = zw_Erg + MAX_CNT;      // erg=(max-A)+B = (B-A)+max
}
pw_us = pw_us + (overflow_cnt * MAX_CNT); // 2)
pw_s = (double)pw_us / 1E6;        // Pulsweite in Sekunden
frq_Hz = 1 / pw_s;                // Frequenz in Hz
snprintf(str, sizeof(str),
    "\n\rpw_us= %d, Frequenz = %d.%d Hz\r\n", pw_us,
    (uint32_t) (frq_Hz * 1000) / 1000,
    (uint32_t) (frq_Hz * 1000) % 1000);
CLI_Write(str);
}
HAL_Delay(1);    // Wartefunktion in ms, basiert auf dem Systick Timer
}
```

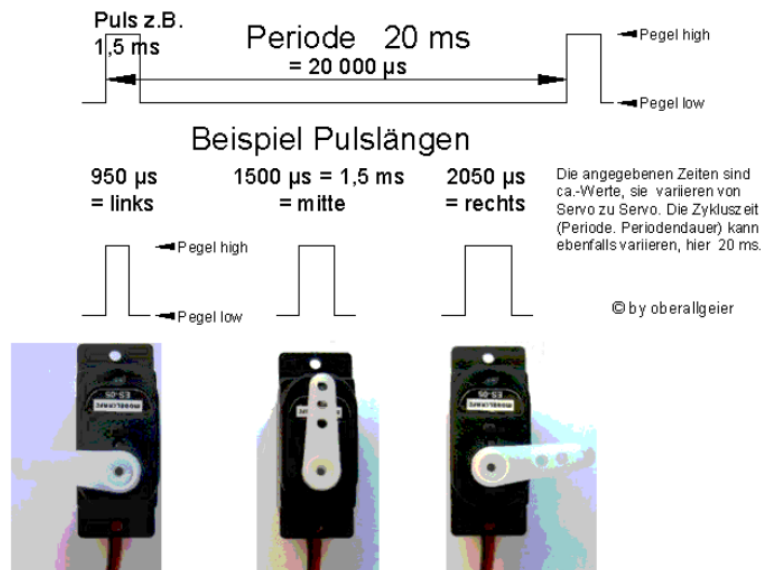
- 1) Für die Berechnung ist es notwendig zu unterscheiden ob das Ergebnis der Subtraktion positiv oder negativ ist.
- 2) Die Überläufe müssen natürlich mitgezählt werden und ergeben in dem Beispiel einen Wert in Mikrosekunden.

## 10.7.2.6 Output-Compare (PWM)

Eine häufige Aufgabe für Mikrocontroller ist die Erzeugung von PWM<sup>13</sup>-Signalen, zum Beispiel für Motorsteuerungen. Daher sind in den meisten Mikrocontrollern PWM-Einheiten als Hardware vorhanden. Sie sind in der Regel als Spezialfunktion mit den Timern verbunden und nutzen diese als Taktquelle. Der Anwender muss die entsprechenden PWM Register beschreiben um den PWM mit den gewünschten Einstellungen zu aktivieren. Der Rest läuft automatisch und unabhängig vom restlichen Programm, ohne den uC ständig zu beschäftigen wie bei einer PWM-Lösung in Software.

Eine einfache Anwendung ist das Ansteuern eines Modellbauservos:

Die Aufgabenstellung ist also eine Pulsweite zwischen ca. 1 – 2 ms bei einer PWM Frequenz von ca. 50 Hz.



Prinzipiell geht es um drei zusammenhängende Taktvorgaben:

- Systemtakt des Mikrocontrollers
- Wiederholfrequenz bzw. Periodendauer des PWM-Signals
- Pulsweite des PWM-Signals

Im Beispiel wird für diese Aufgabe der Timer 2 in Verbindung mit dem Port-Pin PA0 für Kanal1 konfiguriert. Zuerst muss der Pin PA0 konfiguriert werden:

```
static void GPIO_PWM_CH1_Init(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    __HAL_RCC_GPIOA_CLK_ENABLE(); //Takt einschalten für die Peripherie
    /**TIM2 GPIO Configuration
    PA0      -> TIM2_CH1
    */
    GPIO_InitStructure.Pin = GPIO_PIN_0;
    GPIO_InitStructure.Mode = GPIO_MODE_AF_PP;
    GPIO_InitStructure.Pull = GPIO_NOPULL;
    GPIO_InitStructure.Speed = GPIO_SPEED_FREQ_LOW;
    GPIO_InitStructure.Alternate = GPIO_AF2_TIM2;
    HAL_GPIO_Init(GPIOA, &GPIO_InitStructure); //1)
}
```

<sup>13</sup> Puls Weiten Modulation

1)

PA.0 ist der PWM-Ausgang für den Kanal1. Die einzustellende Alternativfunktion findet man im Datenbuch des Mikrocontrollers:

**Table 14. Alternate functions selected through GPIOA\_AFR registers for port A**

Pin name	AF0	AF1	AF2	AF3	AF4	AF5
PA0	-	USART2_CTS	TIM2_CH1_ETR	TSC_G1_IO1	USART4_TX	-
PA1	EVENTOUT	USART2_RTS	TIM2_CH2	TSC_G1_IO2	USART4_RX	TIM15_CH1N
PA2	TIM15_CH1	USART2_TX	TIM2_CH3	TSC_G1_IO3	-	-
PA3	TIM15_CH2	USART2_RX	TIM2_CH4	TSC_G1_IO4	-	-

Für den Kanal 4 müsste z.B. der Port-Pin PA3 auf die Alternativfunktion AF2 gestellt werden.

Nun wird der Timer konfiguriert. Dies ist vergleichsweise aufwendig, aber den vielfältigen Konfigurationsmöglichkeiten des Timers geschuldet:

```
static void TIM2_PWM_CH1_Init(void)
{
    TIM_HandleTypeDef htim2;
    TIM_ClockConfigTypeDef sClkConfig;
    TIM_MasterConfigTypeDef sMasterConfig;
    TIM_OC_InitTypeDef sConfigOC;

    __HAL_RCC_TIM2_CLK_ENABLE();           //Takt einschalten für die Peripherie

    htim2.Instance = TIM2;
    htim2.Init.Prescaler = 47;              //1)
    htim2.Init.CounterMode = TIM_COUNTERMODE_UP;
    htim2.Init.Period = 1000;               //2)

    htim2.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;           // default
    htim2.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_ENABLE; // default
    HAL_TIM_Base_Init(&htim2);                                     // 3)

    sClkConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;           // default
    HAL_TIM_ConfigClockSource(&htim2, &sClkConfig);              // default

    sConfigOC.OCMode = TIM_OCMode_PWM1;                          // 4)
    sConfigOC.Pulse = 1000;                                       // 5)
    sConfigOC.OCpolarity = TIM_OCPOLARITY_HIGH;
    sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
    HAL_TIM_PWM_ConfigChannel(&htim2, &sConfigOC, TIM_CHANNEL_2) // 6)
}
```

1)

Der eingestellte Prescaler von 47 ergibt bei  $f = 48 \text{ MHz}$  eine Taktrate des Timers von  $1 \text{ MHz}$  bzw. einen Timetick von  $T_{\text{timetick}} = 1 \text{ us}$ :

$$f_{\text{timetick}} = \frac{f_{\text{Prozessortakt}}}{\text{Prescaler} + 1}$$

## Einführung in C für STM32

$$T_{timetick} = T_{Prozessortakt} * (Prescaler + 1)$$

2)

Hier wird die Wiederholrate des PWM Signal eingestellt.

Für die Berechnung kann die folgende Formel verwendet werden:

$$f_{PWM} = \frac{f_{timetick}}{Reload + 1}$$

$$Reload = \frac{f_{timetick}}{f_{PWM}} - 1$$

Beispiel:

Eingestellt werden soll  $f_{PWM}=50\text{Hz}$

$f_{timetick} = 48 \text{ MHz}/48 = 1 \text{ MHz}$ . Siehe oben

Reloadwert =  $(1\text{MHz}/50\text{Hz}) - 1 = \text{htim2.Init.Period} = 19999$

Beachten müssen Sie dass es sich bei dem Timer des STM32F0 um einen 16 Bit Timer handelt, der maximale Reloadwert ist also 65535.

3)

Die Funktion schreibt die passenden Bits in die Register.

5)

Hier wird der Wert für die Pulsweite (`sConfig0C.Pulse`) eingetragen. Die Pulsweite kann mit der folgenden Formel berechnet werden:

$$\text{Pulsweite} = \frac{(Reload + 1) * dutycycle}{100} - 1$$

Beispiel:

Gegeben Duty cycle = 10%

Reload = `htim2.Init.Period` = 19999

Pulsweite = `sConfig0C.Pulse` = 1999

Weitere Beispiele:

25% duty cycle: Pulsweite =  $((19999 + 1) * 25) / 100 - 1 = 4999$

50% duty cycle: Pulsweite =  $((19999 + 1) * 50) / 100 - 1 = 9999$

75% duty cycle: Pulsweite =  $((19999 + 1) * 75) / 100 - 1 = 14999$

100% duty cycle: Pulsweite =  $((19999 + 1) * 100) / 100 - 1 = 19999$

Zurück zu dem Servo mit Pulsweiten zwischen 1,9ms und 2,2ms ergeben sich diese Minimal- und Maximalwerte durch einsetzen der Formel für den Duty cycle:

$$\text{Pulsweite} = \frac{(Reload + 1) * (T_{an})}{T_{PWM}} - 1$$

$T_{PWM} = 20\text{ms}$  bei einem  $f_{PWM}=50 \text{ Hz}$  (siehe oben).

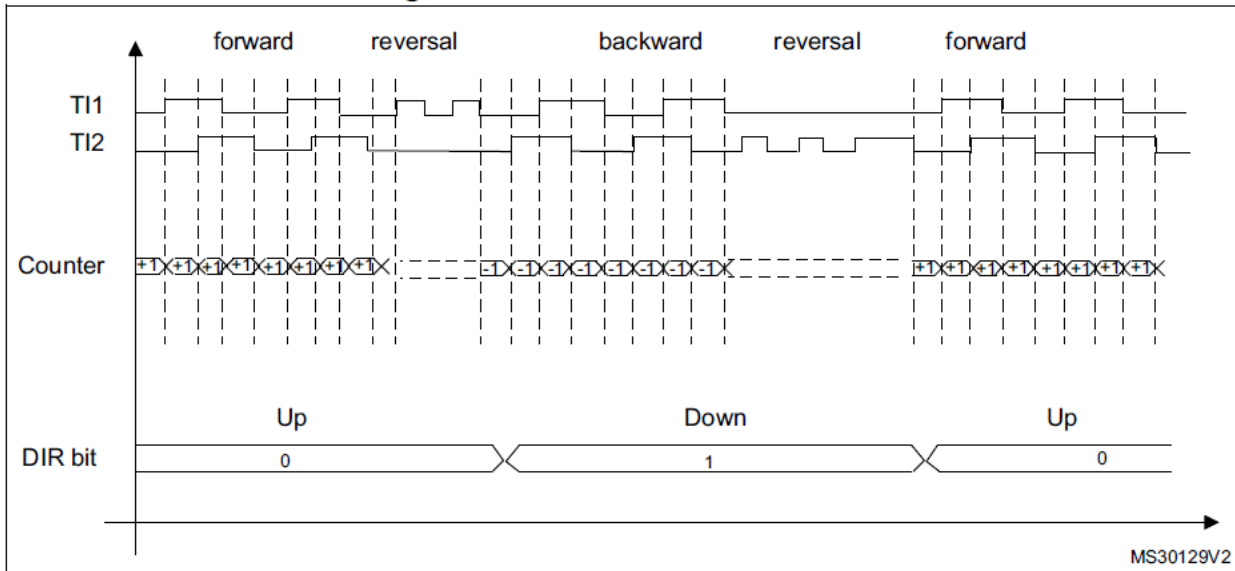
$T_{an}=1,9\text{ms}$ : Pulsweite =  $(20000*1,9)/20 - 1 = 1899$

$T_{an}=2,2\text{ms}$ : Pulsweite =  $(20000*2,2)/20 - 1 = 2199$

## 10.7.2.7 Encoder-Signale auswerten

Mit den Timern TIM1 und TIM2 können die Signale von Quadratur Encodern<sup>14</sup> ausgewertet werden.

**Figure 7. Position at X4 resolution**



The timer's counter is incremented or decremented for each transition on the selected input TI1 or TI2.

Es funktioniert allerdings nur mit den Inputsignalen der Channel1 und Channel2 der jeweiligen Timer. Die Timer müssen wie folgt konfiguriert werden:

```
//Timer 1 als Quadraturdecoder mit Ch1 und Ch2
__HAL_RCC_TIM1_CLK_ENABLE();

htim1.Instance = TIM1;
htim1.Init.Prescaler = 0;
htim1.Init.CounterMode = TIM_COUNTERMODE_UP;
htim1.Init.Period = 0;
htim1.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
htim1.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
sConfig.EncoderMode = TIM_ENCODERMODE_TI12; //1)
sConfig.IC1Polarity = TIM_ICPOLARITY_BOTHEDGE; //2)
sConfig.IC1Selection = TIM_ICSELECTION_DIRECTTI;
sConfig.IC1Prescaler = TIM_ICPSC_DIV1;
sConfig.IC1Filter = 0;
sConfig.IC2Polarity = TIM_ICPOLARITY_BOTHEDGE; //2)
sConfig.IC2Selection = TIM_ICSELECTION_DIRECTTI;
sConfig.IC2Prescaler = TIM_ICPSC_DIV1;
sConfig.IC2Filter = 0;
HAL_TIM_Encoder_Init(&htim1, &sConfig);
HAL_TIM_Base_Start(&htim1); //3)
```

<sup>14</sup> AN4013: STM32 cross-series timer overview



- 1) Es sollen beide Signale des Quadraturencoders ausgewertet werden.
- 2) Es soll sowohl die steigende Flanke als auch die fallende Flanke ausgewertet werden
- 3) Starten des Timers

Die gezählten Werte können in einer Interruptserviceroutine abgefragt werden:

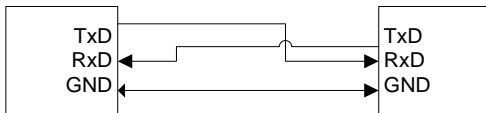
```
volatile int16_t g_wert;           //1)
. . .
void HAL_SYSTICK_Callback(void)
{
    g_wert = TIM1->CNT;
}
. . .
```

1. Die Variable g\_wert muss natürlich global deklariert werden.

## 10.8 UART

Diese serielle Schnittstelle ist in fast jedem Mikrocontroller enthalten und ein Relikt aus den Anfängen der Computerzeit. Aufgrund des einfachen Aufbaus und der weiten Verbreitung hat sich diese Schnittstelle bis heute gehalten.

UART bedeutet *Universal Asynchronous Receiver Transmitter*, es handelt sich also um eine asynchrone Schnittstelle, die das Senden und das Empfangen von seriellen Daten ermöglicht. Die Verbindung zweier Geräte mit UART Schnittstellen zeigt das folgende Bild:



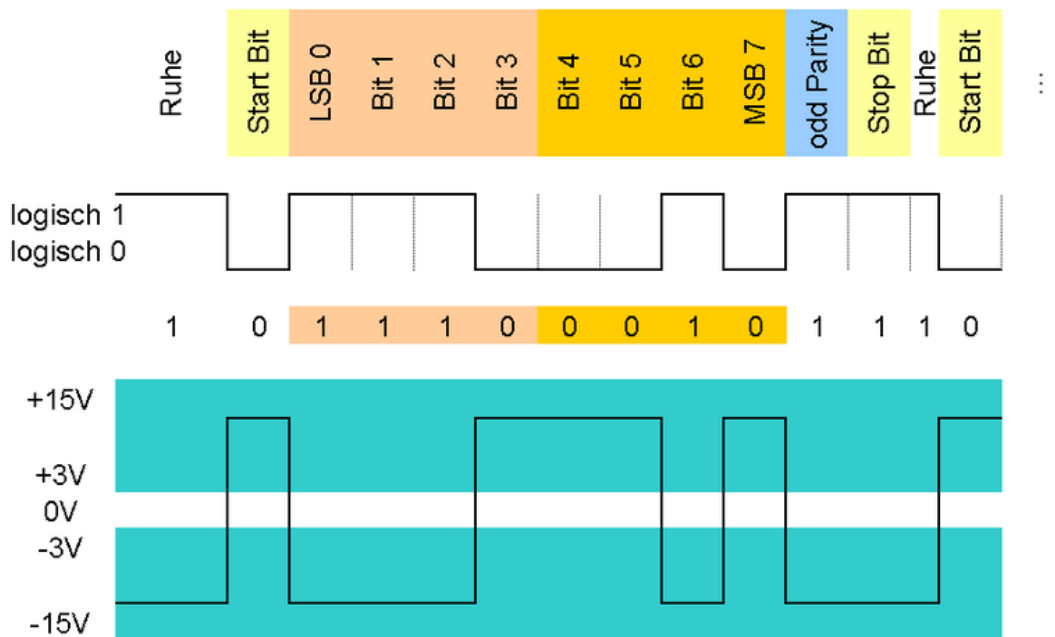
Die beiden Signalleitungen TxD und RxD müssen gekreuzt werden. Beide Signalspannungen werden gegen Masse gemessen und ausgewertet. Da sowohl für das Senden als auch das Empfangen von Daten eine eigene Leitung vorhanden ist, spricht man auch von Vollduplex. Standardisiert ist sowohl die Datenrate (Baud = bit/s) als auch der Datenrahmen. Üblich ist ein Datenrahmen mit 1 Startbit, 8 Datenbits und einem Stoppbit (8-n-1). Das Startbit kann nicht verändert werden, es gibt immer nur ein Startbit, wogegen es auch Geräte mit 1, 1,5 oder 2 Stoppbits gibt. Das (n) bedeutet *no parity*, es ist auch möglich ein gerades oder ungerades Paritätsbit hinzuzufügen. Ein Datenrahmen mit 8 Bit und 1 Stoppbit und geradem Paritätsbit würde dann mit 8-e-1 bezeichnet werden, mit ungeradem Paritätsbit mit 8-o-1. Das folgende Bild<sup>15</sup> zeigt an einem Beispiel (Übertragung von 0x47 = ASCII 'G') die zeitlichen Verläufe der Spannungen. Der obere Zeitverlauf zeigt das Signal des UART mit den Spannungspegeln des Mikrocontrollers, das untere Signal das zugehörige RS232 Signal.

---

<sup>15</sup> Wikipedia

Synchronisation  
 Daten low & high  
 Check

9600 8O1 = 9600 Baud; 8 Datenbits; odd Parity; 1 Stopbit  
 ASCII "G" = \$47 = 0100 0111



## 10.8.1 Senden mit Cube-HAL

Der folgende Code zeigt die Initialisierung und das Senden eines Strings mit der Cube-HAL Library:

```

GPIO_InitTypeDef GPIO_InitStructure; //1)
UART_HandleTypeDef huart2; //1)

//Pins für UART einstellen, aber zuerst den Takt enablen
__HAL_RCC_GPIOA_CLK_ENABLE();

/**USART2 GPIO Configuration
PA2 -----> USART2_TX
PA3 -----> USART2_RX
*/
GPIO_InitStructure.Pin = GPIO_PIN_2 | GPIO_PIN_3; //2)
GPIO_InitStructure.Mode = GPIO_MODE_AF_PP; //3)
GPIO_InitStructure.Pull = GPIO_NOPULL;
GPIO_InitStructure.Speed = GPIO_SPEED_FREQ_LOW;
GPIO_InitStructure.Alternate = GPIO_AF1_USART2; //4)

HAL_GPIO_Init(GPIOA, &GPIO_InitStructure);

//jetzt einstellen der UART Schnittstelle und den Takt enablen
__HAL_RCC_USART2_CLK_ENABLE();

huart2.Instance = USART2;
huart2.Init.BaudRate = 115200;
huart2.Init.WordLength = UART_WORDLENGTH_8B;
  
```

## Einführung in C für STM32

```
huart2.Init.StopBits = UART_STOPBITS_1;
huart2.Init.Parity = UART_PARITY_NONE;
huart2.Init.Mode = UART_MODE_TX_RX;
huart2.Init.HwFlowCtl = UART_HWCONTROL_NONE; //default
huart2.Init.OverSampling = UART_OVERSAMPLING_16; //default
huart2.Init.OneBitSampling = UART_ONE_BIT_SAMPLE_DISABLE; //default
huart2.AdvancedInit.AdvFeatureInit = UART_ADVFEATURE_NO_INIT; //default

HAL_UART_Init(&huart2); //5)
```

- 1) Diese Variablen werden in der Regel lokal deklariert. Für die Verwendung einer interruptgesteuerten Übertragung müssen sie global deklariert werden um sie auch in einer ISR benutzen zu können.
- 2) Die Pins findet man im Layout des Nucleo-Boards.
- 3) Die Pins werden nicht als GPIO verwendet, sondern in einer Alternativfunktion
- 4) Die Lösung liefert das Datenblatt des Mikrocontrollers:



Table 14. Alternate functions selected through GPIOA\_AFR registers for port A

Pin name	AF0	AF1	AF2	AF3	AF4	AF5	AF6	AF7
PA0	-	USART2_CTS	TIM2_CH1_ETR	TSC_G1_IO1	USART4_TX	-	-	COMP1_OUT
PA1	EVENTOUT	USART2_RTS	TIM2_CH2	TSC_G1_IO2	USART4_RX	TIM15_CH1N	-	-
PA2	TIM15_CH1	USART2_TX	TIM2_CH3	TSC_G1_IO3	-	-	-	COMP2_OUT
PA3	TIM15_CH2	USART2_RX	TIM2_CH4	TSC_G1_IO4	-	-	-	-

- 5) Diese Funktion schreibt die Werte der Struktur in die jeweiligen Register.

Das Senden ist dann recht einfach:

```
char str[] = "Hallo";
uint32_t timeout = 100;
HAL_UART_Transmit(huart2, str, sizeof(str), timeout);
```

### 10.8.2 Empfangen mit Cube-HAL, interruptgesteuert

Empfangen von Daten kann prinzipiell entweder über **polling** oder **interruptgesteuert** erfolgen. Bei **polling** muss der Rechner permanent die Daten abfragen und ist in der Regel geblockt bis Daten vorhanden sind. Da das Programm an der Programmzeile blockiert ist bis Zeichen ankommen ist das Verfahren nicht effektiv.

