C Programmier-Regeln



Regeln zur Code Gestaltung im QAM-Projekt

@author: Kevin Perillo

Inhaltsverzeichnis

[2 Vorwort 4](#_Toc115811062)

[3 Aufbau C-Programm 4](#_Toc115811063)

[3.1 Aufbau eines "normalen" C-Programms 4](#_Toc115811064)

[3.2 Aufbau eines "streng" modularen C-Programms 5](#_Toc115811065)

[4 Programmstruktur 6](#_Toc115811066)

[4.1 Namensgebung 6](#_Toc115811067)

[4.2 Einbindung von Modulen 6](#_Toc115811068)

[4.3 Variablen 7](#_Toc115811069)

[4.4 Datentypen 7](#_Toc115811070)

[5 Variablentypen und Geltungsbereich 8](#_Toc115811071)

[5.1 Lokale Variablen 8](#_Toc115811072)

[5.2 Statische lokale Variablen 8](#_Toc115811073)

[5.3 Globale Variablen 8](#_Toc115811074)

[5.4 Modulspezifische Variablen 8](#_Toc115811075)

[6 Funktionen 9](#_Toc115811076)

[6.1 Globale Funktionen 9](#_Toc115811077)

[6.2 Modulspezifische Funktionen 9](#_Toc115811078)

[7 Header-Datei (H-File) 10](#_Toc115811079)

[7.1 Includes 10](#_Toc115811080)

[8 Makros 10](#_Toc115811081)

[8.1 Einfache Makros ohne Parameter, Definitionen 10](#_Toc115811082)

[8.2 Komplexere Makros ohne Parameter 11](#_Toc115811083)

[8.3 Komplexere Makros mit Parameter 11](#_Toc115811084)

[9 Enumeration (Aufzählungen) 11](#_Toc115811085)

[10 Strukturen 12](#_Toc115811086)

[10.1 Verwendung von modulglobalen Strukturen 12](#_Toc115811087)

[11 Funktionsprototypen 13](#_Toc115811088)

[12 Implementierungsdatei (C-File) 13](#_Toc115811089)

[13 Konstante Variablen 13](#_Toc115811090)

[14 Kommentierung 14](#_Toc115811091)

[15 Codeblöcke 14](#_Toc115811092)

[16 Logische Operationen 14](#_Toc115811093)

[17 Ausdrücke 15](#_Toc115811094)

[18 Bedingte Übersetzungen (#ifdef … #else … #endif) 15](#_Toc115811095)

[19 Pointer 15](#_Toc115811096)

# Vorwort

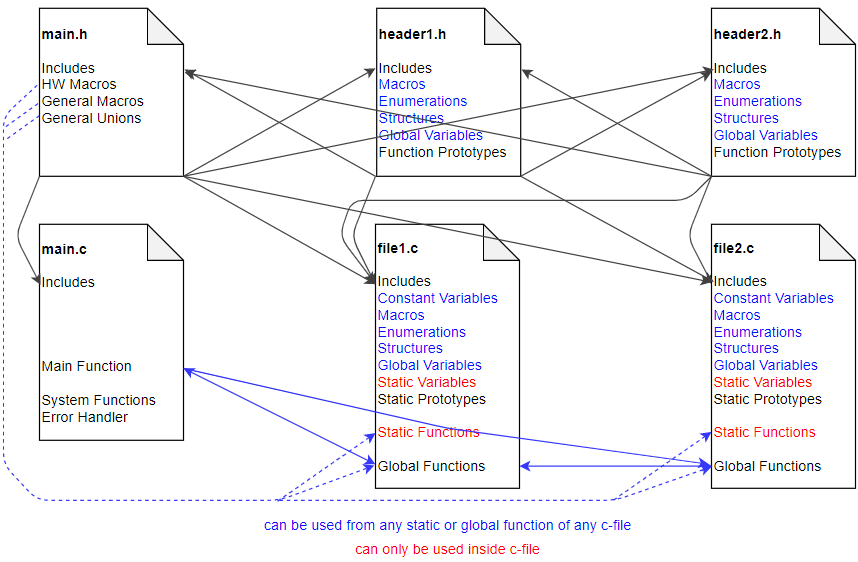
Programmierregeln sind eine qualitätssichernde Massnahme und dienen zur Verbesserung von Wiederverwendbarkeit, Verständlichkeit, Übersichtlichkeit, Änderbarkeit und Wartbarkeit von Modulen und Programmen sowie zur Fehlervermeidung beim Entwurf von C-Softwareprojekten.