```
char str[80];
uint32_t timeout = 100;
HAL_UART_Receive(huart2, str, sizeof(str), timeout);
```

Bei einer **interruptgesteuerten** Bedienung der UART Schnittstelle ist der Ablauf ein wenig komplizierter.

Zunächst muss der entsprechende Interrupt freigeschaltet werden, im Beispiel ist diese der Interrupt für alle Ereignisse des UART Kanals2:

```
. . . . .
HAL_NVIC_EnableIRQ(USART2_IRQn);
__HAL_UART_ENABLE_IT(&huart2, UART_IT_RXNE); // Enable usart receive IRQ
. . . . .
```

Gezeigt werden zwei Möglichkeiten mit der Cube-HAL:

- 1) Behandlung des Interrupts komplett in der Funktion USART2\_IRQHandler (...). Die folgende Funktionsdefinition müssen Sie in die Datei **stm32f0xx\_it.c** einfügen:

```
// Datei stm32f0xx_it.c
. . . . .
extern uint8_t RxBuf[10]; // ist in der Applikation global deklariert
. . . . .

void USART2_IRQHandler(void)
{
    HAL_UART_IRQHandler(&huart2);
    HAL_UART_Receive_IT(&huart2, RxBuf, sizeof(RxBuf));
    HAL_UART_Transmit(&huart2, RxBuf, sizeof(RxBuf), 10);
}
```

In dem oben gezeigten Beispiel werden immer 10 Zeichen über das Terminal ausgegeben.

- 2) Die Original-ISR wird unverändert übernommen und eine von der Cube-HAL aufgerufene Callback-Funktion verwendet:

```
// Datei stm32f0xx_it.c
extern UART_HandleTypeDef huart2;          //ist in main deklariert (hoffentlich)

.....
void USART2_IRQHandler(void)
{
    HAL_UART_IRQHandler(&huart2);
}
```

Die Aufrufsequenz zeigt das folgende Bild:



Die Beispielapplikation zeigt den prinzipiellen Ablauf in main:

```
.....
HAL_UART_Receive_IT(&huart2, RxBuf, sizeof(RxBuf));          //nicht blockierend

if (flag == SET){
    HAL_UART_Transmit(&huart2, RxBuf, sizeof(RxBuf), 10);
    flag = RESET;
}
.....
```

Diese Funktion blockiert nicht!!!

Nachdem die eingestellte Anzahl von Charaktern empfangen wurde wird automatisch die folgende ISR aufgerufen:

```
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
{
    flag = SET;
}
```

## 10.9 I2C-Bus

Bei dem I2C-Bus handelt es sich um eine weit verbreitete serielle 2 Draht Schnittstelle für kurze Distanzen im cm Bereich. Weiterführende Literatur ist im Internet bzw. in den Vorlesungsunterlagen zu finden.

### 10.9.1 I2C-Master mit der HAL-Bibliothek

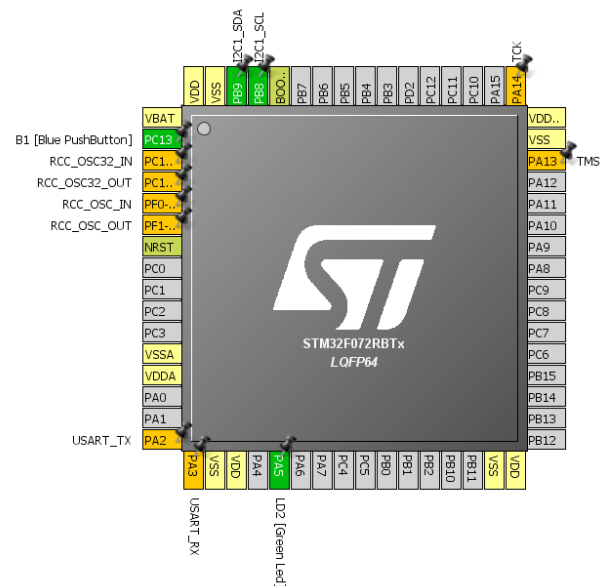
Mit den vorhandenen Funktionen des *Hardware Abstraction Layers* ist die Bedienung dieser Schnittstelle recht unkompliziert, zur Initialisierung müssen vorgegebene *structs* mit Leben erfüllt werden.

Im ersten Schritt müssen Sie den Takt für die benötigten GPIO-Pins einschalten. Im folgenden Code werden exemplarisch alle GPIOs eingeschaltet:

```
/* GPIO Ports Clock Enable */
__HAL_RCC_GPIOC_CLK_ENABLE();
__HAL_RCC_GPIOF_CLK_ENABLE();
__HAL_RCC_GPIOA_CLK_ENABLE();
__HAL_RCC_GPIOB_CLK_ENABLE();
```

Sie müssen nun zuerst die Pins für die Verwendung als I2C-Bus Signale SCL und SDA einstellen, hier werden die Pins PA8 und PA9 für den ersten I2C-Bus Kanal verwendet.

Wir verwenden dafür jetzt die von ST bereitgestellte HAL- Bibliothek. Sie müssen jetzt nicht mehr mit Bits setzen, löschen und verschieben, sondern Variablen des Datentyps *struct* beschreiben. Anschließend müssen Sie eine vorgegebene Funktion aufrufen die diese Werte dann in die Register schreibt. Das folgende Codeschnippel zeigt die Vorgehensweise:



```
GPIO_InitTypeDef G; //1)
```

```

/**I2C1 GPIO Configuration
PB8      -----> I2C1_SCL
PB9      -----> I2C1_SDA
*/
G.Pin = GPIO_PIN_8 | GPIO_PIN_9;
G.Mode = GPIO_MODE_AF_OD; //2)
G.Pull = GPIO_PULLUP;
G.Speed = GPIO_SPEED_FREQ_HIGH;
G.Alternate = GPIO_AF1_I2C1; //3)
HAL_GPIO_Init(GPIOB, &G);

```

Die einzelnen Zeilen sind nachfolgend erläutert:

1)

Zuerst müssen Sie eine Variable deklarieren mit dem passenden Datentyp, für die GPIO ist dies die Struktur **GPIO\_InitTypeDef**. Hier ist diese Variable aus Bequemlichkeitsgründen mit G bezeichnet. Sie finden diese struct in dem **stm32f0xx\_hal\_gpio.h**. Hier ist ein kleiner Ausschnitt aus dieser Datei abgebildet:

```
/**
 * @brief  GPIO Init structure definition
 */
typedef struct
{
    uint32_t Pin;           /*!< Specifies the GPIO pins to be configured.
                           This parameter can be any value of @ref GPIO_pins */

    uint32_t Mode;          /*!< Specifies the operating mode for the selected pins.
                           This parameter can be a value of @ref GPIO_mode */

    uint32_t Pull;          /*!< Pull-up or Pull-Down activation for the selected pins.
                           This parameter can be a value of @ref GPIO_pull */

    uint32_t Speed;         /*!< Specifies the speed for the selected pins.
                           This parameter can be a value of @ref GPIO_speed */

    uint32_t Alternate;     /*!< Peripheral to be connected to the selected pins
                           This parameter can be a value of @ref
                           GPIOEx_Alternate_function_selection */
}GPIO_InitTypeDef;
```

Die Bezeichner für GPIO\_pins, GPIO\_mode , GPIO\_pull und GPIO\_speed finden Sie in dem Headerfile **stm32f0xx\_hal\_gpio.h**.

Die Bezeichner für GPIOEx\_Alternate\_function\_selection finden Sie in dem Headerfile **stm32f0xx\_hal\_gpio\_ex.h**.

2)

Da die I2C-Bus Signale auf einer Open-Drain Konfiguration aufbauen müssen Sie für den Modus *Open Drain Mode* einstellen.

3)

Weshalb Sie die Alternativfunktion 1, also AF1 einstellen müssen geht aus der folgenden Tabelle hervor (Datenbuch stm32f072).



# Einführung in C für STM32

42/128

Table 15. Alternate functions selected through GPIOB\_AFR registers for port B

Pin name	AF0	AF1	AF2	AF3	AF4	AF5
PB0	EVENTOUT	TIM3_CH3	TIM1_CH2N	TSC_G3_IO2	USART3_CK	-
PB1	TIM14_CH1	TIM3_CH4	TIM1_CH3N	TSC_G3_IO3	USART3_RTS	-
PB2	-	-	-	TSC_G3_IO4	-	-
PB3	SPI1_SCK, I2S1_CK	EVENTOUT	TIM2_CH2	TSC_G5_IO1	-	-
PB4	SPI1_MISO, I2S1_MCK	TIM3_CH1	EVENTOUT	TSC_G5_IO2	-	TIM17_BKIN
PB5	SPI1_MOSI, I2S1_SD	TIM3_CH2	TIM16_BKIN	I2C1_SMBA	-	-
PB6	USART1_TX	I2C1_SCL	TIM16_CH1N	TSC_G5_IO3	-	-
PB7	USART1_RX	I2C1_SDA	TIM17_CH1N	TSC_G5_IO4	USART4_CTS	-
PB8	CEC	I2C1_SCL	TIM16_CH1	TSC_SYNC	CAN_RX	-
PB9	IR_OUT	I2C1_SDA	TIM17_CH1	EVENTOUT	CAN_TX	SPI2_NSS, I2S2_WS
PB10	CEC	I2C2_SCL	TIM2_CH3	TSC_SYNC	USART3_TX	SPI2_SCK, I2S2_CK
PB11	EVENTOUT	I2C2_SDA	TIM2_CH4	TSC_G6_IO1	USART3_RX	-
PB12	SPI2_NSS, I2S2_WS	EVENTOUT	TIM1_BKIN	TSC_G6_IO2	USART3_CK	TIM15_BKIN
PB13	SPI2_SCK, I2S2_CK	-	TIM1_CH1N	TSC_G6_IO3	USART3_CTS	I2C2_SCL
PB14	SPI2_MISO, I2S2_MCK	TIM15_CH1	TIM1_CH2N	TSC_G6_IO4	USART3_RTS	I2C2_SDA
PB15	SPI2_MOSI, I2S2_SD	TIM15_CH2	TIM1_CH3N	TIM15_CH1N	-	-

DocID025004 Rev 5

Nachdem die Pins eingestellt sind können Sie den I2C-Bus Kanal initialisieren. Zuerst muss der Takt dafür eingestellt werden, dies geschieht wieder wie bei den GPIO-Ports mit einem Makro.

```
/* I2C1 Clock Enable */  
__HAL_RCC_I2C1_CLK_ENABLE();
```

Die Einstellung des I2C-Bus geschieht nach demselben Rezept wie bei den GPIO-Pins. Es gibt eine vorgegebene Struktur und die Elemente dieser Struktur müssen Sie auffüllen.

```
I2C_HandleTypeDef i2c1; //1)  
  
i2c1.Instance = I2C1; //2)  
i2c1.Init.Timing = 0x2000090E; //3)  
i2c1.Init.OwnAddress1 = 0;  
i2c1.Init.AddressingMode = I2C_ADDRESSINGMODE_7BIT; //default  
i2c1.Init.DualAddressMode = I2C_DUALADDRESS_DISABLE; //default  
i2c1.Init.OwnAddress2 = 0; //default  
i2c1.Init.OwnAddress2Masks = I2C_OA2_NOMASK; //default  
i2c1.Init.GeneralCallMode = I2C_GENERALCALL_DISABLE; //default  
i2c1.Init.NoStretchMode = I2C_NOSTRETCH_DISABLE; //default  
HAL_I2C_Init(&i2c1); //4)
```

Die Default-Einträge können auch weggelassen werden.

- 1)  
In dem Beispiel wird die Variable `I2C_HandleTypeDef` mit `i2c1` deklariert.
- 2)  
Im Praktikum wird der Kanal 1 verwendet, also `I2C1`, Kanal 2 wäre dann `I2C2`.
- 3)

## Einführung in C für STM32

In dieser Struktur ist für unsere Praktikumsaufgabe lediglich der Eintrag für `i2c1.Init.Timing` wichtiger. Dafür gibt es glücklicherweise ein Excelsheet von ST zur Berechnung der Werte (siehe nächste Seite)

Die auskommentierten Zeilen sind für uns im Moment nicht wichtig, es sich um Defaultwerte handelt.

4)

Mit dieser Funktion werden die Werte in die entsprechenden Register geschrieben.

### 10.9.1.1 Senden

Nun können mit der von der HAL bereitgestellten Funktion `HAL_I2C_Master_Transmit(..)` beliebig viele Bytes gesendet werden:

```
#define SLAVE_ADD_7BIT 0x3F
#define SLAVE_ADD_8BIT 0x3F << 1

char TxBuf[] = "Hallo";
uint32_t timeout = 10;

HAL_I2C_Master_Transmit(&i2c1, SLAVE_ADD_8BIT, TxBuf, sizeof(TxBuf), timeout);
```

Achtung: An die Funktion muss die 8-Bitadresse übergeben werden --> `SLAVE_ADD_8BIT`.

Um das geringstwertige Bit der 8-Bitadresse muss man sich nicht kümmern, das erledigt die Funktion automatisch.

Diese Funktion bricht nach dem ersten fehlgeschlagenen Acknowledge ab.

### 10.9.1.2 Empfangen

Die Empfangsfunktion ist wie die Sendefunktion aufgebaut:

```
char RxBuf[10];
...
HAL_I2C_Master_Receive(&hi2c1, SLAVE_ADR_8BIT, RxBuf, 10, timeout);
```

Das Ergebnis wird in dem Feld abgelegt. Sie müssen lediglich darauf achten nicht mehr Bytes zu empfangen als das Feld groß ist. In dem Beispiel können Sie 10 Bytes empfangen

## Excel Sheet für die Berechnung der Taktfrequenz:

Beispiel für  $f_{\text{CPU}} = 8 \text{ MHz}$  und 100 kHz Takt für den I2C

**I2C Timing Configuration Tool for STM32F3xx and STM32F0xx devices V1.0.1** © COPYRIGHT STMicroelectronics MCD Application Team

**Please enter the Input parameters :**

Device Mode:

I2C Speed Mode:

I2C Speed Frequency (KHz):

I2C Clock Source Frequency (KHz):

For STM32F3xx devices, max clock frequency is 72 MHz  
For STM32F0xx devices, max clock frequency is 48 MHz

Analog Filter Delay:

Coefficient of Digital Filter:

Rise Time (ns):

Fall Time (ns):

**Output Result (Timing register):**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
PRESC				Reserved				SCLDEL				SDADEL			
0								2				0			
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SCLH								SCLL							
29								43							

TIMINGR register Value :  (Double Click to copy)

Error (%) :

Run Reset

**!!! This program work only if "macros" are enabled in "EXCEL" !!!**

2003 version :To enable "macros":>>> TOOLS >> MACRO >> SECURITY >> we recommend to set it to "medium" (you will be asked for macros to be executed)

2007-2010 version :To enable "macros":>>> Developer >> Macro Security >> MacroSettings>>Choose "Enable all macros"

Please active the "Developer" tab menu if is not active :>>> File>> Options >> Customize Ribbon >> Enable Developer tab menu

Beispiel für  $f_{\text{CPU}} = 48 \text{ MHz}$  und 100 kHz Takt für den I2C:

**I2C Timing Configuration Tool for STM32F3xx and STM32F0xx devices V1.0.1** © COPYRIGHT STMicroelectronics MCD Application Team

**Please enter the Input parameters :**

Device Mode:

I2C Speed Mode:

I2C Speed Frequency (KHz):

I2C Clock Source Frequency (KHz):

For STM32F3xx devices, max clock frequency is 72 MHz  
For STM32F0xx devices, max clock frequency is 48 MHz

Analog Filter Delay:

Coefficient of Digital Filter:

Rise Time (ns):

Fall Time (ns):

**Output Result (Timing register):**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
PRESC				Reserved				SCLDEL				SDADEL			
1								8				0			
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SCLH								SCLL							
94								137							

TIMINGR register Value :  (Double Click to copy)

Error (%) :

Run Reset

**!!! This program work only if "macros" are enabled in "EXCEL" !!!**

2003 version :To enable "macros":>>> TOOLS >> MACRO >> SECURITY >> we recommend to set it to "medium" (you will be asked for macros to be executed)

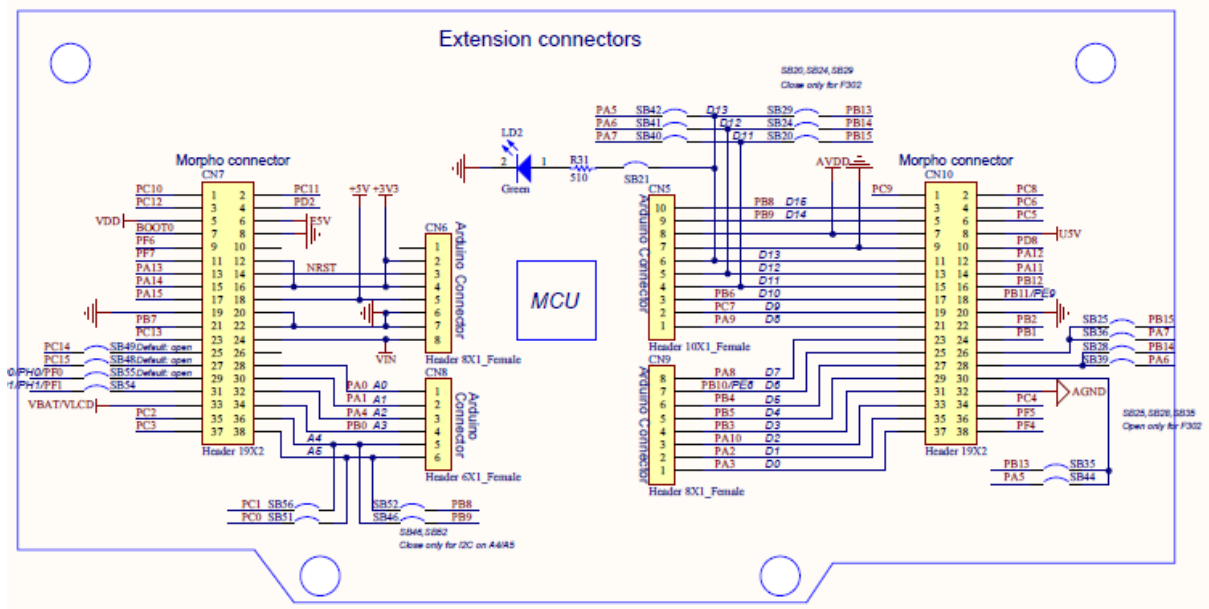
2007-2010 version :To enable "macros":>>> Developer >> Macro Security >> MacroSettings>>Choose "Enable all macros"

Please active the "Developer" tab menu if is not active :>>> File>> Options >> Customize Ribbon >> Enable Developer tab menu

## 10.10 SPI-Bus

Bei dem SPI-Bus handelt es sich um eine weit verbreitete serielle 4- Draht Schnittstelle für kurze Distanzen im cm Bereich. Der  $\mu$ C STM32 bietet zum einen eine Standard SPI Schnittstelle an, aber zusätzlich auch Erweiterungen für eine National Microwire TI-4Wire. Diese Schnittstelle ist auf eine Stiftleiste auf dem Praktikumsboard herausgeführt. Weiterführende Literatur ist im Internet bzw. in den Vorlesungsunterlagen zu finden.

Für die Pinzuordnung des exemplarisch gezeigten SPI Kanals 1 wird als Referenz das Nucleo-Board verwendet:



Die SPI Pins sind hier wie folgt herausgeführt:

Nucleo	SPI	Arduino
PA7	MOSI	D11
PA6	MISO	D12
PA5	SCK	D13
PB6	/CS	D10

Da hier PB6 nicht der hardwaremäßig vorgesehene Pin für diesen SPI Kanal ist wird er mithilfe von GPIO Kommandos softwaremäßig bedient.

### 10.10.1 SPI-Master mit der HAL-Bibliothek