Diese Programmierregeln sind bei allen C-Softwareprojekten im QAM-Projekt an der Juventus Techniker Schule anzuwenden.

# Aufbau C-Programm

## Aufbau eines "normalen" C-Programms

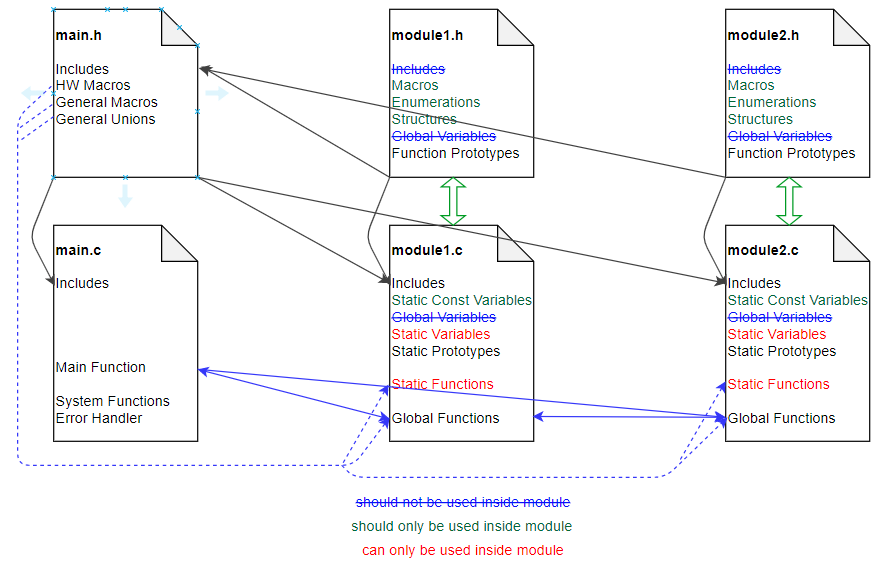
Der Programmaufbau basiert auf einem alltäglichen C- Programm



Die statischen und globalen Funktionen jedes einzelnen C-Files eines Programms können auf sämtliche globalen Elemente eines anderen C- oder H-Files zugreifen

## Aufbau eines "streng" modularen C-Programms

Der Programmaufbau basiert auf einem mit CubeMX erzeugten main-File für eine STM32 Anwendung.



Berücksichtigt man einige grundlegenden Vorgaben, so kann man auch in C modular programmieren, obwohl nach wie vor alle globalen Elemente eines anderen C- oder H-Files von jeder Funktion aus einem Modul heraus benutzt werden könnten

# Programmstruktur

## Namensgebung

* Im Modulnamen soll erkennbar sein, um was für ein Modul es sich handelt
* Der Name sollte möglichst kurz, prägnant und in englischer Sprache sein, z.B. **flash.c** und **flash.h** für die Bezeichnung eines Moduls, welches die Zugriffe auf einen FLASH-Speicher verwaltet
* Der Name des Moduls kann aus Kleinbuchstaben, Unterstrichzeichen '**\_**' und Zahlen **0** … **9** zusammengesetzt sein. Andere Zeichen sind nicht erlaubt.