```
int main(void)
{
    HAL_Init();           //1)
    SystemClock_Config(); //2)

    /* GPIO Ports Clock Enable */
    __HAL_RCC_GPIOA_CLK_ENABLE(); //3)
    __HAL_RCC_GPIOB_CLK_ENABLE();
}
```

## Einführung in C für STM32

1) und 2)

Diese beiden Funktionen müssen immer als Erstes im Programm aufgerufen werden.

3)

Diese Makro schaltet den Takt für die jeweiligen GPIO ein. Diese Makros müssen in unserem Fall für die Ports GPIOA und GPIOB aufgerufen werden. Dies muss vor der Einstellung der Pins geschehen!!

### 10.10.1.1 Pins einstellen

Die Einstellung der Portbits für den SPI zeigt der folgende Code:.

```
GPIO_InitTypeDef G; //1)

/**SPI1 GPIO Configuration
PA5  -----> SPI1_SCK
PA6  -----> SPI1_MISO
PA7  -----> SPI1_MOSI
*/
G.Pin = GPIO_PIN_5|GPIO_PIN_6|GPIO_PIN_7;
G.Mode = GPIO_MODE_AF_PP;
G.Pull = GPIO_NOPULL;
G.Speed = GPIO_SPEED_FREQ_HIGH;
G.Alternate = GPIO_AF0_SPI1; //2
HAL_GPIO_Init(GPIOA, &G); //3
```

1)

Zuerst müssen Sie eine Variable deklarieren mit dem passenden Datentyp, für die GPIO ist dies die Struktur **GPIO\_InitTypeDef**. Wie Sie die Variable benenn bleibt natürlich Ihnen überlassen, hier ist sie mit G deklariert.

2)

Welche Alternativfunktion Sie einstellen müssen geht aus einer Tabelle im User Manual hervor (Tabelle aus dem Datenbuch stm32f072). PA4 wird hier nicht genutzt, da /CS softwaremäßig eingestellt wird.

3)  
Mit dieser Funktion werden die Bits in die Register geschraubt.

Den Pin PB6 für das /CS müssen Sie softwaremäßig bedienen, also als einfacher GPIO Port Pin Ausgang:

DocID025004 Rev 5

41/128

Table 14. Alternate functions selected through GPIOA\_AFR registers for port A

Pin name	AF0	AF1	AF2	AF3	AF4	AF5
PA0	-	USART2_CTS	TIM2_CH1_ETR	TSC_G1_IO1	USART4_TX	-
PA1	EVENTOUT	USART2_RTS	TIM2_CH2	TSC_G1_IO2	USART4_RX	TIM15_CH1N
PA2	TIM15_CH1	USART2_TX	TIM2_CH3	TSC_G1_IO3	-	-
PA3	TIM15_CH2	USART2_RX	TIM2_CH4	TSC_G1_IO4	-	-
PA4	SPI1_NSS, I2S1_WS	USART2_CK	-	TSC_G2_IO1	TIM14_CH1	-
PA5	SPI1_SCK, I2S1_CK	CEC	TIM2_CH1_ETR	TSC_G2_IO2	-	-
PA6	SPI1_MISO, I2S1_MCK	TIM3_CH1	TIM1_BKIN	TSC_G2_IO3	USART3_CTS	TIM16_CH1
PA7	SPI1_MOSI, I2S1_SD	TIM3_CH2	TIM1_CH1N	TSC_G2_IO4	TIM14_CH1	TIM17_CH1
PA8	MCO	USART1_CK	TIM1_CH1	EVENTOUT	CRS_SYNC	-
PA9	TIM15_BKIN	USART1_TX	TIM1_CH2	TSC_G4_IO1	-	-
PA10	TIM17_BKIN	USART1_RX	TIM1_CH3	TSC_G4_IO2	-	-
PA11	EVENTOUT	USART1_CTS	TIM1_CH4	TSC_G4_IO3	CAN_RX	-
PA12	EVENTOUT	USART1_RTS	TIM1_ETR	TSC_G4_IO4	CAN_TX	-
PA13	SWDIO	IR_OUT	USB_NOE	-	-	-
PA14	SWCLK	USART2_TX	-	-	-	-
PA15	SPI1_NSS, I2S1_WS	USART2_RX	TIM2_CH1_ETR	EVENTOUT	USART4_RTS	-

```

/**SPI1 GPIO Configuration
    PB6 ----->CS
*/
G.Pin      =      GPIO_PIN_6;
G.Mode     =      GPIO_MODE_OUTPUT_PP;
G.Pull     =      GPIO_NOPULL;
G.Speed    =      GPIO_SPEED_FREQ_HIGH;

HAL_GPIO_Init(GPIOB, &G);

```

Dies geschieht nach dem Rezept für das Einstellen eines GPIO-Ports als Ausgang.

Damit sind die nötigen Pins für die Verwendung als SPI-Bus Signale eingestellt und Sie können den SPI-Bus konfigurieren

## 10.10.1.2 SPI-Bus konfigurieren

Der folgende Code zeigt die Initialisierung der SPI-Bus Schnittstelle:

```

__HAL_RCC_SPI1_CLK_ENABLE();          /* ohne Clock geht nix */

SPI_HandleTypeDef hspi1;                // 1)

hspi1.Instance = SPI1;                  // Kanal 1
hspi1.Init.Mode = SPI_MODE_MASTER;      // Master
hspi1.Init.Direction = SPI_DIRECTION_2LINES; // voll duplex
hspi1.Init.DataSize = SPI_DATASIZE_8BIT; // 8 Bit Übertragung

```

## Einführung in C für STM32

```
hspi1.Init.CLKPolarity = SPI_POLARITY_LOW;           // SCK in Ruhe Low
hspi1.Init.CLKPhase = SPI_PHASE_1EDGE;              // Daten werden bei der ersten
                                                    // Flanke von SCK übernommen
hspi1.Init.NSS = SPI_NSS_SOFT;                      // /CS softwaremäßig
hspi1.Init.BaudRatePrescaler = SPI_BAUDRATEPRESCALER_4; //2)
hspi1.Init.FirstBit = SPI_FIRSTBIT_MSB;
hspi1.Init.TTMode = SPI_TTMODE_DISABLE;
hspi1.Init.CRCCalculation = SPI_CRCCALCULATION_DISABLE;
hspi1.Init.CRCPolynomial = 7;
hspi1.Init.CRCLength = SPI_CRC_LENGTH_DATASIZE;
hspi1.Init.NSSPMODE = SPI_NSS_PULSE_ENABLE;

if (HAL_SPI_Init(&hspi1) != HAL_OK)
{
    Error_Handler();
}
```

- 1) Zuerst muss eine Variable des Datentyps struct deklariert werden, beim SPI-Bus ist dies die struct `SPI_HandleTypeDef`.
- 2) Hier wird die Taktrate des SPI Bus eingestellt. In dem Beispiel ist  $f_{\text{SPI}} = 48\text{MHz}/4 = 12\text{ MHz}$  eingestellt.

### 10.10.1.3 Senden und Empfangen

Das Senden und Empfangen der Daten ist nun einfach. Es wird hier exemplarisch an der Temperaturmessung bei einem TC72 Temperatursensor gezeigt. Nach dem Senden eines Kommandobytes (hier 0x02) liefert der Sensor 3 Bytes als Antwort. Hinter dem Aufruf von `ASSERT_CS()` verbergen sich 2 Makros:

```
/* Enable chip select TC72*/
#define ASSERT_CS() (HAL_GPIO_WritePin(GPIOB, GPIO_PIN_6, GPIO_PIN_SET))
/* Disable chip select */
#define DEASSERT_CS() (HAL_GPIO_WritePin(GPIOB, GPIO_PIN_6, GPIO_PIN_RESET))
```

Der Sensor benötigt im Gegensatz zu dem Standard SPI ein Chip-select mit active high, deshalb wird vor dem Sende- und Empfangsvorgang der Pin PB6 auf high gesetzt.

```
. . . . .
    TxBuf[0] = 0x02;
    TxBuf[1] = 0x00;
    TxBuf[2] = 0x00;
    TxBuf[3] = 0x00;
    ASSERT_CS();
    status = HAL_SPI_TransmitReceive(&hspi1, TxBuf, RxBuf, 4, timeout);
    DEASSERT_CS();
    temp_msb = RxBuf[1];
    temp_lsb = RxBuf[2];
    temp = (temp_msb << 8) | temp_lsb;
. . . . .
```

Bei diesem Sensor liefert das erste Byte das High-Byte und das zweite das Low-Byte. `RxBuf[0]` ist inhaltslos, das ja zuerst das Kommandobyte gesendet werden muss.

## 10.11 CAN-Bus

Der CAN-Bus wurde ursprünglich für die Automobilindustrie entwickelt und hat sich mittlerweile auch einen festen Platz in vielen weiteren Bereichen der Automatisierungstechnik oder Flugzeugtechnik erobert.

### 10.11.1 Konfigurieren

Für die Verwendung des CAN-Bus müssen wie bei anderen Schnittstellen auch zuerst die Pins eingestellt werden. Bei dem eingesetzten STM32F0 sind das die Port-pins PA11 und PA12.

```
/* ---- Pins einstellen ---- */
GPIO_InitTypeDef GPIO_InitStructure;           // 1)

__HAL_RCC_GPIOA_CLK_ENABLE();                  // 2)

/**CAN-Bus GPIO Configuration
PA11      -----> CAN-RX
PA12      -----> CAN-TX
*/
GPIO_InitStructure.Pin = GPIO_PIN_11 | GPIO_PIN_12;
GPIO_InitStructure.Mode = GPIO_MODE_AF_PP;
GPIO_InitStructure.Pull = GPIO_NOPULL;
GPIO_InitStructure.Speed = GPIO_SPEED_FREQ_HIGH;
GPIO_InitStructure.Alternate = GPIO_AF4_CAN;    // 3)

HAL_GPIO_Init(GPIOA, &GPIO_InitStructure);    // 4)
```

1)

Variablen sollen immer zu Beginn einer Funktion deklariert werden. Hier ist es eine Variable des Datentyps `GPIO_InitTypeDef`.

2)

Zuerst muss der Takt eingeschaltet werden

3)

Die Nummer der Alternativfunktion findet man im Datenblatt des Mikrocontrollers, in dem Beispiel ist die Alternativfunktion 4:





Table 14. Alternate functions selected through GPIOA\_AFR registers for port A

Pin name	AF0	AF1	AF2	AF3	AF4	AF5	AF6	AF7
PA0	-	USART2_CTS	TIM2_CH1_ETR	TSC_G1_IO1	USART4_TX	-	-	COMP1_OUT
PA1	EVENTOUT	USART2_RTS	TIM2_CH2	TSC_G1_IO2	USART4_RX	TIM15_CH1N	-	-
PA2	TIM15_CH1	USART2_TX	TIM2_CH3	TSC_G1_IO3	-	-	-	COMP2_OUT
PA3	TIM15_CH2	USART2_RX	TIM2_CH4	TSC_G1_IO4	-	-	-	-
PA4	SPI1_NSS, I2S1_WS	USART2_CK	-	TSC_G2_IO1	TIM14_CH1	-	-	-
PA5	SPI1_SCK, I2S1_CK	CEC	TIM2_CH1_ETR	TSC_G2_IO2	-	-	-	-
PA6	SPI1_MISO, I2S1_MCK	TIM3_CH1	TIM1_BKIN	TSC_G2_IO3	USART3_CTS	TIM16_CH1	EVENTOUT	COMP1_OUT
PA7	SPI1_MOSI, I2S1_SD	TIM3_CH2	TIM1_CH1N	TSC_G2_IO4	TIM14_CH1	TIM17_CH1	EVENTOUT	COMP2_OUT
PA8	MCO	USART1_CK	TIM1_CH1	EVENTOUT	CRS_SYNC	-	-	-
PA9	TIM15_BKIN	USART1_TX	TIM1_CH2	TSC_G4_IO1	-	-	-	-
PA10	TIM17_BKIN	USART1_RX	TIM1_CH3	TSC_G4_IO2	-	-	-	-
PA11	EVENTOUT	USART1_CTS	TIM1_CH4	TSC_G4_IO3	CAN_RX	-	-	COMP1_OUT
PA12	EVENTOUT	USART1_RTS	TIM1_ETR	TSC_G4_IO4	CAN_TX	-	-	COMP2_OUT
PA13	SWDIO	IR_OUT	USB_NOE	-	-	-	-	-
PA14	SWCLK	USART2_TX	-	-	-	-	-	-
PA15	SPI1_NSS, I2S1_WS	USART2_RX	TIM2_CH1_ETR	EVENTOUT	USART4_RTS	-	-	-

STM32F072x8 STM32F072xB

DocID025004 Rev 5

- 4)  
Diese Funktion schreibt die Einstellungen in die Register.

Nun kann der CAN-Bus konfiguriert werden:

```

__HAL_RCC_CAN1_CLK_ENABLE();           //1)

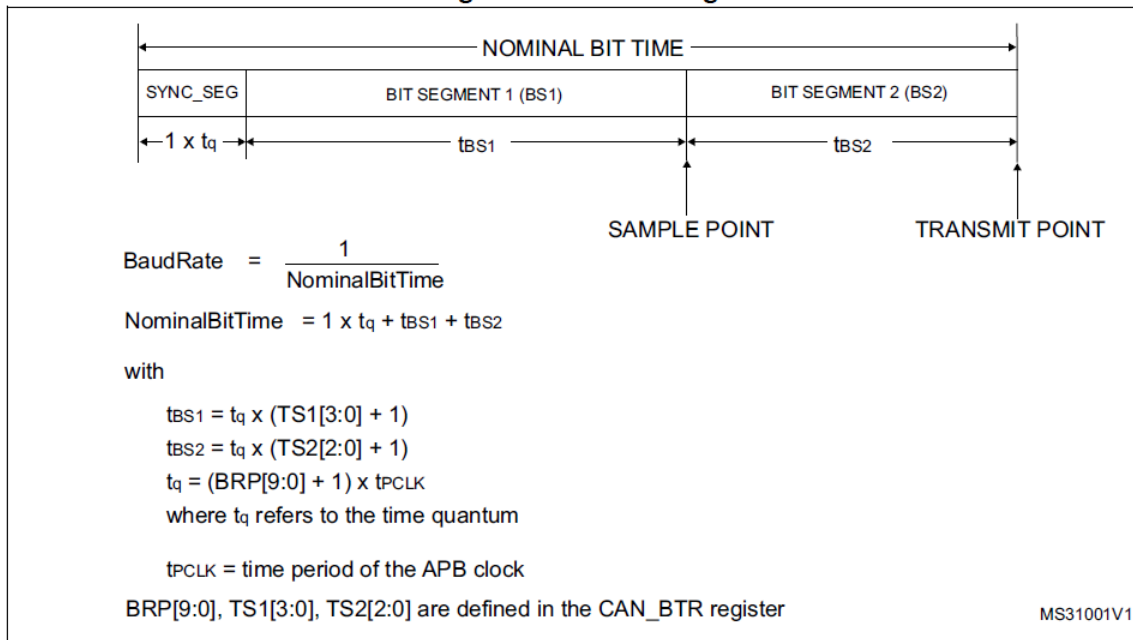
C.Instance = CAN;
C.Init.Mode = CAN_MODE_NORMAL;         //2)
C.Init.Prescaler = 24;                  //3)
C.Init.SJW = CAN_SJW_1TQ;              //3)
C.Init.BS1 = CAN_BS1_11TQ;             //3)
C.Init.BS2 = CAN_BS2_4TQ;              //3)
C.Init.TTCM = DISABLE;                  //4)
C.Init.ABOM = DISABLE;                  //4)
C.Init.AWUM = DISABLE;                  //4)
C.Init.NART = DISABLE;                  //4)
C.Init.RFLM = DISABLE;                  //4)
C.Init.TXFP = DISABLE;                  //4)

if (HAL_CAN_Init(&C) != HAL_OK){        //5)
    Error_Handler();
}

```

- 1)  
Takt einschalten wie bei allen Peripheriemodulen
- 2)  
Dieser Modus muss für eine CAN-Bus Übertragung eingestellt werden. Es gibt noch einige weitere Modi für das Testen des CAN-Busses, diese werden hier nicht behandelt.
- 3)  
Über diese Variablen wird die Bitrate des CAN-Busses eingestellt. Im folgenden Bild ist die Berechnung der Bitrate gezeigt:

**Figure 319. Bit timing**



Die Bitrate des CAN-Busses errechnet sich ausgehend vom APB- Clock. Für die Berechnung müssen drei Segmente von Zeitquanten berücksichtigt werden, die in folgender Reihenfolge auftreten, SYNC\_SEG (beim STM32 immer 1), BS1 und BS2. Für die Berechnung gibt es von STM bereits ein vorgefertigtes Excel Sheet:

				fPCLK= 48 MHz		-> tPCLK= 0.020833333 µs							
STM32F10x_STDPERIPH_LIB (3.4.0) CAN_InitStructure.				CAN_BTR			tq	tBS1	tBS2	CAN-Bit	Samplepoint	NominalBitTime	Baudrate
CAN_Prescaler=	CAN_BS1=	CAN_BS2=		BRP	TS1	TS2	[µs]	[µs]	[µs]		[%]	[µs]	[kbit/s]
2	CAN_BS1_16 tq	CAN_BS2_7 tq		1	15	6	0.0416667	0.666666667	0.291666667	25 tq	70.83	1	1000.00
4	CAN_BS1_11 tq	CAN_BS2_4 tq		3	10	3	0.0833333	0.916666667	0.333333333	19 tq	75.00	1.333333333	750.00
6	CAN_BS1_11 tq	CAN_BS2_4 tq		5	10	3	0.125	1.375	0.5	21 tq	75.00	2	500.00
12	CAN_BS1_11 tq	CAN_BS2_4 tq		11	10	3	0.25	2.75	1	27 tq	75.00	4	250.00
24	CAN_BS1_11 tq	CAN_BS2_4 tq		23	10	3	0.5	5.5	2	39 tq	75.00	8	125.00

Hier sind exemplarisch für eine  $f_{PCLK} = 48$  MHz die nötigen Einstellungen gezeigt.

4)

Hier können Sie die Defaultwerte übernehmen, bzw. können die Elemente der struct auch weglassen..

5)

Diese Funktion schreibt die Werte in die passenden Register des Mikrocontrollers.

## 10.11.2 Schreiben von Nachrichten

Der folgende Code zeigt das Schreiben von Daten über den CAN-Bus:

```

/*      ---- CAN-Bus senden      ---- */
static CanTxMsgTypeDef TxMessage;           //(1)
char TxBuf[] = "Hallo!";                   //(1)
uint32_t timeout_ms = 100;                  //(1)

C.pTxMsg = &TxMessage;                     //(2)

C.pTxMsg->DLC = sizeof(TxBuf);               //(3)
C.pTxMsg->IDE = CAN_ID_STD;
C.pTxMsg->RTR = CAN_RTR_DATA;
  
```

```
C.pTxMsg->StdId = 0x122;
for (int i=0;i<sizeof(TxBuf);i++)           //4)
    C.pTxMsg->Data[i] = TxBuf[i];

HAL_CAN_Transmit( &C, timeout_ms);         //5)
```

1)

Sie müssen zuerst die benötigten Variablen deklarieren. Eine Variable vom Datentyp `CanTxMsgTypeDef` und ein Feld für die zu sendenden Daten, im Beispiel ist das `TxBuf`. Diese Variablen deklarieren Sie bitte zu Beginn von `main()` oder der Funktion in der sie verwendet wird.

2)

Der Pointer `C.pTxMsg` zeigt nun auf die Adresse dieser Variable.

3)

Diese Codezeilen enthalten nun die zu sendende Nachricht, bestehend aus den Komponenten DLC, IDE, RTR und StdId. Anstelle der oben gezeigten Zeilen können Sie auch den folgenden Code schreiben:

```
//Beispiel 1: Größe von TxBuf mit Standardidentifizier 0x122 senden

TxMessage.DLC = sizeof(TxBuf);           //Länge der Daten in Byte
TxMessage.IDE = CAN_ID_STD;              //Standard Identifizier
TxMessage.RTR = CAN_RTR_DATA;           //Daten Frame
TxMessage.StdId = 0x122;                //Identifizier
for (int i=0; i< sizeof(TxBuf); i++)     //Daten in die struct schreiben
    TxMessage.Data[i] = TxBuf[i];

//Beispiel 2: 8 Bytes mit Extendedidentifizier 0x234 senden

TxMessage.DLC = 8;                      //Länge der Daten in Byte
TxMessage.IDE = CAN_ID_EXT;              //Extended Identifizier
TxMessage.RTR = CAN_RTR_DATA;           //Daten Frame
TxMessage.ExtId = 0x234;                 //Identifizier
for (int i=0; i<8; i++)                  //Daten in die struct schreiben
    TxMessage.Data[i] = TxBuf[i];

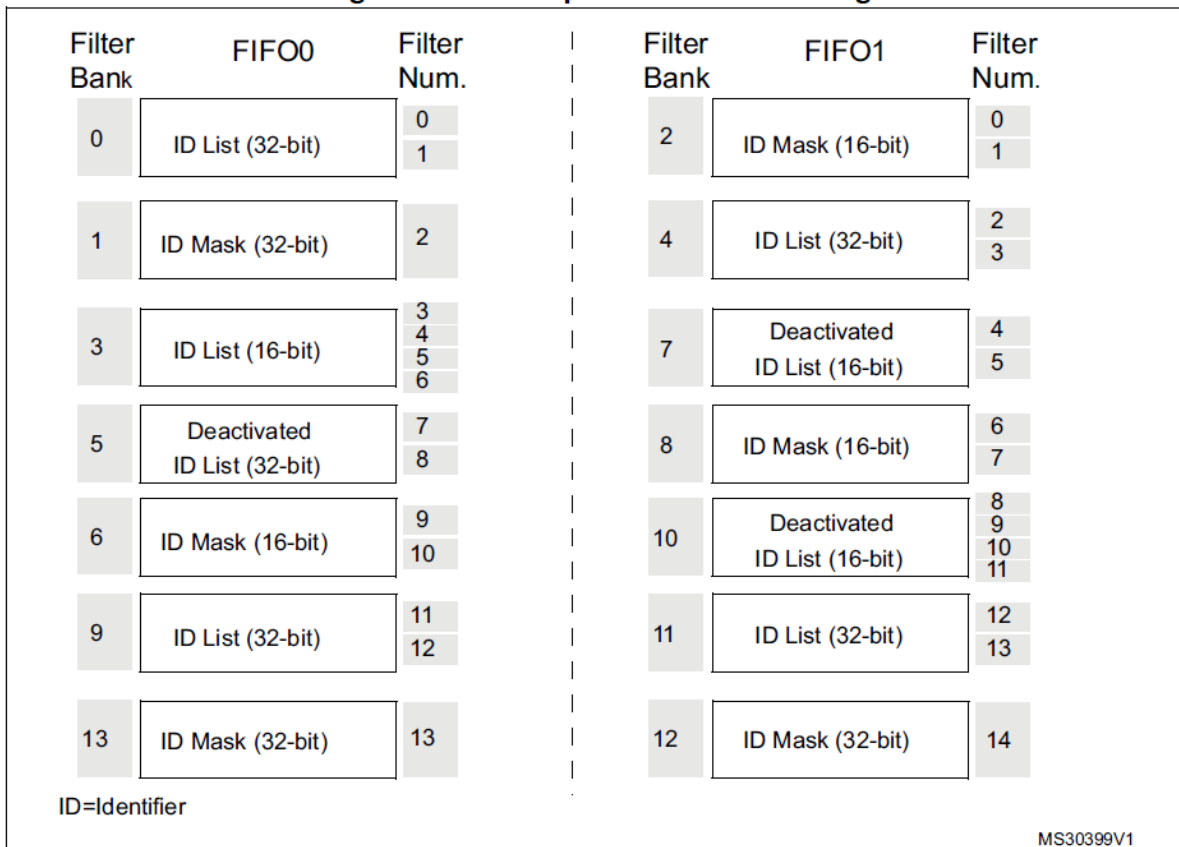
//Beispiel 3: Remote Frame senden mit Extendedidentifizier 0x1234 senden

TxMessage.DLC = 0;                      //Länge der Daten in Byte
TxMessage.IDE = CAN_ID_EXT;              //Extended Identifizier
TxMessage.RTR = CAN_RTR_REMOTE;         //Remote Frame
TxMessage.StdId = 0x1234;                //Identifizier
```

## 10.11.3 Akzeptanzfilterung von Nachrichten

Alle Can-Bus Nachrichten haben einen mit der Nachricht verbundenen Identifier. Der Empfänger kann auf Hardwarebasis entscheiden welche Identifier akzeptiert werden, dies erfolgt über 27 Filterbänke die jeweils ein 32 Bit Filter oder zwei 16 Bitfilter implementieren. Das Vorgehen bei der Verwendung hängt ab ob Sie Standard-, oder Extended Identifier verwenden.

**Figure 316. Example of filter numbering**



Exemplarisch wird im nachfolgend von Extended Identifier ausgegangen (29 Bit ID).

Ein Akzeptanzfilter basiert auf zwei 29 Bit Zahlen, bestehend aus einer `FilterId` und einer `FilterMask`. Es können zwei Modi eingestellt werden, der `IdList Mode` und der `Mask Mode`.

## 10.11.3.1 Mask-Mode

Die Filter Maske spezifiziert diejenigen Bits, die mit der Filter ID verglichen werden:  
Bei der Filtermaske lässt eine 0 das entsprechende Identifierbit passieren.

- Wenn ein Maskenbit auf Null gesetzt ist, wird das entsprechende ID-Bit unabhängig vom Wert des Filterbits automatisch akzeptiert.
- Wenn ein Maskenbit auf Eins gesetzt ist, wird das entsprechende ID-Bit mit dem Wert des Filterbits verglichen. Wenn sie übereinstimmen, wird dies akzeptiert, andernfalls wird der Rahmen abgelehnt.

Beispiele:

MASK = 0x1FFFFFFF und	ID = 0x00001567	für ID = 0x00001567
MASK = 0x1FFFFFFF und	ID = 0x00000008	für ID 8.
MASK = 0x1FFFFFFE und	ID = 0x00000002	für IDs 2 and 3.
MASK = 0x1FFFFFF8 und	ID = 0x00000000	für IDs 0 to 7.
MASK = 0x00000001 und	ID = 0x00000001	für alle ungeraden IDs.

Sie können 14 verschiedene Filter konfigurieren und haben zwei FIFO Speicher zur Verfügung

```
/*
 * Die folgende Funktion stellt das Akzeptanzfilter für FIFO0 ein.
 * Es werden in dieser Einstellung alle Identifier durchgelassen.
 */
void AcceptanceFilter(void)
{
    CAN_FilterConfTypeDef Config;

    Config.FilterNumber = 0;
    Config.FilterMode = CAN_FILTERMODE_IDMASK; // 1)
    Config.FilterScale = CAN_FILTERSCALE_32BIT;
    Config.FilterIdHigh = 0x0000;
    Config.FilterIdLow = 0x0000;
    Config.FilterMaskIdHigh = 0x0000;
    Config.FilterMaskIdLow = 0x0000;
    Config.FilterFIFOAssignment = CAN_FIFO0; // 2)
    Config.FilterActivation = ENABLE;
    Config.BankNumber = 0;
    HAL_CAN_ConfigFilter(&C, &Config);
}
```

Diese Konfiguration lässt alle Nachrichten passieren. Eine gute Beschreibung für das Einstellen der Akzeptanzfilter ist in einem sehr guten Tutorial von Diller Technologies zu finden<sup>16</sup>.

<sup>16</sup> <http://www.diller-technologies.de/stm32.html>

## 10.11.3.2 IDList-Mode

Die Einstellung um ausschließlich Nachrichten bestimmter Identifier zu erhalten zeigt der folgende Code am Beispiel, Sie müssen dafür den Filtermode IDLIST verwenden:

```
Config.FilterNumber = 0;
Config.FilterMode = CAN_FILTERMODE_IDLIST;
Config.FilterScale = CAN_FILTERSCALE_32BIT;

// CAN_FxR1 Konfiguration
Config.FilterIdHigh = (0x701<<5);
Config.FilterIdLow = 0x0000;
// CAN_FxR2 Konfiguration
Config.FilterMaskIdHigh = (0x581<<5);
Config.FilterMaskIdLow = 0x0000;

Config.FilterFIFOAssignment = CAN_FIFO0;
Config.FilterActivation = ENABLE;
if (HAL_CAN_ConfigFilter(&C, &Config) != HAL_OK){
    Error_Handler();
}
```

Eine Funktion um beliebige Identifier zu filtern zeigt die folgende Funktion mit einer variablen Argumentliste:

```
/* Akzeptanzfilter für FIFO0
 * Parameter sind die einzelnen IDs, die das Filter passieren dürfen
 * Alle 2 Filternummern muss die Filterbank inkrementiert werden
 * wenn nur FIFO0 verwendet wird, gibts bei 32Bit Filtern 14 Bänke und 28 Filter
 * Es werden nur 32 Bitfilter verwendet
 */
void AccFilterList(CAN_HandleTypeDef* hcan, uint32_t id0, ...) {
    CAN_FilterConfTypeDef Config;
    va_list args;
    uint32_t id = id0;
    uint8_t bank_nr = 0, filter_nr = 0;

    /* Parameterabfrage initialisieren */
    va_start(args, id0);

    Config.FilterMode = CAN_FILTERMODE_IDLIST;
    Config.FilterScale = CAN_FILTERSCALE_32BIT;
    Config.FilterFIFOAssignment = CAN_FIFO0;
    Config.FilterActivation = ENABLE;
    while (1) {
        if ((id == 0) || (filter_nr > 27) || (bank_nr > 28))
            break;
        Config.FilterNumber = filter_nr;
        Config.BankNumber = bank_nr;
        Config.FilterIdHigh = (id << 5);
        Config.FilterIdLow = 0x0000;
        if (HAL_CAN_ConfigFilter(hcan, &Config) != HAL_OK) {
            _Error_Handler(__FILE__, __LINE__);
        }
        if (filter_nr % 2) //wenn Rest
            bank_nr++;
        filter_nr++;

        /* Naechste id holen */
        id = (uint32_t) va_arg(args, uint32_t);
    }
    va_end(args);
}
```

### 10.11.4 Empfangen von Nachrichten - polling

Die Initialisierung ist identisch zu der für das Senden, zuerst die Pins und dann den CAN-Bus initialisieren.

Für die Nachrichten die gelesen werden sollen muss eine Variable des Datentyps `CanRxMsgTypeDef` global deklariert werden:

```
static CanRxMsgTypeDef RxMessage;           //Nachricht  
C.pRxMsg = &RxMessage;
```

In diese Variable wird die Nachricht geschrieben. Sie enthält nicht nur die empfangenen Daten, sondern wie beim Senden beschrieben DLC, StdId oder ExtId, Remote oder Daten Frame und den Identifier.

Nun können Sie Nachrichten empfangen. Den Funktionsaufruf für das Empfangen von Nachrichten zeigt der folgende Code:

```
while (HAL_CAN_Receive(&C, CAN_FIFO0, 5000) == HAL_BUSY) ;//Daten aus FIFO0
```

Die CAN-Bus Implementierung des STM32 besitzt 2 FIFO Speicher in denen die Nachrichten abgelegt werden. In dem gezeigten Code wird der erste FIFO Speicher verwendet.

Die Nachrichten können nun aus der Variablen `RxMessage` extrahiert werden, z.B.:

```
sprintf(str, "Id = %x, DLC = %x, Rx = %s\r\n", RxMessage.StdId,  
                                                RxMessage.DLC,  
                                                RxMessage.Data);
```



### 10.11.5 Empfangen von Nachrichten – interruptgesteuert

Die Initialisierung läuft genauso ab wie bei polling Betrieb bzw. beim Senden von Daten. Anschließend muss der entsprechende Interrupt freigegeben werden:

```
/*Initialisierung der Pins und des CAN-Bus abgeschlossen*/

HAL_NVIC_SetPriority(CEC_CAN_IRQn, 0, 0);
HAL_NVIC_EnableIRQ(CEC_CAN_IRQn);
```

Nun muss einmal die folgende Funktion in Ihrem Code aufgerufen werden:

```
/*Muss einmal aufgerufen werden um den Interruptempfang zu starten*/
if(HAL_CAN_Receive_IT(&C, CAN_FIFO0) != HAL_OK)
    Error_Handler();
```

Nun kommen die Interruptroutinen ins Spiel:

Der Name der folgenden ISR ist vom Startup-Code vorgeben und wird bei jedem Ereignis des CAN-Bus automatisch aufgerufen:

```
void CEC_CAN_IRQHandler(void)
{
    HAL_CAN_IRQHandler(&C);                //Diese Funktion liefert die Daten
    CLI_Write("Interrupt\r\n");
}
```

Die folgende Funktion wird automatisch vom IRQ-Handler am Ende aufgerufen:

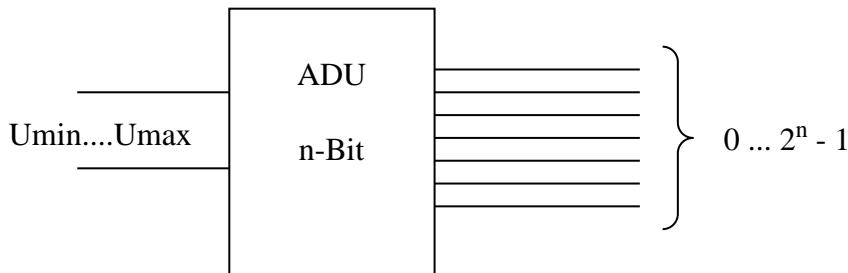
```
void HAL_CAN_RxCpltCallback(CAN_HandleTypeDef *CanHandle)
{
    char str[80];
    HAL_StatusTypeDef statusFIFO0, statusFIFO1;