## Einbindung von Modulen

Damit Module eines Programms im ganzen Programm sicht- und einsetzbar sind, müssen sie dem Programm bekannt gemacht werden. Dies geschieht am einfachsten, indem das Programm eine HeaderDatei **main.h** aufweisen muss, in welcher sämtliche Module mit **#include** bekannt gemacht werden müssen.

|  |  |
| --- | --- |
| **Beispiel** | /\*\*  \* File main.h  \* …  \*/  // Includes of standard libraries  **#include** <stdint.h>    // Integer data types  **#include** <stdio.h>     // I/O operations  **#include** <math.h>      // For common math. operations    // Includes of hardware specific modules (from CubeMX)  **#include** "usart.h"     // USART interface (created from CubeMX)  **#include** "spi.h"       // SPI interface (created from CubeMX)  **#include** "gpio.h"      // GPIO interface (created from CubeMX)    // Includes of modules  **#include** "debug.h"     // Debug module definitions  **#include** "led.h"       // LED module definitions  **#include** "dac.h"       // DAC module definitions  **#include** "flash.h"     // FLASH module definitions |

Als grosser Vorteil der konsequenten Bekanntmachung aller Module innerhalb von **main.h** erweist sich, dass in den einzelnen C-Files der verschiedenen Module nur noch **main.h** und das zum Modul gehörige File **module.h** einzubinden ist.

|  |  |
| --- | --- |
| **Beispiel** | // Includes of module flash.c  **#include** "main.h"   // Remaining module definitions  **#include** "flash.h"  // FLASH module definitions |

## Variablen

* Variablen besitzen einen Datentyp und einen Namen
* Der Name einer Variablen kann aus Klein- und Grossbuchstaben, Unterstrichzeichen '**\_**' und Zahlen **0** … **9** zusammengesetzt sein. Andere Zeichen sind nicht erlaubt.

Es ist nicht erlaubt nur Grossbuchstaben für den Variablennamen zu verwenden

* Die Deklaration einer Variablen macht dem Compiler den Namen der Variablen bekannt
* In den meisten Fällen wird eine Variable bei ihrer Deklaration auch gleich definiert. Der Compiler reserviert in Abhängigkeit des Datentyps, Speicherplatz für die Variable
* Initialisiert wird eine Variable, indem ihr ein Wert zugewiesen wird

|  |  |
| --- | --- |
| **Beispiel** | // Declaration, Definition and Initialization of variable humidity  **double** light;   // Declaration and Definition  light = 135.21;  // Initialization |

## Datentypen

* Jeder Variablen muss ein Datentyp vorangestellt werden
* Damit die Datenbreite weder vom Compiler noch vom verwendeten Prozessor abhängt, dürfen nur Integer-Datentypen aus der Header-Datei **<stdint.h>** verwendet werden
* Die Standard Header-Datei **<stdint.h>** muss in **main.h** eingebunden sein

|  |  |  |  |
| --- | --- | --- | --- |
| **Mögliche Datentypen** | | | |
| **Typname** | **Bitbreite** | **Wertebereich** | **Alias** |
| **int8\_t** | 8 | -128 … 127 | signed char **1** |
| **int16\_t** | 16 | -32768 … 32767 | signed int **1** |
| **int32\_t** | 32 | -2147483648 …  2147483647 | signed long int **1** |
| **int64\_t** | 64 | -9223372036854775808 ...  9223372036854775807 | signed long long **1** |
|  |  |  |  |
| **uint8\_t** | 8 | 0 … 255 | unsigned char **1** |
| **uint16\_t** | 16 | 0 … 65535 | unsigned int **1** |
| **uint32\_t** | 32 | 0 … 4294967295 | unsigned long int **1** |
| **uint64\_t** | 64 | 0 … 18446744073709551615 | unsigned long long **1** |
|  |  |  |  |
| **void** |  |  | "empty" type |
| **char** | 8 | 0 … 255 or -128 … 127 | use only for char arrays (strings) |
| **float** | 32 | 1.1756E-38 … 3.4028E38 | single-precision floating point |
| **double** | 64 | 2.225E-308 … 1.798E308 | double-precision floating point |
| **long double** | 80 | 3.362E-4932 … 1.190E4932 | extended precision floating point |
|  |  |  |  |
| **bool** | 1 | 0, 1 or false, true | bool variable **2** |