    //sprintf(str, "Id = %x, Rx = %s\r\n", RxMessage.StdId, RxMessage.Data);
    sprintf(str, "Id = %x, Rx = %s\r\n",
        CanHandle->pRxMsg->StdId, CanHandle->pRxMsg->Data);
    CLI_Write(str);
    //Die folgende Funktion löscht die Interruptflags wieder
    statusFIFO0 = HAL_CAN_Receive_IT(&CanHandle, CAN_FIFO0);
    if (statusFIFO0 != HAL_OK)
        Error_Handler();
}
```

## 10.12 Analoge Schnittstellen

### 10.12.1 ADC-Grundlagen

Analog-Digital-Umsetzer, abgekürzt ADU (ADC, Analog-Digital-Converter engl.) sind Bauteile, die mit einem analogen Eingang ausgestattet sind und digitale Ausgangsleitungen besitzen. Ein am Eingang anliegendes analoges Signal wird in eine ganze Zahl umgerechnet und in binärer Darstellung an den Ausgangsleitungen ausgegeben.



**Bild 1:**  
**Analog-**  
**Digital-**  
**Umsetzer**

Man spricht bei dem ADU von einer Auflösung von n-Bit. Das analoge Eingangssignal  $U_e$  muss im Bereich zwischen  $U_{min}$  ...  $U_{max}$  liegen, die erzeugte Zahl  $Z$  ist dann in einem Bereich von  $0 \dots 2^n - 1$ . Ein (typischer) ADU mit 10 Bit Auflösung erzeugt beispielsweise Zahlen im Bereich von  $0 \dots 1023$ .

Die Formel für die Umsetzung lautet:

$$Z = \frac{U_e - U_{min}}{U_{max} - U_{min}} \cdot (2^n - 1)$$

Die Höhe der Quantisierungsstufen  $U_{LSB}$ , hängt von der Auflösung des ADU ab:

$$U_{LSB} = \frac{U_{max} - U_{min}}{2^n - 1}$$

Beispiel: An einem ADU mit 10 Bit Auflösung und einem Eingangsbereich von  $0 \dots 5V$  liegt ein analoges Signal von  $3,5V$  an. Die Umsetzung ergibt dann einen Digitalwert von  $Z = 3,5V / 5V \cdot 1023 = 717$ . Die Höhe der Quantisierungsstufen ist  $5V / 1023 = 4,89 \text{ mV}$ .

Alle ADU sind mit Fehlern behaftet. Prinzipiell unvermeidbar ist der **Quantisierungsfehler**, der durch die Rundung auf die ganze Zahl entsteht. Auch bei einem idealen ADU beträgt der Fehler bis zu  $0,5 U_{LSB}$ . Dazu kommen noch weitere Fehler durch Fertigungstoleranzen wie z.B.

Linearitätsfehler.

Die Zeit, die der ADU für die Wandlung einer Eingangsgröße benötigt, also die Wandlungszeit begrenzt die mögliche Zahl der Wandlungen pro Sekunde, die **Abtastfrequenz**.

Je nach dem Wandlungsprinzip des ADU sind Abtastfrequenzen bis in den MHz-Bereich möglich.

Unterschieden werden können die ADU nach den verschiedenen Umsetzverfahren.

Technik	Zahl der Schritte	Merkmale
Parallelverfahren	1	Sehr schnell, mittlere Genauigkeit, sehr aufwendig
Wägeverfahren (sukzessive Approximation)	$N = 1d N$	Mittlere Geschwindigkeit, mittlere Genauigkeit, S/H erforderlich
Zählverfahren	$N=2^n$	Langsam, hohe Auflösung und Genauigkeit, einfacher Aufbau

Viele Mikrocontroller haben einen ADU on-chip, in der Regel mit einer Auflösung zwischen 8 und 14 Bit. Meist wandeln Sie nach dem Verfahren der sukzessiven Approximation. Meist lässt sich der Eingangsspannungsbereich  $U_{\min}$  und  $U_{\max}$  in festgelegten Schritten programmieren. Damit kann der Eingangsspannungsbereich aufgespreizt und an das analoge Signal angepasst werden. Das ebenfalls erforderliche Abtast- und Halteglied ist auch meist on-chip, es sorgt dafür, dass das Signal während der Wandlung zwischengespeichert wird und konstant bleibt.

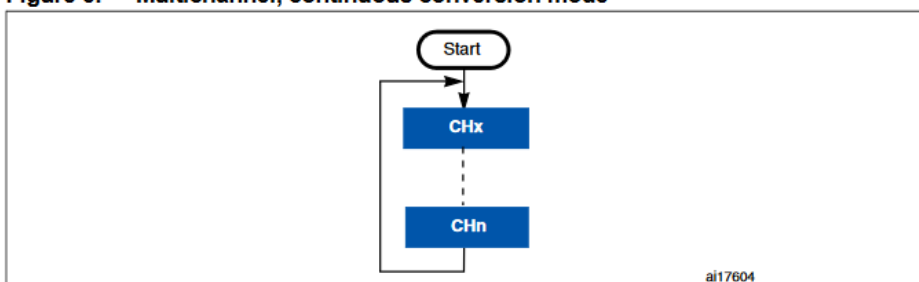
## 10.12.2 ADC- Implementierung im Mikrocontroller

Der ADC im STM32 bietet eine Software-Kalibrierung, die vor der Verwendung einmalig durchgeführt werden sollte (Verringerung von Offsetfehlern):

```
/*  
 * Kalibrieren des ADC  
 */  
HAL_ADCEx_Calibration_Start(&hadc);
```

Im folgenden Beispiel wird die ADC Funktionalität an einer einfachen kontinuierlichen Wandlung von 2 Kanälen gezeigt. Für eine Übersicht über die vielfältigen Möglichkeiten siehe AN3116<sup>17</sup>.

**Figure 5. Multichannel, continuous conversion mode**



Wie bei allen Peripheriemodulen muss zuerst für den oder die GPIO und den ADC der Takt eingeschaltet und die GPIO entsprechend konfiguriert werden. Wichtig ist, den analogen Pin mit `GPIO_MODE_ANALOG` einzustellen:

```
GPIO_InitTypeDef G;  
  
__HAL_RCC_GPIOA_CLK_ENABLE();           // ohne Takt geht nix!!!!  
__HAL_RCC_GPIOB_CLK_ENABLE();  
__HAL_RCC_GPIOC_CLK_ENABLE();  
  
__HAL_RCC_ADC1_CLK_ENABLE();  
  
/**ADC GPIO Configuration  
PA0      -----> ADC_IN0  
*/  
G.Pin = GPIO_PIN_0;  
G.Mode = GPIO_MODE_ANALOG;  
G.Pull = GPIO_NOPULL;  
HAL_GPIO_Init(GPIOA, &G);  
.  
.  
..
```

Der folgende Code zeigt eine Initialisierungsfunktion für den ADC des STM32 Mikrocontrollers.

```
/* ADC init function */  
static void MX_ADC_Init(void)  
{  
    ADC_HandleTypeDef hadc;  
    ADC_ChannelConfTypeDef sConfig;           //1)
```

<sup>17</sup> STM32™'s ADC modes and their applications

```

hadc.Instance = ADC1;
hadc.Init.ClockPrescaler = ADC_CLOCK_ASYNC_DIV1;           //default
hadc.Init.Resolution = ADC_RESOLUTION_12B;                 //default
hadc.Init.DataAlign = ADC_DATAALIGN_RIGHT;                 //default
hadc.Init.ScanConvMode = ADC_SCAN_DIRECTION_FORWARD;       //default
hadc.Init.EOCSelection = ADC_EOC_SINGLE_CONV;              //default
hadc.Init.LowPowerAutoWait = DISABLE;                       //default
hadc.Init.LowPowerAutoPowerOff = DISABLE;                  //default
hadc.Init.ContinuousConvMode = ENABLE;                      //2)
hadc.Init.DiscontinuousConvMode = DISABLE;                 //default
hadc.Init.ExternalTrigConv = ADC_SOFTWARE_START;           //2)
hadc.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_NONE;
hadc.Init.DMAContinuousRequests = DISABLE;
hadc.Init.Overrun = ADC_OVR_DATA_PRESERVED;
HAL_ADC_Init(&hadc);                                       //3)

/**Channel 1 */
sConfig.Channel = ADC_CHANNEL_1;                           //4)
sConfig.Rank = ADC_RANK_CHANNEL_NUMBER;
sConfig.SamplingTime = ADC_SAMPLETIME_1CYCLE_5;
HAL_ADC_ConfigChannel(&hadc, &sConfig);

/**Channel Temp-Sensor */
sConfig.Channel = ADC_CHANNEL_TEMPSENSOR;
HAL_ADC_ConfigChannel(&hadc, &sConfig);
}

```

1)

Wie bei allen Peripheriemodulen muss zuerst eine Variable mit einem beliebigen C konformen Namen deklariert werden.

2)

Die Wandlung kann auf vielerlei Arten gestartet werden. Im Beispiel erfolgt die Wandlung durch einen Softwarebefehl. Andere Möglichkeiten sind Timer-, Interrupt- oder DMA-Ereignisse. Der Continuous Mode startet die Wandlung jedesmal automatisch neu.

3)

Diese Funktion schreibt die Konfiguration in die Register.

4)

Exemplarisch wird hier der Kanal1 und der interne Temperatursensor verwendet

Ein (primitives) Codeschnipsel für das Starten der Wandlung und Lesen des Wandlungsergebnisses ist hier gezeigt:

```

//starten der wandlung ( muss nur einmal erfolgen, da continuous Mode eingestellt ist)
HAL_ADC_Start(&hadc);

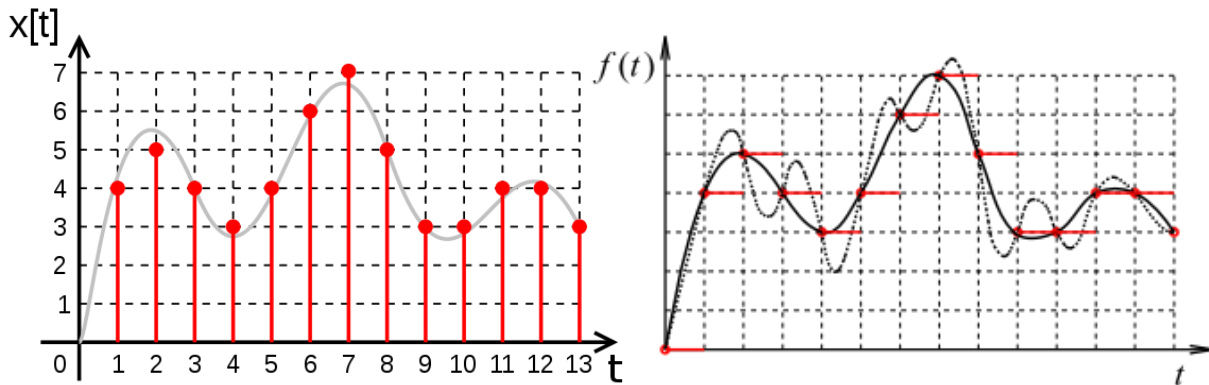
adc_val = HAL_ADC_GetValue(&hadc);

```

#####

## 10.12.3 DAC-Grundlagen

Das linke Bild zeigt ein digitalisiertes



Analogsignal und das rechte Bild eine Wandlung des Digital- in ein Analogsignal<sup>18</sup>. Mit den 7 Stufen in dem Bild, also mit 3 Bit ist das Analogsignal nur ein grobes Abbild des ursprünglichen Signals. Übliche DAC haben 10 oder mehr Bit ermöglichen also über 1000 Stufen.

Eine der Anwendungsmöglichkeiten ist die Erzeugung von definierten Wechselsignalen oder die Ausgabe von Musik.

## 10.12.4 DAC-Implementierung im Mikrocontroller

-----#####

Für eine einfache Initialisierung des DAC muss lediglich der entsprechende Pin auf analoge Ausgabe eingestellt werden:

```
LPC_PINCON->PINSEL1 |= (2<<20); // enable AOUT (P0.26) pin
```

Das folgende Programm gibt einen Sinus mit der Amplitude 3V3 und aus:

```
int i=0;
uint32_t periode_us;
periode_us = 10000/360;
LPC_GPIO0->FIODIR |= (1<<24);
for(i=0; i<360; i++)
    sinus[i] = 511 * sin(i * M_PI/180.); //1)

while(1){
    for(i = 0; i < 360; i++){
        LPC_DAC->DACR = (sinus[i]+512) << 6; //2)
        delayUs(periode_us ); //3)
    }
}
```

zu 1)

Hier werden 360 Werte für eine Sinusschwingung (für eine Periode) berechnet. Die Werte bewegen sich zwischen -511 und +511, der 10 Bit Wertebereich des DAC wird damit vollständig ausgenutzt.  $M\_PI$  und die Sinusfunktion stammen aus der C Standardbibliothek für mathematische Funktionen.

zu 2)

<sup>18</sup> Wikipedia

## Einführung in C für STM32

Der DAC kann nur positive Spannungen ausgeben, deshalb müssen alle Werte um 512 nach oben geschoben werden. Im *DAC Controlregister (DACR)* beginnen die DAC Werte beim Bit 6, deshalb müssen alle Werte um 6 Bit nach links verschoben werden.

zu 3)

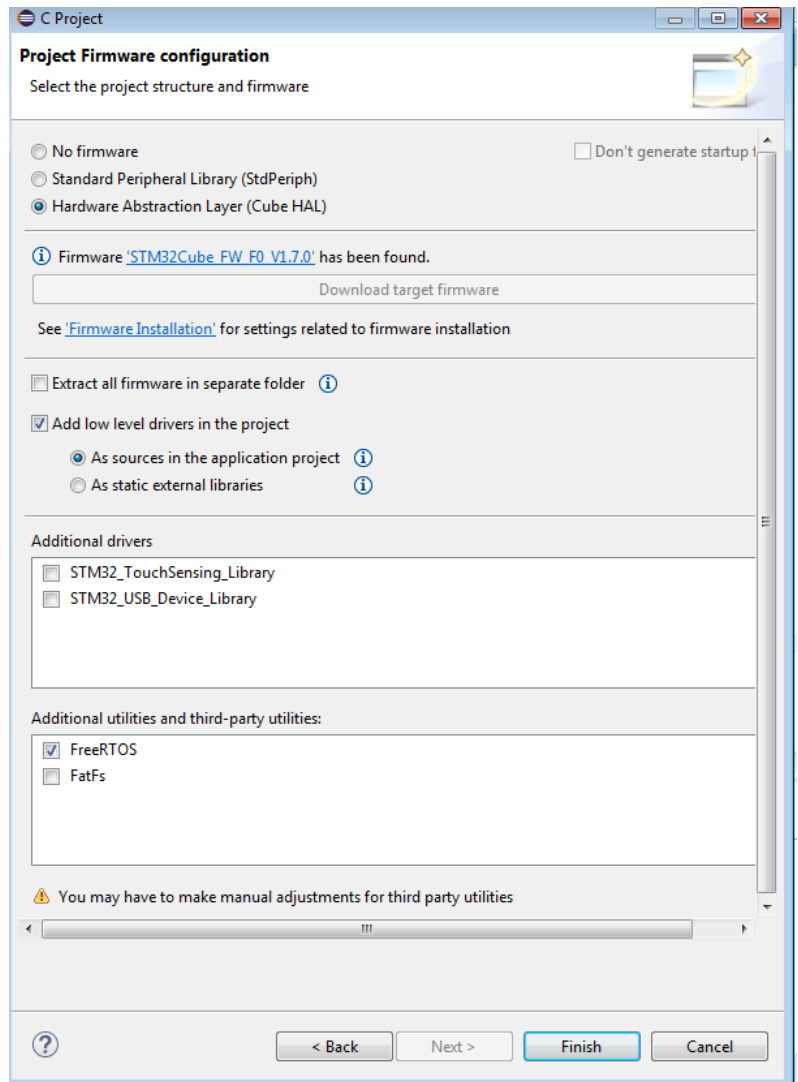
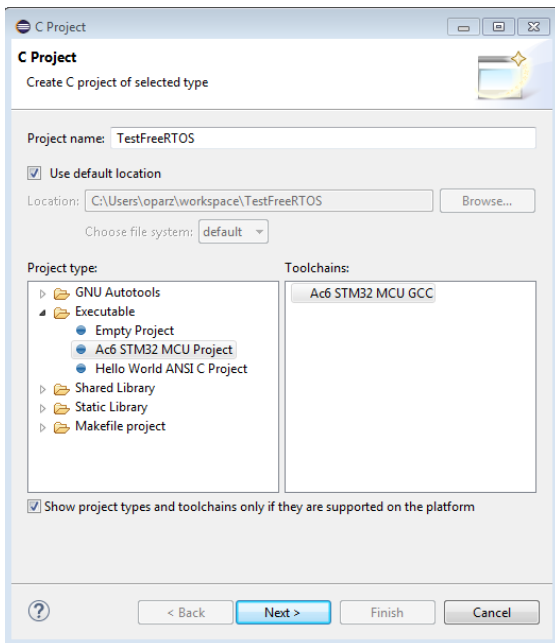
Jetzt muss nur noch die Frequenz eingestellt werden. In dem Beispiel wird eine Periode von 10ms, also eine Frequenz von 100 Hz eingestellt.

## 10.13 RTOS am Beispiel FreeRTOS

Exemplarisch wird hier ein einfaches Blinklicht mit einem STM32 und der OpenSTM-Umgebung gezeigt:

Erstellen Sie zunächst ein neues C-Projekt wie in dem Bild unten links gezeigt. Die beiden nächsten Dialogfenster sind nicht dargestellt, wichtig ist die Auswahl des passenden Prozessors bzw. Boards, In unserem Fall ist das das STM32 Nucleo F072RB Board.

Wichtig ist aber das letzte Dialogfenster unten rechts gezeigt. Hier werden die benötigten Bibliotheken eingebunden unter anderem auch FreeRTOS.





## Einführung in C für STM32

Ein einfaches Programmgerüst für zwei Tasks und toggeln von GPIOs sieht dann folgendermaßen aus:

```
int main(void)
{
    HAL_Init();
    SystemClock_Config();           // Takteinstellung wie bereits bekannt

    GPIO_Init();                   // 1)

    xTaskCreate(vTask1, "Task 1", 128, NULL, 5, NULL ); // 2) höhere Prio
    xTaskCreate(vTask2, "Task 2", 128, NULL, 4, NULL ); // 2) geringere Prio
    vTaskStartScheduler();
    return 0;
}
```

1)

Die folgende Funktion stellt den Portpin PB6 auf Ausgang:

```
void GPIO_Init(void)
{
    GPIO_InitTypeDef G;

    __HAL_RCC_GPIOB_CLK_ENABLE();

    G.Pin = GPIO_PIN_6;
    G.Mode = GPIO_MODE_OUTPUT_PP;
    G.Pull = GPIO_NOPULL;
    G.Speed = GPIO_SPEED_FREQ_HIGH;
    HAL_GPIO_Init(GPIOB, &G);
}
```

2)

Die Aufrufsyntax der Tasks ist recht einfach zu verstehen:

**vTask1:** Name des Tasks

**„Task1“:** Label des Tasks bzw. Bezeichnung

**128:** Größe des Stacks in Byte

**5:** Task hat die Priorität 5, je höher die Zahl umso höher die Prio

**NULL:** nicht verwendete Parameter

Die genaue Beschreibung ist im Referenzmanual von FreeRTOS zu finden.

Die Tasks müssen vor Ihrer Verwendung zunächst definiert werden. In dem einfachen Beispiel schaltet ein Task eine LED an und der andere wieder aus:

```
void vTask1 (void * pd)
{
    vTaskDelay(5);
    while(1){
        HAL_GPIO_WritePin(GPIOB, GPIO_PIN_6, GPIO_PIN_SET);
        vTaskDelay(10);
    }
}
```

```
void vTask2 (void * pd)
{
    while(1){
        HAL_GPIO_WritePin(GPIOB, GPIO_PIN_6, GPIO_PIN_RESET);
        vTaskDelay(20);
    }
}
```

Zeitdiagramm:

Der Scheduler wird in dem Beispiel alle  $T=1\text{ms}$  aufgerufen, d.h. es kann alle 1 ms ein Taskwechsel



## 10.13.1 Headerdateien für die Konfiguration

FreeRTOS kann über Headerdateien flexibel an die Anforderungen der Applikation angepasst werden.

### 10.13.1.1 FreeRTOSConfig.h

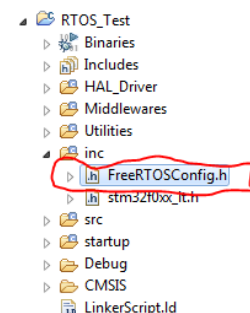
Der folgende Ausschnitt aus diesem Header zeigt einige Einstellungen für das Nucleoboard im Praktikum:

```
#define configUSE_PREEMPTION 1
#define configUSE_IDLE_HOOK 1 // 1)
#define configUSE_TICK_HOOK 1 // 2)
#define configCHECK_FOR_STACK_OVERFLOW 2 // 3)
#define configUSE_MALLOC_FAILED_HOOK 1 // 4)

#define configCPU_CLOCK_HZ ( SystemCoreClock ) // 5)
#define configTICK_RATE_HZ ((TickType_t)1000) // 6)
#define configMAX_PRIORITIES ( 7 )
#define configMINIMAL_STACK_SIZE ((uint16_t)128)
#define configTOTAL_HEAP_SIZE ((size_t)3072)
#define configMAX_TASK_NAME_LEN ( 16 )
#define configUSE_TRACE_FACILITY 1
#define configUSE_16_BIT_TICKS 0
#define configUSE_MUTEXES 1
#define configQUEUE_REGISTRY_SIZE 8

/* Co-routine definitions. */
#define configUSE_CO_ROUTINES 0
#define configMAX_CO_ROUTINE_PRIORITIES ( 2 )

/* Set the following definitions to 1 to include the API function, or zero
to exclude the API function. */
#define INCLUDE_vTaskPrioritySet 1
#define INCLUDE_uxTaskPriorityGet 1
#define INCLUDE_vTaskDelete 1
#define INCLUDE_vTaskCleanUpResources 0
#define INCLUDE_vTaskSuspend 1
```



```
#define INCLUDE_vTaskDelayUntil    0
#define INCLUDE_vTaskDelay        1
```

5)

Im Beispiel ist der Systemtakt des Betriebssystems der Takt des Prozessors

6)

Der Scheduler wird alle Millisekunden aufgerufen, also mit 1000 Hz.

### 10.13.1.2 Hook Funktionen

1) bis 4)

„Hook“ Funktionen werden an strategisch wichtigen Stellen des RTOS aufgerufen um z.B. Fehler oder unerwartete Reaktionen der Applikation lokalisieren zu können. Wenn Sie anstelle der 0 die 1 in diese Zeilen eintragen müssen Sie in Ihrem Programm die folgenden Funktionen definieren, diese werden dann vom Betriebssystem automatisch aufgerufen bei jedem Idle-Task bzw. bei jedem TimeTick:

```
void vApplicationIdleHook( void )
{
    Idle_blinken_oder_sonstwas();
}

void vApplicationTickHook( void )
{
    Tick_blinken_oder_sonstwas();
}
```

3)

Der StackOverflowHook ist sehr wertvoll um einen Überlauf des Stacks zu erkennen. Die Funktion muss exakt so benannt werden wie in dem folgenden Beispiel:

```
void vApplicationStackOverflowHook(TaskHandle_t pxTask, char *pcTaskName) {
    CLI Write("StackoverflowHook: Stack Overflow durch den Task: %s\r\n",
              pcTaskName);
    while (1);
}
```

In dem Beispiel wird der Taskname, der den Überlauf verursacht auf der seriellen Konsole (mit Teraterm ausgegeben) und dann das Programm in einer Endlosschleife beendet.

4)

Überläufe des Heapspeichers können mit der folgenden Funktion erkannt werden:

```
void vApplicationMallocFailedHook(void) {
    CLI Write("MallocHook: Malloc failure\r\n");
    while (1);
}
```

### 10.13.1.3 RunTime Statistik

Oftmals ist es sehr interessant zu wissen wie hoch die CPU Auslastung der Applikation ist. Dafür bietet FreeRTOS einen “relativ” einfachen Mechanismus an. Das Vorgehen zeigen die folgenden Codeschnipsel:

- 1) In der Konfigurationsdatei *FreeRTOSConfig.h* müssen folgende Zeilen angepasst bzw. eingefügt werden:

```
//Test runtime statistics
extern void vConfigureTimerForRunTimeStats( void );
extern uint32_t vGetTimerForRunTimeStats( void );
#define portCONFIGURE_TIMER_FOR_RUN_TIME_STATS() vConfigureTimerForRunTimeStats()
#define portGET_RUN_TIME_COUNTER_VALUE() vGetTimerForRunTimeStats( )
```

- 2)

Die folgenden Funktionen müssen Sie in Ihrer Applikation definieren, also z.B, in main.c.

Die Funktion `vConfigureTimerForRunTimeStats()` müssen Sie mit Leben erfüllen. Die RuntimeStatistik benötigt einen Timer mit Timeticks wesentlich größer als der Heartbeat. Im folgenden Beispiel wird der Timer 2 verwendet mit einem Timetick von 1Mikrosekunde und einer Laufzeit bis Überlauf von 1000s:

```
/* Timer 2 wird verwendet für die Erzeugung des Ticks für die Statistic. */
void vConfigureTimerForRunTimeStats(void) {
    TIM_HandleTypeDef htim2;
    TIM_ClockConfigTypeDef sClkConfig;

    __HAL_RCC_TIM2_CLK_ENABLE();
    htim2.Instance = TIM2;
    htim2.Init.Prescaler = 47; //1) ergibt f = 1000kHz Timetick = 1us
    htim2.Init.CounterMode = TIM_COUNTERMODE_UP;
    htim2.Init.Period = 1E9; //2) 1e9*1us = 1000s

    htim2.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1; // default
    htim2.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_ENABLE; // default
    HAL_TIM_Base_Init(&htim2);

    sClkConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL; // default
    HAL_TIM_ConfigClockSource(&htim2, &sClkConfig); // default
    HAL_TIM_Base_Start(&htim2);
}
```

Die Funktion, die den Timetick für FreeRTOS liefert ist sehr kurz:

```
uint32_t vGetTimerForRunTimeStats(void) {

    return TIM2->CNT;
}
```

Nach diesen Vorbereitungen können Sie eine Funktion von FreeRTOS aufrufen, die Ihnen eine Statistik über den Ressourcenverbrauch der Applikation liefert . Einen Auszug aus der Konsole zeigt das folgende Bild:

```
TSende: Queue gesendet
vTEmpfange: Queue empfangen = 7
TSende: Queue gesendet
vTEmpfange: Queue empfangen = 8
TSende: Queue gesendet
vTEmpfange: Queue empfangen = 9
Task Statistik 12449 <1
IDLE 4950992 98
Task Blink 963 <1
Task Empfange 30455 <1
Task Sende 22006 <1
TSende: Queue gesendet
vTEmpfange: Queue empfangen = 10
```

Der IDLE Task verbraucht 98% Ressourcen und alle weiteren Task weniger als 1%, die CPU ist also praktisch im Leerlauf.

Die Kernelfunktion in dem Task für die Statistik lautet `vTaskGetRunTimeStats` und eine Definition des Statistik Tasks könnte so aussehen:

```
void vTStatistik(void * pd){
    const uint8_t anzTasks = 5;
    char str[anzTasks * 40]; //lt. FreeRTOS Doku max ca. 40 Bytes pro Task
    while(1){
        vTaskGetRunTimeStats(str);
        CLI_Write(str);
        vTaskDelay(5000); //wird alle 5 Sekunden aufgerufen
    }
}
```

Sinnvollerweise gibt man diesem Task die geringste Priorität im System und lässt ihn nur sehr selten laufen, da er sonst selber die Statistik beeinflusst (die Ausgabe der Statistik über die serielle Schnittstelle benötigt ja auch nicht unerhebliche Rechenzeit der CPU).

### 10.13.1.4 Queue

Mit einer Queue können Sie Variablen bzw. beliebige Daten zwischen Tasks austauschen.

1) Diese Includedatei müssen Sie hinzufügen:

```
#include "queue.h"
```

2) Diese globale Variable enthält die Queue Handle:

```
QueueHandle_t xQueue; //
```

Sie wird in mehreren Tasks verwendet und muss deshalb unbedingt global deklariert werden

3) Vor dem Verwenden einer Queue muss diese erzeugt werden. Der folgende Code zeigt einen Task in dem zunächst der Task erzeugt wird und anschließend Daten in die Queue geschrieben werden. Das Beispiel zeigt eine Queue für eine 16Bit Variable und Platz für 10 Einträge:

```
void vTSende(void * pd) {
    int16_t var = 0;
```

```
xQueue = xQueueCreate(10, sizeof(int16_t)); //
if (xQueue == NULL)
    CLI_Write("TSende: Queue konnte nicht erzeugt werden\r\n");

while (1) {
    if (xQueueSendToBack(xQueue, &var, 10) != pdPASS) { //
        CLI_Write("TSende: Senden hat nicht geklappt\r\n");
    } else
        CLI_Write("TSende: Queue gesendet\r\n");
    vTaskDelay(100);
}
```

Die Sendefunktion benötigt als

- ersten Parameter den Namen der Queue (xQueue in dem Beispiel),
- als zweiter Parameter muss die Adresse der Variablen übergeben werden (var ist die Variable, also muss &var übergeben werden) und
- als dritter Parameter folgt der Timeout ( 10 bedeutet 10 Heartbeats).

Da die Sendefunktion nicht blockierend ist, muss in jedem Fall der Task wenigstens für einen Heartbeat geblockt werden damit andere Tasks laufen können.

Ein einfacher Task für das Lesen aus der Queue ist hier gezeigt:

```
void vTEmpfange(void * pd) {
    int16_t var;
    char str[80];
    while (1) {
        if (xQueueReceive(xQueue, &var, 1000)) { //
            sprintf(str, "vTEmpfange: Queue empfangen = %d\r\n", var);
            CLI_Write(str);
        } else
            CLI_Write("vTEmpfange: Queue leer\r\n");
        vTaskDelay(10);
    }
}
```

Die Empfangsfunktion benötigt als

- ersten Parameter den Namen der Queue (xQueue in dem Beispiel),
- als zweiter Parameter muss die Adresse der Variablen übergeben werden (var ist die Variable, also muss &var übergeben werden) und
- als dritter Parameter folgt der Timeout (1000 bedeutet 1000 Heartbeats).

Die blockierende vTaskDelay Funktion benötigt nicht unbedingt ein TaskDelay, allerdings hängt dies davon ab wie oft die Queue befüllt wird. Wenn der SendeTask z.B. im Abstand von 50Millisekunden in die Queue schreibt haben andere Tasks genügend Zeit, anders sieht es aus wenn die Zykluszeit des Sendetasks immer weiter erhöht wird.

Eine typische main ist nachfolgend gezeigt:

```
int main(void) {
    HAL_Init();
    SystemClock_Config();

    GPIO_Init();
    vConfigureTimerForRunTimeStats();           //1)
    CLI_Configure(115200);                       //2)
    CLI_Write("Start\r\n");                     //2)

    xTaskCreate(vTStatistik, "Task Statistik", 256, NULL, 4, NULL); //3)
    xTaskCreate(vTEmpfange, "Task Empfange", 256, NULL, 6, NULL); // mittlere Prio
    xTaskCreate(vTSende, "Task Sende", 256, NULL, 7, NULL);       // höhere Prio

    vTaskStartScheduler();
    return 0;
}
```

- 1) Diese Funktion ist weiter oben beschrieben und wird für den Statistik Task benötigt.
- 2) Dies sind lediglich Wrapperfunktionen für die UART Schnittstelle. Dahinter verbirgt sich die Initialisierung der UART und das Senden mit der bekannten Funktion `HAL_UART_Transmit(...)`
- 3) Um den Statistik Task verwenden zu können muss er natürlich auch erzeugt werden.

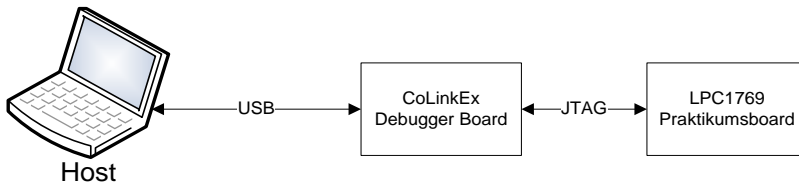
## 10.14 Debuggingverfahren

Das Debuggen, also die Fehlersuche in einem Embedded System unterscheidet sich von dem Debuggen an einer PC Umgebung. Das einfachste Debuggen in einer PC Umgebung ist die Ausgabe von Meldungen mit `printf(...)` auf eine Konsole.

- In einer Mikrocontrollerumgebung ist das einfachste Debuggen ein Blinklicht, dazu muss eine LED auf jeder neu entworfenen Platine unbedingt vorgesehen werden. Mit diesem Blinklicht kann der Download eines Programms und die prinzipielle Funktionsweise der Programmierung getestet werden.
- Weitergehende Tests können mit einem angeschlossenen LCD durchgeführt werden, hier kann dann an strategischen Stellen der Wert von Variablen ausgegeben werden etc.
- Wenn die serielle Schnittstelle bereits in Betrieb genommen werden kann bietet sich die Ausgabe über die UART an den PC an.
- Für Tests an der Hardware sind Oszilloskop und Logikanalysator unabdingbare Hilfsmittel. Mit dem Oszilloskop kann der Spannungsverlauf von Signalen auf der Platine untersucht werden, in der Regel aber nur 2 Kanäle. Falls nur die digitalen Signale interessieren (High oder Low) ist der Logikanalysator besser geeignet, da hier mindestens 8 Kanäle gleichzeitig betrachtet werden können.

### 10.14.1 Crossdebugger

Mit sogenannten Crossdebuggern kann auf die einzelnen Register und Variablen auf dem Mikrocontrollerboard vom PC (Host) aus zugegriffen werden. Üblicherweise dient als Verbindung zwischen Host und Target die sogenannte JTAG<sup>19</sup> Schnittstelle.



Diese Schnittstelle wurde ursprünglich für den Test von hochintegrierten ICs entwickelt (*Boundary Scan*), im Lauf der Zeit wurde die Schnittstelle in viele Mikroprozessoren und –Controller integriert, hauptsächlich für die Verbindung mit einem Debugger.

### 10.14.2 Simulator

Hier läuft die Software nur auf dem Host. Damit können recht leicht logische Fehler des Programms entdeckt werden, aber es ist in der Regel nicht möglich auch die Peripherie zu simulieren. Einige kommerzielle Entwicklungsumgebungen bieten diese Möglichkeit an.

## 10.15 Profiling

Unter diesem Begriff wird die dynamische Untersuchung der Laufzeit und Abdeckung einzelner Programmteile verstanden. Unter Linux gibt es dafür leistungsfähige OpenSource Programme wie *gprof*<sup>20</sup>.

Für einfache Tests um die Laufzeit von Code zu messen ist es am einfachsten mit GPIO-Ports und Logikanalysator oder einem Timer und Debugger zu arbeiten.

---

<sup>19</sup> Joint Test Action Group

<sup>20</sup> GNU gprof



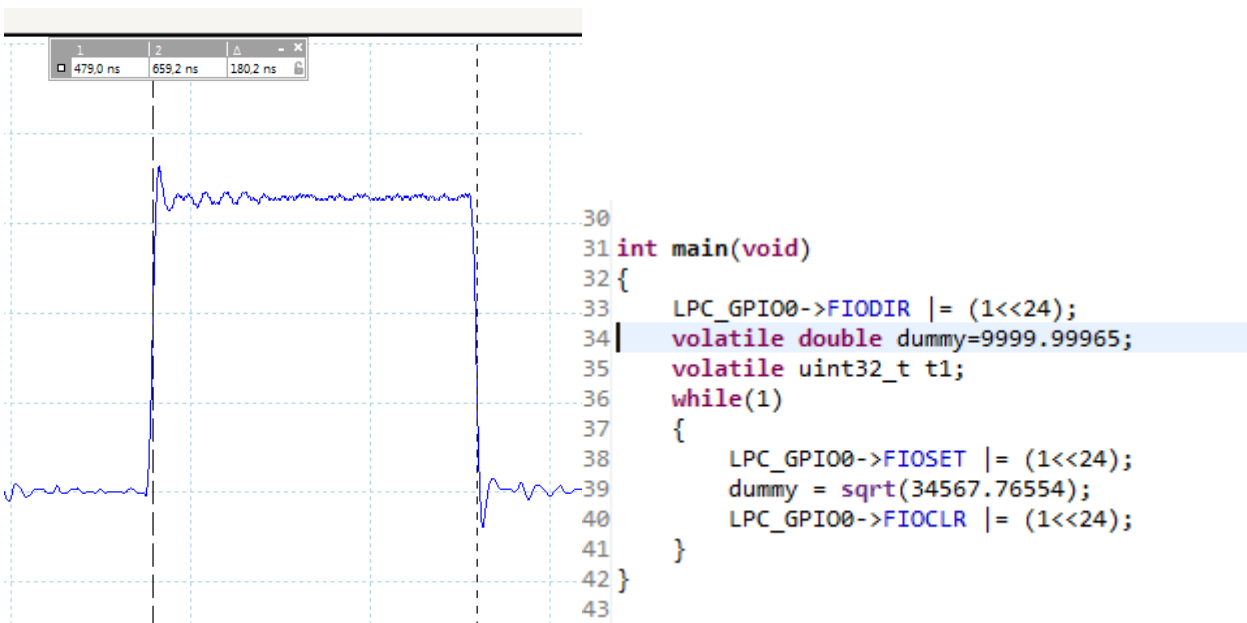
## Einführung in C für STM32

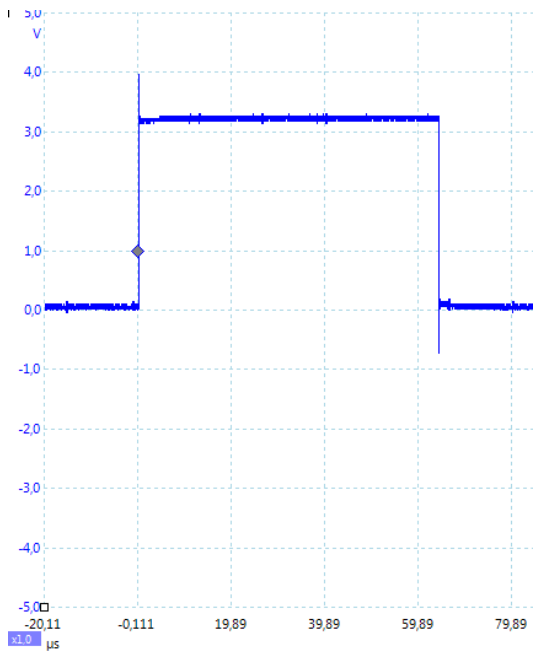
Das Beispiel zeigt die Laufzeitmessung einer Wurzelberechnung. Natürlich ist die Messung mit Einbeziehung des Debuggers nicht 100% genau, aber eine ziemlich genaue Schätzung der Rechenzeit ist möglich. In dem gezeigten Beispiel taktet der Timer mit 25 MHz, das bedeutet einen Timetick von 40 ns, damit dauert eine Wurzelberechnung 10 Ticks, also 400ns.