**1** needs **<stdint.h>** included

**2** needs **<stdbool.h>** included

* Für die Verwendung von Booleschen Variablen muss die Standard Header-Datei **<stdbool.h>** in **main.h** eingebunden werden

# Variablentypen und Geltungsbereich

## Lokale Variablen

Lokale Variablen sind nur in dem Anweisungsblock sichtbar und gültig, indem sie deklariert und definiert wurden. Nach Verlassen des Anweisungsblocks existiert die lokale Variable nicht mehr

## Statische lokale Variablen

Lokale Variablen, die mit dem Schlüsselwort "**static**" versehen sind, werden auf einer festen Speicheradresse gespeichert und behalten ihren Wert, auch nachdem der Anweisungsblock innerhalb dem die Variable definiert wurde, verlassen wurde.

## Globale Variablen

Globale Variablen sind Variablen die, gleich wie das Modul in welchem sie deklariert und definiert worden sind, im ganzen Programm sicht- und verwendbar sind

* Globale Variablen können im H-oder im C-File eines Moduls deklariert und definiert werden

Globale Variablen sollten in der modularen Programmierung möglichst vermieden werden

## Modulspezifische Variablen

Modulspezifische Variablen sind statische globale Variablen, deren Sichtbarkeitsbereich nur innerhalb des Moduls existiert, in welchem sie deklariert und definiert wurden

* Modulspezifische Variablen werden, ausserhalb aller Funktionen, innerhalb des C-Files des Moduls indem sie gültig sein sollen mit dem Schlüsselwort "**static**" deklariert und definiert

|  |  |
| --- | --- |
| **Beispiel** | //===============================================================  // Static variables    // Static auxiliary variable of module.c  **static double** value\_double; |

Für die modulare Programmierung können modulspezifische Variablen, zusammen mit den sie verwaltenden Getter- und Setter-Funktionen, optimal eingesetzt werden.

# Funktionen

* Funktionen besitzen einen Datentyp eine Parameterliste und einen Namen
* Der Name einer Funktion kann aus Klein- und Grossbuchstaben, Unterstrichzeichen ‘**\_**’ und Zahlen **0** … **9** zusammengesetzt sein. Andere Zeichen sind nicht erlaubt.

Nur Grossbuchstaben für Funktionsnamen zu verwenden ist nicht erlaubt!

* Funktionen existieren nur im C-File des Moduls
* Innerhalb eines Moduls existieren globale und statische (modulspezifische) Funktionen
* Funktionen besitzen einen Prototyp.
  + Prototypen von globalen Funktionen werden im H-File des Moduls aufgeführt
  + Prototypen von modulspezifischen Funktionen im Kopf des C-Files des Moduls

## Globale Funktionen

* Globale Funktionen sind Funktionen, die als Schnittstelle zum Programm dienen. Dazu gehören auch alle Getter- und Setter-Funktionen, die den Zugriff auf modulinterne Variablen und Strukturen erlauben

Globale Funktionen müssen den Modulname oder eine entsprechende Abkürzung im Funktionsnamen enthalten damit sofort ersichtlich ist, zu welchem Modul sie gehören

* In dem zum Modul gehörenden H-File müssen globale Funktionen als Prototyp deklariert und kurz beschrieben werden

|  |  |
| --- | --- |
| **Beispiele** | //=============================================================  // Prototypes    // Get identification of sensor  uint32\_t **get\_sensor\_id**( **void** );    // Set identification of sensor  **void** set**\_sensor\_id** (uint32\_t id); |

* Falls die Header-Datei eines Moduls richtig in **main.h** eingebunden wurde, sind globale Funktionen aus allen anderen Modulen eines Programms aufrufbar

## Modulspezifische Funktionen

* Modulspezifische Funktionen sind nur innerhalb des C-Files des Moduls, indem sie definiert wurden, sicht- und erreichbar
* Eine modulspezifische Funktion muss vor dem Funktionscode, innerhalb des C-Files, als Prototyp mit dem Schlüsselwort "**static**" deklariert und kurz beschrieben worden sein

|  |  |
| --- | --- |
| **Beispiel** | //=============================================================  // Static function prototypes    // Clear FLASH page  **static** void **flash\_erase\_page**( uint32\_t addr ); |

* Im C-File des Moduls werden die modulspezifischen Funktionen definiert und kurz beschrieben
* Der Definition der modulspezifischen Funktion muss das Schlüsselwort "**static**" vorangestellt werden

|  |  |
| --- | --- |
| **Beispiel** | //=============================================================  // Definitions of static functions    /\*   \* Erase FLASH page   \* - addr   Address of the selceted FLASH page   \* - return none   \*/  **static** void **flash\_erase\_page**( uint32\_t addr )  {     // Code …  } // End: flash\_erase\_page |

# Header-Datei (H-File)

Die Header-Datei (H-File) beschreibt nur die Schnittstellen zur Implementierungsdatei (C-File) des Moduls.

Das H-File eines Moduls besteht aus einem Header aufgebaut nach Doxygen-Richtlinien, einem Compilerschalter Makro für den Test, ob die Headerdatei bereits aufgerufen worden ist, **#include**-Anweisungen, Makros, Aufzählungen, Strukturen, globalen Variablendeklarationen und Funktionsprototypen der global benutzten Modulfunktionen.

Im QAM-Projekt werden beim Erstellen, der Headerfiles das Template: JS\_Header\_Template.h verwendet.

## Includes

Der Vorteil der konsequenten Bekanntmachung aller Module innerhalb von **main.h** macht es möglich, dass normalerweise keine zusätzlichen Header-Files in die Headerdatei des Moduls eingebunden werden müssen

# Makros

* Makros werden im Allgemeinen zur Definition von konstanten Werten oder für kurze Operationen, die häufig im Programm auftreten, verwendet
* Makros werden mit **#define** ins Programm eingeführt
* Makros werden vom Pre-Compiler verarbeitet und werden danach als Inline-Code in das Programm übernommen. Dies eliminiert den Overhead bei einem Funktionsaufruf.
* Makros können auch Argumente annehmen
* Makros sind im ganzen Programm sicht- und ausführbar
* Die Namen von Makros werden grundsätzlich mit Grossbuchstaben geschrieben
* Erlaubt sind auch das Unterstrichzeichen ‘**\_**’ und Zahlen **0** … **9**

## Einfache Makros ohne Parameter, Definitionen

* Einfache Makros ohne Parameter werden auch als Definitionen bezeichnet
* Definitionen erlauben es, Konstanten zu deklarieren und zu definieren, die im gesamten Programm gültig sind

|  |  |
| --- | --- |
| **Beispiel** | //=============================================================  // Macros (Definitions)    // Definitions for FLASH memory  **#define** FLASH\_PAGE\_SIZE      256  // FLASH page size (256 byte) |

Für den Gebrauch von konstanten Zahlenwerten innerhalb des Codes sind prinzipiell immer Makros zu verwenden – keine reinen Zahlenwerte

Dies gilt für: Grössendefinition von Arrays, Zuweisungen, if-, for-, while-, switch-Konstrukte, etc.

Obwohl Makros im ganzen Programm sicht- und anwendbar sind, sollten einfache Makros nicht in anderen Modulen eingesetzt werden als dem Modul, für das sie bestimmt sind. Auch dann nicht, wenn die gleichen Zahlenwerte in verschiedenen Modulen benutzt werden könnten.