```
12 int main(void)
13 {
14
15     LPC_SC->PCONP |= (1<<1);
16     LPC_TIM0->PR = 0;
17     LPC_TIM0->MR0 = 25000000;
18     LPC_TIM0->MCR |= (1<<0) | (1<<1); //Interrupt | Reset on Match
19     LPC_TIM0->TCR = 0x1; //start timer
20
21     volatile double dummy=9999.99965;
22     volatile uint32_t t1;
23     while(1)
24     {
25         dummy = sqrt(34567.76554);
26         t1 = LPC_TIM0->TC;
27     }
28 }
29
```

Name	Value
dummy	185.92408542198078
t1	10

Eine wesentlich genauere Messung ohne Verwendung eines Debuggers liefert die Messung an einem GPIO Port. In dem Beispiel wird der Port P0.24 vor der Berechnung gesetzt und nachher gelöscht. Das Bild am Oszilloskop zeigt eine Zeit von ca. 180 ns an.





Ein weiteres Beispiel zeigt die Rechendauer für ein FIR Filter. Das Filter wurde mit dem Freewareprogramm [winfilter<sup>21</sup>](#) erstellt und arbeitet mit 31 Taps und Floating-Point Koeffizienten. Die Zeit für die Berechnung ist ca. 65 µs.

//Tiefpassfilter

```
LPC_GPI00->FIODIR|= (1<<16);  
LPC_GPI00->FIOPIN|= (1<<16);  
gefiltert = fir(ungefiltert);  
LPC_GPI00->FIOPIN&=~ (1<<16);
```

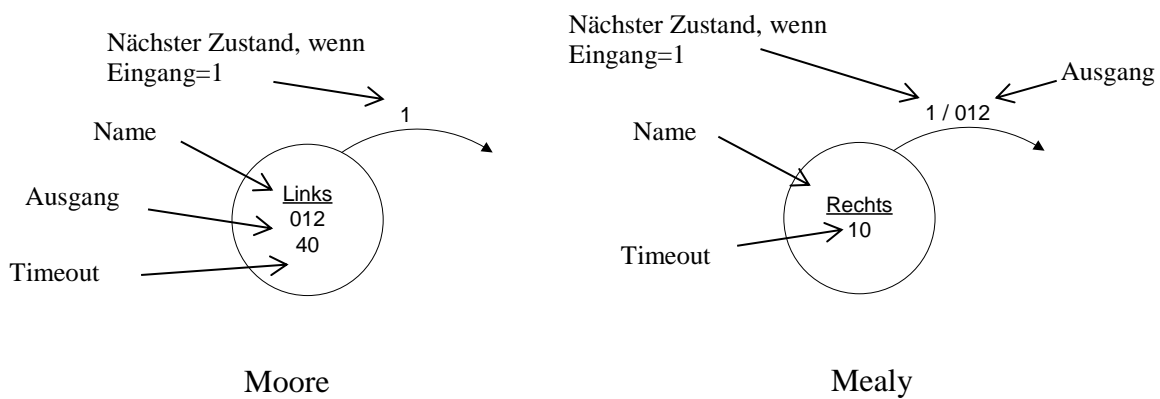
## 11 State Machine

Zustandsautomaten mit einer endlichen Zahl von Zuständen (Finite State Machines = FSM) bieten eine wertvolle Abstraktionsmöglichkeit für die Entwicklung von (Embedded-)Software. Die abstrakten Prinzipien sind dabei Eingänge, Ausgänge, Zustände und Zustandsübergänge. Die FSM Zustandsbeschreibung enthält die zeitabhängige Beziehung zwischen den Ein- und Ausgängen. Es gibt zwei verbreitete Realisierungen von FSM, den *Mealy*- und den *Moore Automat*.

Bei einer Moore FSM hängt der Ausgang nur vom Zustand ab, bei einer Mealy FSM hängt dagegen der Ausgang sowohl vom Eingang als auch vom Zustand ab.

### Zustandsdiagramm

Eine anschauliche Beschreibung des Problems liefert das Zustandsdiagramm (Zustandsgraph).



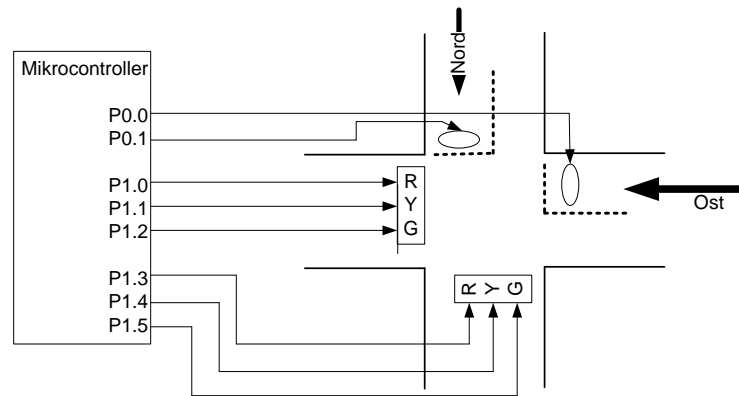
- Die Zustände werden als Kreise oder Ellipsen gezeichnet. Für die Namen der Zustände sollen einprägsame Namen gewählt werden (z.B. Geradeaus, Stop, etc...).
- Der Eingang steht über der Übergangslinie von einem zum nächsten Zustand.
- Der Ausgang steht beim Moore Automat im Kreis unter dem Namen und beim Mealy Automat oberhalb der Übergangslinie, üblicherweise unter dem Eingang.
- Die Übergänge von einem zum nächsten Zustand werden mit einer Linie und einem Pfeil gekennzeichnet.
- Eventuelle Timeouts oder Delays stehen im Kreis unterhalb des Ausgangs bzw. unterhalb des Namens beim Mealy Automat. Üblich ist aber auch ein eigener Zustand für Zeitverzögerungen. Selbstverständlich darf die Statemachine nicht in einer Warteschleife geblockt werden, deshalb werden Zeitverzögerungen üblicherweise mit Timerinterrupts gelöst.

Der prinzipielle Ablauf für einen Moore Automaten ist immer derselbe:

1. Eingänge einlesen
2. Wechseln vom aktuellen Zustand in den Folgezustand
3. Ausgabe der zum neuen Zustand passenden Signale

### 11.1.1 Beispiel Ampelsteuerung

Als erstes Beispiel soll eine Ampelsteuerung erstellt werden:



## Eingänge:

In die Fahrbahn sind sowohl in Nordsüd-, als auch in Ostwestrichtung Induktionsschleifen eingelassen

P0.0: Schleife Nordsüd

P0.1: Schleife Ostwest

Es empfiehlt sich für die Eingänge kurze Bezeichnungen zu verwenden, z.B. P0.0 = SN und P0.1=SO für Schleife Nord etc. Diese Namen können im weiteren als binäre Variable verwendet werden

SN	SO	
0	0	keine Autos auf den Schleifen
0	1	Autos auf der OstWestrichtung
1	0	Autos auf der NordSüdrichtung
1	1	Autos in beiden Richtungen

Wenn auf beiden Richtungen keine Autos sind, kann dies mit der Funktion

$$\overline{SN} \cdot \overline{SO}$$

beschrieben werden.

Zu Beginn soll die Ostwestrichtung Vorrang haben, also Grün für die Ostwestrichtung.

Wenn kein Auto fährt soll die Ampelschaltung im gleichen Zustand bleiben, also Grün in einer der beiden Richtungen.

Die Gelbphase soll 5 s dauern

Die Grünphase soll mindestens 30 s dauern

Wenn Autos in beiden Richtungen kommen sollen die einzelnen Phasen durchlaufen werden

## Zustände

Mögliche Zustände:

`OSTWEST_GRUEN`,

`OSTWEST_GELB`,

ALLE\_ROT\_2,  
 NORDSUED\_ROTGELB,  
 NORDSUED\_GRUEN,  
 NORDSUED\_GELB,  
 ALLE\_ROT\_1,  
 OSTWEST\_ROTGELB

## Ausgänge

Die Ausgänge können wir als Zahl in Binärdarstellung schreiben und die Ports zuordnen:

Rot-Ost RE	Gelb-Ost YE	Grün-Ost GE	Rot-Nord RN	Gelb-Nord YN	Grün-Nord GN
P0.0	P0.1	P0.2	P0.3	P0.4	P0.5

Wenn wir festlegen:

- Ampel leuchtet = „1“ und
- Ampel leuchtet nicht = „0“

muss für den Zustand *Ostwest-Grün* die Dualzahl 001100 ausgegeben werden.

## Ausgangstabelle

Zustand	Ausgang					
	RE	YE	GE	RN	YN	GN
OSTWEST_GRUEN	0	0	1	1	0	0
OSTWEST_GELB	0	1	0	1	0	0
ALLE_ROT_2	1	0	0	1	0	0
NORDSUED_ROTGELB	1	0	0	1	1	0
NORDSUED_GRUEN	1	0	0	0	0	1
NORDSUED_GELB	1	0	0	0	1	0
ALLE_ROT_1	1	0	0	1	0	0
OSTWEST_ROTGELB	1	1	0	1	0	0

## Zustandsübergangstabelle

Da diese Tabelle schon passend geordnet ist, kann sie problemlos in eine Zustandsübergangstabelle erweitert werden:

Zustand	Ausgang	Folgezustand
OSTWEST_GRUEN	001 100	OSTWEST_GELB
OSTWEST_GELB	010 100	ALLE_ROT_2
ALLE_ROT_2	100 100	NORDSUED_ROTGELB
NORDSUED_ROTGELB	100 110	NORDSUED_GRUEN
NORDSUED_GRUEN	100 001	NORDSUED_GELB
NORDSUED_GELB	100010	ALLE_ROT_1
ALLE_ROT_1	100100	OSTWEST_ROTGELB
OSTWEST_ROTGELB	110100	OSTWEST_GRUEN

## Zustandsübergangsdiagramm

###todo

## 11.1.2 Realisierung in C

Für die Realisierung in C bieten sich mehrere Möglichkeiten an:

- Switch Case
- Tabelle
- Funktionszeiger

Für die Bezeichner der Zustände bietet sich der Datentyp *enum* an:

```
typedef enum { OSTWEST_GRUEN, OSTWEST_GELB, ALLE_ROT_2, WARTE_NORDSUED,  
               NORDSUED_ROT_GELB, NORDSUED_GRUEN, NORDSUED_GELB, ALLE_ROT_1,  
               OSTWEST_ROT_GELB, } state_t;  
  
state_t a_state = OSTWEST_GRUEN, f_state; //1)
```

zu 1)

Mit *typedef* wird die Verwendung von *enum* vereinfacht, Sie müssen bei der Deklaration der Variablen nur den selbst gewählten Bezeichner verwenden, in diesem Fall erzeugen Sie zwei Variablen mit dem Datentyp *state\_t*.

Die Variable *a\_state* soll den aktuellen Zustand beschreiben und die Variable *f\_state* den Folgezustand.

```
f_state = a_state;  
switch( a_state ) {  
    case OSTWEST_GRUEN:  
        if(LPC_GPIO0->FIOPIN&0x03 == 0x02 || LPC_GPIO0->FIOPIN&0x03 == 0x03)//2)  
            f_state = OSTWEST_GELB;  
        else  
            f_state = OSTWEST_GRUEN;                //nicht notwendig  
            break;  
    case OSTWEST_GELB:  
        . . .  
        . . .  
    default:  
        break;  
}  
}  
  
//outputs  
if (f_state != a_state){                                //3)  
    switch( a_state ) {  
        case OSTWEST_GRUEN:  
            LPC_GPIO0->FIOSET |= 0x0C;                //001 100  
            break;  
        case OSTWEST_GELB:  
            LPC_GPIO0->FIOSET |= 0x14;                //010 100  
            break;  
        . . .  
        . . .  
        default:  
            break;  
    }  
    a_state = f_state;  
}
```

zu 2)

In dieser Zeile werden die Eingänge abgefragt.

zu 3)

Mit dieser Abfrage wird die *switch case* Anweisung für die Ausgabe nur aufgerufen, wenn sich eine Änderung ergeben hat. Damit werden weniger Rechnerressourcen verbraucht, es dient also der Optimierung des Programms.

### *Unterschiedliche Timeouts für die Zustandsübergänge*

Bei der Ampel sollen die Perioden der einzelnen Zustände natürlich unterschiedlich lang sein, die Grünphase wesentlich länger als der Übergang von Rot zu Grün.

Das einfachste wäre ohne näheres nachdenken eine Wartschleife im Grünzustand! Diese Realisierung ist aber sehr schlecht, da dann die Statemachine nicht mehr regelmäßig (synchron) mit einem festen Takt abläuft. Eine elegante Lösung bieten die Timer in Verbindung mit einem Interrupt an. In dem Beispiel wird der SysTick Timer verwendet:

```
void SysTickInit(void)
{
    SysTick->CTRL &= ~(1<<0);           // enable bit löschen
    SysTick->LOAD = SysTick->CALIB;      // 1)
    SysTick->CTRL = ((1<<2) | (1<<1)|(1<<0)); //enable+cpu_clock source+int enable
}

/* SysTick Interrupt Handler (10ms) */
void SysTick_Handler (void)
{
    static uint16_t cnt_period = 100;    // 2)
    LPC_GPIO0->FIOPIN ^= (1<<24);        // 3)
    if (--cnt_period==0) {                // 4)
        cnt--;                            // 5)
        cnt_period =100;
        //LPC_GPIO0->FIOPIN ^= (1<<24);
    }
}
```

Zu 1)

Der SysTick Timer stellt ein besonderes Register zur Verfügung (CALIB), wenn das Reloadregister mit dessen Inhalt geladen wird, ergibt es einen Überlauf in 10ms Abstand.

Zu 2)

Variablen mit dem **static** Bezeichner werden schon bei dem Compiliervorgang an eine bestimmte Adresse im RAM abgelegt, die Initialisierung erfolgt einmalig, in dem Beispiel mit dem Wert 100.

Zu 3)

Das toggeln des Pins dient nur zum Test der Periode des Interrupts

Zu 4)

Die Variable cnt dient zum einstellen der unterschiedlichen Zeiten in denen jeder Zustand verbringt. Diese Variable muss global deklariert sein und wird in jeder Sekunde dekrementiert. In den jeweiligen Zuständen muss nun im aktuellen Zustand die Variable mit dem passenden Wert für

den Folgezustand gesetzt werden. Der Code zeigt den Ausschnitt für den Übergang vom Zustand *OSTWEST\_GELB* in den Zustand *ALLE\_ROT\_1*. Wenn der Zähler *cnt* nicht den Wert 0 erreicht hat, bleibt die Statemachine im alten Zustand, wechselt bei *cnt=0* in den nächsten Zustand und stellt die Zeit für diesen Zustand ein, im Beispiel 1s.

```
case OSTWEST_GELB:
    if (cnt==0){
        cnt=1;
        f_state = ALLE_ROT_1;
    }
```

Da der Ablauf in dieser Statemachine relativ langsam ist und auch die Reaktionszeiten eher im größeren Millisekundenbereich sind, kann das Raster von 1 ms auch für die Wiederholrate der Statemachine verwendet werden. Auf diese Weise hat der Mikrocontroller noch viel Ressourcen für andere Aufgaben frei.

### Realisierung als Tabelle

Die Zustands-übergangstabelle kann nun leicht erweitert werden um die Eingänge und die jeweiligen Timeouts

<i><b>Zustand</b></i>	<i><b>Eingang [SN SO]</b></i>	<i><b>Ausgang</b></i>	<i><b>Timeout</b></i>	<i><b>Folgezustand</b></i>
<i>OSTWEST_GRUEN</i>	<i>10, 11</i>	<i>001 100</i>	<i>30</i>	<i>OSTWEST_GELB</i>
<i>OSTWEST_GELB</i>	<i>X</i>	<i>010 100</i>	<i>5</i>	<i>ALLE_ROT_2</i>
<i>ALLE_ROT_2</i>	<i>X</i>	<i>100 100</i>	<i>1</i>	<i>NORDSUED_ROTGELB</i>
<i>NORDSUED_ROTGELB</i>	<i>X</i>	<i>100 110</i>	<i>5</i>	<i>NORDSUED_GRUEN</i>
<i>NORDSUED_GRUEN</i>	<i>01, 11</i>	<i>100 001</i>	<i>30</i>	<i>NORDSUED_GELB</i>
<i>NORDSUED_GELB</i>	<i>X</i>	<i>100010</i>	<i>5</i>	<i>ALLE_ROT_1</i>
<i>ALLE_ROT_1</i>	<i>X</i>	<i>100100</i>	<i>1</i>	<i>OSTWEST_ROTGELB</i>
<i>OSTWEST_ROTGELB</i>	<i>X</i>	<i>110100</i>	<i>5</i>	<i>OSTWEST_GRUEN</i>

*X = don't care*



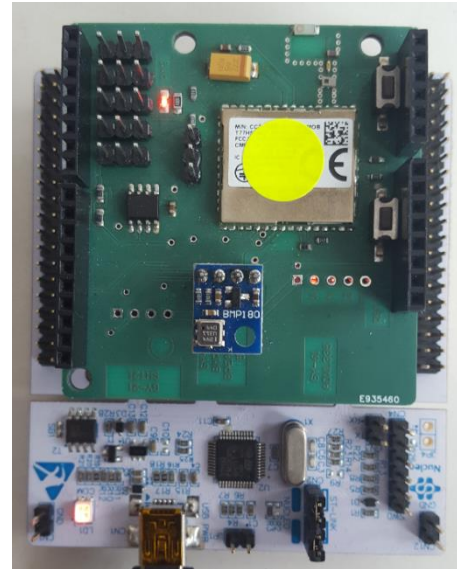
## 12 IoT (Internet of Things)

Das Aufsteckmodul auf das Praktikumsboard liefert eine Anbindung an das *Internet of Things* (IoT). Es geht dabei um Geräte, die mit dem Internet verbunden sind z.B. SmartWatches, Rad-Computer etc.

Stellvertretend wird der Chip CC3100 der Fa. TI betrachtet. Das Bild zeigt das selbstentwickelte Board im Arduino-Shield Footprint.

Von TI werden viele Beispiele bereitgestellt, die alle auf der STM32 Hardware lauffähig sind.

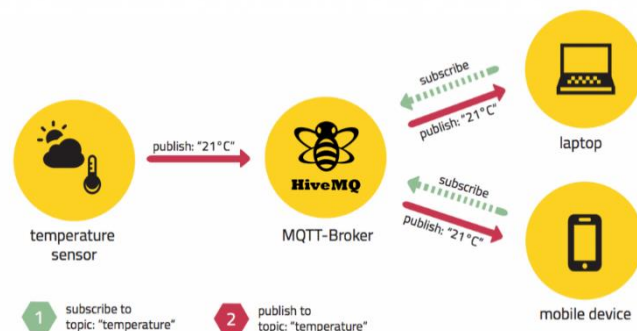
Die Kommunikation zwischen dem CC3100 Modul und dem Praktikumsboard erfolgt über die SPI Schnittstelle.



### 12.1 MQTT Protokoll

Hier wird exemplarisch die Verwendung als mqtt Client vorgestellt.

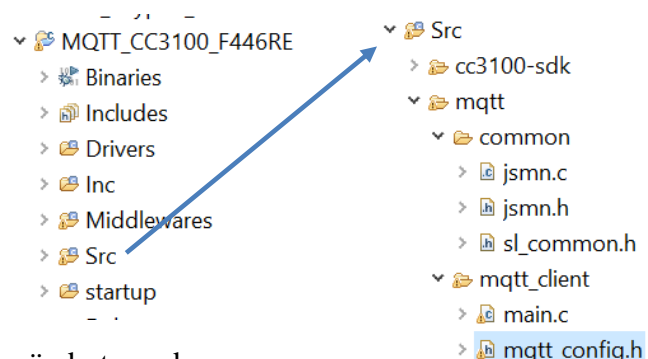
Den prinzipiellen Aufbau einer mqtt Kommunikation zeigt das folgende Bild:



#### 12.1.1 CC3100-MQTT Projekt

Das Dokument bezieht sich auf ein OpenSTM32 Projekt, die Basis ist ein Beispielprojekt aus der TI-CC3100 SDK.

Das Projekt ist umfangreich, für den Anwender sind aber nur einige Dateien und Verzeichnisse interessant. Das Verzeichnis Middlewares enthält den Quellcode von FreeRTOS, in diesem Verzeichnis darf nichts verändert werden.



**sl\_common.h:** In dieser Datei werden die Zugangsdaten zu dem Accesspoint eingestellt:

```
#ifndef D204_NETWORK
#define SSID_NAME "FRITZ!Box Fon WLAN 7170" /* Access point name to connect to. */
```

## Einführung in C für STM32

```
#define SEC_TYPE      SL_SEC_TYPE_WPA_WPA2 /* Security type of the Access point */
#define PASSKEY       "7012605071797767"  /* Password in case of secure AP */
#define PASSKEY_LEN   pal_Strlen(PASSKEY)  /* Password length in case of secure AP */
#endif
```

**mqtt\_config.h:** In dieser Datei wird u.a. der Broker eingetragen:

```
.....
// #define SERVER_ADDRESS      "broker.hivemq.com"
// #define SERVER_ADDRESS      "test.mosquitto.org"
// #define SERVER_ADDRESS      "mqtt.flespi.io"
// #define SERVER_ADDRESS      "broker.mqttdashboard.com"
#define SERVER_ADDRESS        "iot.eclipse.org"          //1)

#define PORT_NUMBER            1883                      //2)