## Komplexere Makros ohne Parameter

* Komplexere Makros ohne Parameter werden mit dem Präfix **m** versehen

|  |  |
| --- | --- |
| **Beispiel** | //=============================================================  // Macros (Definitions)    // Macro to switch green LED on  **#define** mLED\_GREEN\_ON     ( HAL\_GPIO\_WritePin( LED\_GREEN\_GPIO\_Port,                              LED\_GREEN\_Pin, GPIO\_PIN\_SET) ) |

Mit der Verwendung von komplexeren Makros ohne Parameter kann die Übersichtlichkeit von Programmen verbessert werden, indem einem komplexen Sachverhalt ein einfacher Begriff zugeordnet wird

## Komplexere Makros mit Parameter

* Komplexere Makros mit Parametern werden auch als funktionsähnliche Makros bezeichnet und mit dem Präfix **m** versehen

|  |  |
| --- | --- |
| **Beispiel** | //=============================================================  // Macros (Definitions)    // Macro: Add values a and b  **#define** mADD( a, b )    ( a + b )    // Select maximum of 2 variables  **#define** mMAX( a, b )    ( ( a ) < ( b ) ? ( b ) : ( a ) ) |

# Enumeration (Aufzählungen)

* Enumeration sind benutzerdefinierte Datentypen und werden hauptsächlich verwendet, um Namen für zusammengehörige Konstanten zu vergeben
* Die so definierten Namen machen ein Programm leicht lesbar und wartungsfreundlich
* Die Konstanten der Aufzählung und die Enumeration selber sind im ganzen Programm sicht- und einsetzbar, falls die Enumeration im H-File des Moduls definiert wurde
* Die Namen der Konstanten und der Enumeration werden grundsätzlich mit Grossbuchstaben geschrieben
* Erlaubt sind auch das Unterstrichzeichen '**\_**' und Zahlen **0** … **9**
* Enumeration werden mit dem Präfix **en** versehen

|  |  |
| --- | --- |
| **Beispiel** | //============================================================  // Enumerations    // Enumeration of the different LED Colors  **typedef** **enum** {  *RED*,    // 0: LED Color Red  *GREEN*,  // 1: LED Color Green  *BLUE*  // 2: LED Color Blue  } enCOLORS; |

Für den Einsatz von Enumeration sollte immer das Schlüsselwort **typedef** verwendet werden, damit die Aufzählung über ihren Namen benutzt werden kann

# Strukturen

* Eine Struktur ist ein Datentyp, der verwendet werden kann, um Elemente von unterschiedlichen Typen in einem einzigen Typ zu gruppieren
* Strukturen können innerhalb des C-Files des Moduls als statische Variablen oder als globale Variablen verwendet werden
* Die Namen der einzelnen Variablen innerhalb der Struktur und die Struktur selbst können aus Klein- und Grossbuchstaben, Unterstrichzeichen '**\_**' und Zahlen **0** … **9** zusammengesetzt sein
* Andere Zeichen sind nicht erlaubt
* Der Strukturname wird mit dem Präfix **st\_** versehen

|  |  |
| --- | --- |
| **Beispiel** | //=============================================================  // Structures    // Define structure of sensor data to be read  **typedef** **struct**  {    uint8\_t  mode;    // Mode of sensor    uint8\_t  status;       // Status of sensor    uint8\_t  sel;    // Sensor selection  **double**   humidity;      // Measured value of sensor (humidity)  **double**   temperature;   // Measured value of sensor (temp.)    uint16\_t crc;           // CRC bytes  } st\_sensor; |

## Verwendung von modulglobalen Strukturen

* Eine modulglobale Struktur ist eine Struktur, deren Variablen über einen Zeiger aus allen Funktionen des Programms aufgerufen werden kann
* Dies kann bei komplexen Programmen und Strukturen mit vielen Variablen den Zugriff auf die Struktur und deren Variablen vereinfachen
* Die modulglobale Struktur muss in der Headerdatei mit dem Schlüsselwort "**extern**" deklariert werden
* Die effektive Definition der modulglobalen Struktur wird erst im C-File des Moduls ausgeführt