#define CLIENT_ID               "labor_d204"              //3)
.....
```

- 1) Es gibt eine Vielzahl von frei verfügbaren und kostenpflichtigen Brokern, der Domainname muss hier eingetragen werden.
- 2) Die 1883 ist die übliche Portnummer für das mqtt Protokoll.
- 3) Diese ID muss eindeutig sein!



Wenn sie aber sich aber z.B. über einen Web-Client mit dem Broker verbinden und sich zu einem Topic anmelden („subscribe“), dürfen Sie nicht dieselbe Client-ID verwenden wie der Client der den Topic gesendet hat („publish“).

**main.c:** In dieser Datei werden die Sensoren erfasst, in dem Beispiel ist es ein Temperatursensor

```
/**
 * Temperaturmessung mit dem DS1621 Sensor
 */
void DS1621Task(void *pdata) {
    char str[PRINT_BUF_LEN];
    int16_t temp;
    publish_message var;

    initLEDs();
    while (1) {
        g_publishCount++;
        temp = ReadDS1621(); //1)
        var.event = TEMPERATURE_MEASURED; //2)
        strcpy(var.pub_topic, PUB_TOPIC_2); //2)
        sprintf(str, "%d: %d.%d", g_publishCount, temp/10, temp%10); //2)
        strcpy(var.data, str); //2)

        osi_MsgQWrite(&g_PBQueue, (publish_message *)&var, OSI_NO_WAIT); //3)
        CLI_Write("%s: %s\r\n", str, SERVER_ADDRESS);
        toggleLed(LED1);
        osi_Sleep(2000);
    }
}
```

## Einführung in C für STM32

- 1) Diese Funktion liefert die Temperatur als 16 bitbreiten Integerwert, Die Zahl 273 bedeutet 27,3 Grad C.
- 2) Hier werden die Daten in eine Struktur geschrieben

```
var.event =      TEMPERATURE_MEASURED
var.pub_topic =  "/cc3100/Temperature"
var.data =      "153:  27.3"
```

- 3) Diese FreeRTOS Funktion schreibt die Struktur in eine Queue, die Funktion ist eine einfache Wrapperfunktion um die FreeRTOS Funktion `xQueueSendFromISR`.

Der Inhalt der Queue wird dann in einer Endlosschleife im Hauptprogramm gelesen und an den Broker gesendet:

```
static void MqttClient(void *pvParameters)
{
    . . . . .
    /*
     * Endlosschleife für das Senden der Daten an den Broker
     */
    for (;;) {
        osi_MsgQRead(&g_PBQueue, &RecvQue, OSI_WAIT_FOREVER);           //(1)
        . . .
        if (TEMPERATURE_MEASURED == RecvQue.event) {
            publishData((void*) local_con_conf[iCount].clt_ctx,
                        RecvQue.pub_topic, RecvQue.data);
        }
        . . . . .
        else if (BROKER_DISCONNECTION == RecvQue.event) {
            iConnBroker--;
            if (iConnBroker < 1) {
                /*
                 * Device not connected to any broker
                 */
                goto end;
            }
        }
    }
}
```

- 1) Die Lesefunktion aus der Queue ist wiederum eine Wrapperfunktion um eine FreeRTOS Funktion (`xQueueReadFromISR`).

Der Task für das Schreiben der Messwerte in die Queue muss natürlich erst mal erzeugt werden, dies ist ein Ausschnitt aus der `main(void)` Funktion:

```
retVal = osi_TaskCreate(DS1621Task, "DS1621", 1024, NULL, 10, NULL);
if (retVal < 0) {
    LOOP_FOREVER();
}
```

In dem Beispiel wird der Task mit 1024 Byte Stack und der Priorität 10 erzeugt, falls dies nicht funktioniert (`retVal` kleiner als Null) endet das Programm in einer Endlosschleife.

## 13 Anhang

### 13.1 Wieso Murphy die Sprache C liebt<sup>22</sup>

Nachfolgend werden einige typische Probleme aufgelistet, welche durch die Verwendung der Programmiersprache C auftreten können. Diese Hinweise haben nicht den Anspruch, vollständig zu sein, zeigen jedoch einige der Hauptprobleme dieser Programmiersprache auf.

#### 13.1.1 Speicherüberlauf von Arrays

In C können Sie, ohne es zu merken (oder eben erst bei einem Systemabsturz), über die Grenzen eines Arrays hinausschreiben. Beispielsweise ist folgender Code syntaktisch korrekt (d.h. der Compiler wird ihn vermutlich ohne Meldung compilieren):

```
char array[10];
char counter = 0;
...
for (counter = 0; counter <= 10; counter++){
    array[counter] = 0;
}
```

Im obigen Code schreiben Sie für counter = 10 über das Array hinaus. Was passiert nun? Wenn Sie Pech haben, hat der Linker die Variable counter im Speicher gleich hinter dem Array angelegt. Mit der Anweisung array[10] = 0; löschen Sie den counter wieder und die for-Schleife wird zur Endlosschleife. Insofern haben Sie nun Glück, weil Sie das im ersten Testdurchgang herausfinden werden. In anderen Fällen kann das Finden des Fehlers aber wesentlich aufwändiger werden.

Um diesem Problem zu begegnen gibt es zwei Ansätze:

- Während dem Codieren achten Sie besonders auf Arraygrenzen. Sie können auch die Indizes vor dem Zugriff auf das Array zusätzlich im Code prüfen. Durch ausführliche Tests zeigen Sie, dass vermutlich nicht über Arraygrenzen geschrieben wird. Eine gewisse Unsicherheit bleibt jedoch bestehen.
- Spendieren Sie dem Array ein zusätzliches Element und initialisieren Sie den letzten Index mit einem bestimmten Wert (z.B. 0x55). Führen Sie nun die Systemtests durch. Schauen Sie anschließend, ob sich der Wert Ihres letzten Elementes nicht verändert hat.

#### 13.1.2 Stacküberlauf

Dieses Thema ist verwandt mit dem Überschreiben von Arraygrenzen. Wenn Ihnen der Stack überläuft (sei das, weil Sie zu viele verschachtelte Funktionsaufrufe haben, oder weil Sie zu große lokale Variablen angelegt haben) überschreiben Sie RAM-Bereiche, an denen Variablen liegen.

Sollte das Programm aus dem RAM ausgeführt werden, kann Ihnen ein überlaufender Stack natürlich auch Programmcode überschreiben.

Die daraus resultierenden Fehler äußern sich irgendwann, je nachdem was überschrieben wurde und wann dies benötigt wird. Entsprechend schwierig sind diese Fehler dann auch zu finden.

##### 13.1.2.1 Wie man einen Stacküberlauf feststellt:

Initialisieren Sie Ihren Stack beim Systemstart mit einem Muster, beispielsweise wieder 0x55. Nach einem Systemtest können Sie nachprüfen, wie viel Prozent Ihres Stacks vom Programm verwendet wurde (die Grenze liegt dort, wo das Pattern 0x55 mit irgendwelchen Werten überschrieben wurde). Was man dagegen tun kann:

- Vergrößern Sie den Stack. Bei vielen Entwicklungsumgebungen können Sie die Stackgröße in den Linker-Optionen definieren.

---

<sup>22</sup> Aus „Microcontroller in Embedded Systems“, FH Bern

- Reduzieren Sie die Größe lokaler Variablen in Ihren Funktionen.
- Vermeiden Sie rekursive Funktionsaufrufe!

### 13.1.3 Nullpointer

Nullpointer sind in C sehr häufig! Entsprechend viele Möglichkeiten gibt es, einen Nullpointer zu erhalten. Wenn Sie beispielsweise einen Pointer anlegen aber nicht initialisieren, wird er Ihnen auf Null (oder auch irgendwohin ins Nirwana) zeigen. Oder wenn Sie Speicher für den Aufbau einer dynamischen Liste anfordern, aber gerade kein Speicher vorhanden ist, werden Sie als Rückgabewert einen Nullpointer erhalten. Der erste Fall ist ein klassischer Programmierfehler. Der zweite Fall ist jedoch kein Fehler sondern er kann durchaus zu Laufzeiten auftreten.

Was man dagegen tun kann:

Prüfen Sie jeden Pointer auf Null oder ungleich Null, bevor Sie weiter mit ihm arbeiten!

Die Prüfung erfolgt entweder durch

```
if(pointer != NULL)
//oder durch
assert(pointer != NULL).
```

### 13.1.4 Interrupts

Ein Mikrocontroller hat viele Interrupt-Vektoren, welche in der Vektor-Tabelle zusammengefasst werden.

Normalerweise werden Sie in Ihrem Code nur ein Paar dieser Vektoren verwenden (Timer, serielle Schnittstelle, vielleicht noch zwei oder drei IRQ-Ports). Die restlichen Interrupts brauchen Sie nicht. Was geschieht nun, wenn ein Interrupt auftritt, mit dem Sie nicht gerechnet haben (beispielsweise eine Division durch Null)? In diesem Falle springt der Controller an die entsprechende Stelle in der Vektor-Tabelle und liest einen undefinierten Wert aus. Wenn Sie Glück haben steht da Null und der Code wird an der Stelle Null ausgeführt, was in der Regel einem Reset entspricht. Wenn Sie Pech haben steht da irgendetwas und das Programm wird irgendwohin springen, was vermutlich zu einem undefinierten Systemverhalten führen wird.

Was man dagegen tun kann:

Initialisieren Sie immer alle Interrupt-Vektoren. Sollten Sie einen Interrupt nicht verwenden, so tragen Sie an dieser Stelle eine Default-ISR ein. In dieser ISR können Sie entweder eine Fehlermeldung ausgeben, oder für den Debug-Betrieb eine Endlosschleife einbauen. Beim Testen werden Sie auf diese Arte undefinierte Interrupts schnell erkennen.

In der Regel wird dies aber bereits im Star-Up Code der Entwicklungsumgebung erledigt.

### 13.1.5 Dynamisch Speicher allozieren

In Programmen wird oft dynamisch Speicher alloziert, etwa um Listen zu generieren. Das geht so lange gut, wie es freien Speicher hat. Wenn der angeforderte Speicher nicht mehr zurückgegeben wird, tritt früher oder später ein Problem auf, d.h. Sie werden keinen Speicher erhalten. Ein weiteres Problem ist die Fragmentierung des Speichers, die durch dynamisches allozieren entstehen kann.

Was man dagegen tun kann:

- Machen Sie alles statisch wenn es irgendwie geht.
- Achten Sie darauf, dass Sie allen angeforderten Speicher auch wieder zurückgeben. Darum müssen Sie sich in C und C++ selber kümmern.
- Testen Sie das System auch in Bezug auf die Speicher-Ressourcen: Lassen Sie es ein paar Stunden oder Tage laufen und prüfen Sie anschließend, ob der freie Speicher etwa konstant bleibt.

### 13.1.6 Race Conditions

*Race Condition* bedeutet Wettlaufsituation. Wenn *Race Conditions* vorhanden sind bedeutet dies, dass das Resultat des ausgeführten Codes vom zeitlichen Ablauf abhängig ist. *Race Conditions* können immer auftreten wenn Hardware und Software asynchron zusammenarbeiten, so etwa bei Timern oder A/D-Wandlern. *Race Conditions* treten dann auf, wenn wir Daten mehrmals lesen müssen, während sie von der Hardware asynchron zur Software verändert werden.

Beispiel Timer: Folgender Code führt zu einer *Race Condition* :

```
unsigned int timer_hi;

interrupt timer()
{
    ++timer_hi
}

unsigned long read_timer(void)
{
    unsigned int low, high;
    low = read_word(timer_register);
    high = timer_hi;
    return (((unsigned long)high)<<16 + (unsigned long)low);
}
```

Obiger Code ist fehlerhaft, wenn nach dem Einlesen des Timer-Registers ein Timer-Overflow stattfindet, bevor die Variable `timer_hi` gelesen wird.

Das Problem kann durch verschiedene Ansätze gelöst werden:

- Berücksichtigung im Design
- Interrupts disable während dem Zugriff auf die Hardware
- Capture-Register verwenden

Achtung: *Race Conditions* sind oft die Ursache von schwer auffindbaren SW-Fehlern

### 13.1.7 Reentrant Code

Embedded Software verwendet fast immer Interrupts, teilweise werden auch Echtzeit-Betriebssysteme eingesetzt. Dies bedeutet, dass eine Funktion jederzeit von einem Interrupt oder von einem anderen Task unterbrochen werden kann. Sobald nun verschiedene Programmteile ohne weitere Vorkehrungen auf gleiche Ressourcen zugreifen (Hardware-Schnittstellen, globale Variablen), ist das Chaos perfekt. Funktionen müssen deshalb reentrant-fähig programmiert werden, d.h.:

- Auf globale Variablen muss exklusiv zugegriffen werden ("atomic")
- Es dürfen keine non-reentrant Funktionen aufgerufen werden (Vorsicht bei Library-Funktionen)
- Auf Hardware-Schnittstellen muss exklusiv zugegriffen werden

Die einfachste Art, exklusiven Zugriff zu erhalten, ist das disabling der Interrupts während dem Zugriff. Dies bedeutet aber eine Verlängerung der Interrupt-Latenzzeiten! Deshalb muss der Zugriff sehr kurz sein.

Eine wichtige Regel lautet in diesem Zusammenhang:

**"Share the code, not the data"!**

**Also wenn möglich so wenig Daten als möglich global verwenden!**

### 13.1.8 Defensive Programmierung

Defensiv bedeutet für die Programmierung, dass man mit allem rechnet: Parameterwerte von Funktionen die gar nicht sein können, korrupte Protokolle die man empfangen hat, zeitliche Abläufe die nicht erwartet wurden usw.

Wenn Sie defensiv programmieren, was insbesondere bei sicherheitsrelevanten Systemen zu empfehlen ist, prüfen Sie alles, bevor Sie es verwenden. Insbesondere bedeutet es:

- Eingangs von Funktionen werden alle übergebenen Parameter auf ihren Wertebereich überprüft.
- Empfangene Protokolle werden auf Korrektheit geprüft (Checksumme).
- Für Resultate wird eine Plausibilitätskontrolle durchgeführt.

Durch die defensive Programmierung erhöhen Sie die Systemsicherheit. Gratis ist das aber nicht zu haben: Sie erzeugen mehr Code, welcher Rechenzeit, Speicher und nicht zuletzt Entwicklungszeit kostet.

### 13.1.9 `assert()`

`assert()` ist ein Makro aus der C-Standard-Bibliothek. Mit `assert()` können während dem Programmfluss sehr einfach und effizient Bedingungen geprüft werden:

```
assert(expression);
```

Hat die Bedingung den Wert Null oder False, gibt das Makro eine Fehlermeldung auf `stderr` aus. Diese hat typischerweise die Form:

```
Assertion failed: expression, file filename, line nnn
```

Wenn beim Einfügen von `<assert.h>` der Makroname `NDEBUG` definiert ist, wird das `assert`-Makro ignoriert.

Beispiele für die Verwendung von `assert()` sind die Prüfung von Pointern (ungleich Nullpointer), die Wertebereiche von Parametern usw.

### 13.1.10 Optimierter C-Code

C-Compiler versuchen, möglichst optimalen Code für den Mikrocontroller zu generieren, d.h. entweder möglichst schnellen oder möglichst kompakten Code. Das Resultat des Compilers können Sie durch die Art und Weise, wie Sie C-Code schreiben, wesentlich beeinflussen:

1. Verzichten Sie möglichst auf globale Variable, legen Sie Variablen lokal in den Funktionen an. Der Compiler hat so die Möglichkeit, für Variable Register zu verwenden, statt diese im Speicher abzulegen. Der Zugriff wird dadurch wesentlich schneller. Versuchen Sie auch, möglichst wenige lokale Variable zu verwenden, die Zahl der Register ist beschränkt. Alle Variablen, die nicht Registern zugewiesen werden, müssen auf den Stack gelegt werden. Wenn Sie dennoch viele lokale Variable brauchen, verwenden Sie diese möglichst nur in einem kleinen Teil der Funktion. So kann der Compiler einem Register während der Ausführung der Funktion diverse Variable zuweisen.
2. Verwenden Sie keine Adressen von lokalen Variablen, der Compiler kann diese sonst nicht in ein Register ablegen.
3. Verwenden Sie Inline-Assembler nur wenn unbedingt notwendig. Der Compiler darf den C-Code um den Inline-Assembler nicht optimieren. Verwenden Sie besser Assembler-Subroutinen.
4. Definieren Sie keine Funktionen mit variabler Parameterliste (wie etwa `printf`). Dies führt zu großem Overhead.

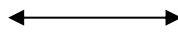
5. Ersetzen Sie sehr kurze Funktionen (1 bis 3 Zeilen Code) durch Makros oder Inline-Funktionen, dadurch wird der Overhead für den Funktionsaufruf eingespart. Die Codezeilen werden dadurch an die jeweilige Stelle im Programm "kopiert", was einerseits schneller, bei kurzen Funktionen aber sogar weniger Assembler-Instruktionen bedeuten kann.
6. Übergeben Sie einer Funktion immer Parameter, verwenden Sie keine globale Variable. Der Compiler wird für die Parameterübergabe Register verwenden. Das ist wesentlich schneller als der Zugriff auf globale Variable im Speicher. Oft werden die ersten Parameter in Registern übergeben, die restlichen auf dem Stack. Reduzieren Sie deshalb die Anzahl Parameter auf das Nötigste.
7. Wenn Sie Datenstrukturen als Parameter einer Funktion übergeben müssen, übergeben Sie einen const- Pointer auf die Struktur. Der Compiler würde sonst die ganze Datenstruktur auf den Stack kopieren.
8. Verwenden Sie das Schlüsselwort "volatile" um dem Compiler mitzuteilen, dass er eine Variable nicht in einem Register ablegen darf. Er wird diese immer im Speicher ablegen. Dies ist zwingend notwendig bei Variablen, die Hardware abbilden (z.B. bei Timer).
9. Prüfen Sie beim Compiler verschiedene Optimierungs-Level. Die höchste Optimierungsstufe bedeutet nicht zwingend, dass Sie den schnellsten oder kompaktesten Code erhalten werden.
10. Verwenden Sie für Variable immer die richtige Datengröße, abhängig von der verwendeten CPU (8, 16 oder 32 Bit). Der Compiler muss sonst einen oder mehrere cast durchführen. Auf einem 8-Bit Mikrocontroller ist das Rechnen mit char sehr effizient. Wenn Sie 16- oder 32-Bit Operationen durchführen, werden dazu Library-Funktionen verwendet, was nicht effizient ist. Überfordert ist ein 8-Bit Mikrocontroller mit Float-Berechnungen. Die Library-Funktionen dazu sind sehr groß und die Ausführungszeit sehr lang.
11. 32-Bit Mikrocontroller haben oft Alignment-Vorschriften. Definieren Sie in einer Datenstrukturen immer zuerst die 32-Bit Werte, dann die 16-Bit Werte und erst am Schluss die 8-Bit Werte falls dies möglich ist. Dadurch verhindern Sie, dass der Compiler Zwischenräume einfügen muss ("padding"), was die Datengröße der Struktur vergrößern würde.
12. Schreiben Sie keinen "cleveren" Code, den Ihr Kollege oder Ihre Kollegin nicht versteht (und Sie nach ein paar Tagen auch nicht mehr). Der Compiler wird diesen Code auch nicht verstehen und kann ihn deshalb nicht optimieren. Schreiben Sie den Code besser "straightforward".

**"Write simple and understandable code"!**



### 13.2 Symbole für Programmabläufe

Gegenüberstellung: Klassische Darstellung



Struktogramme

Struktur- element	Klassische Darstellung	Struktogramme
Schnittstelle		
Sequenz		
Verzweigung		
Wiederholung		

### 13.3 Anschlussplan Nucleo-F072RB