# Funktionsprototypen

* Funktionsprototypen innerhalb der Headerdatei stehen für globale Funktionen, die im gesamten Programm sichtbar und ausführbar sind

# Implementierungsdatei (C-File)

Das C-File eines Moduls besteht aus einem Header aufgebaut, falls nötig **#include**-Anweisungen, Konstantendefinitionen, globale und statische Variablendeklarationen, modulspezifische (statische) Funktionsprototypen und die Funktionsdefinitionen für die deklarierten statischen und globalen Funktionen.

* Im C-File des Moduls wird grösst mögliche Datenkapselung betrieben
* Es darf nur über die dafür vorgesehenen Schnittstellenfunktionen (Getter- und Setter-Funktionen) auf Daten anderer Module zugegriffen werden
* Im C-File werden nur modulinterne Definitionen und Deklarationen beschrieben, z.B. die in der Header-Datei deklarierten Funktionen (Prototypen)
* Es dürfen keine Makros (**#define**) und Aufzählungstypen (Enumerations) deklariert werden

# Konstante Variablen

* Eine konstante Variable ist eine Variable, deren Wert nach der [Initialisierung](http://new.c-howto.de/tutorial/variablen/deklaration#64/) nicht mehr geändert werden kann. Man kennzeichnet sie mit dem Schlüsselwort "**const**"
* Konstante Variablen besitzen einen Datentyp, den der Compiler bei der Definition prüft
* Konstanten Variablen muss bei der Deklaration ein Wert zugewiesen werden, ansonsten gibt der Compiler eine Fehlermeldung aus
* Konstante Variablen sind dort sichtbar und gültig, wo sie deklariert, definiert und initialisiert wurden – gleich wie bei den Variablen
* Ist es in einem Programm nötig grössere Tabellen einzusetzen, so sollten diese Tabelle zu Beginn des C-Files eingesetzt werden
* Die Namen von konstanten Variablen werden grundsätzlich mit Grossbuchstaben geschrieben.
* Erlaubt sind auch das Unterstrichzeichen '**\_**' und Zahlen **0** … **9**
* Der Name einer konstanten Variablen wird mit dem Präfix **c** versehen

|  |  |
| --- | --- |
| **Beispiele** | // Table to calculate CRC16 values  **const** **static** uint16\_t cCRC16\_LOOKUP\_TABLE[CRC\_256] = {    0x0000, 0xC0C1, 0xC181, 0x0140, 0xC301, 0x03C0, 0x0280, 0xC241,    0xC601, 0x06C0, 0x0780, 0xC741, 0x0500, 0xC5C1, 0xC481, 0x0440,    0xCC01, 0x0CC0, 0x0D80, 0xCD41, 0x0F00, 0xCFC1, 0xCE81, 0x0E40,    …    0x4400, 0x84C1, 0x8581, 0x4540, 0x8701, 0x47C0, 0x4680, 0x8641,    0x8201, 0x42C0, 0x4380, 0x8341, 0x4100, 0x81C1, 0x8081, 0x4040  };    **const** **double** cPI = 3.14159;  // PI |

# Kommentierung

C ist eine Hochsprache, dadurch ist es nicht notwendig, alle Code-Zeilen zu kommentieren. Die Kommentare dienen zu einer besseren Lesbarkeit und Übersichtlichkeit von Programmen.

* Kommentiert werden muss "was" geschieht, "wie" es gemacht wird, wie Parameter zu interpretieren sind
* Alle Funktionsprototypen erhalten einen Blockkommentar, der ihre Aufgabe erklärt sowie Rückgabe und Übergabeparameter beschreibt
* C-Kommentare, die sich auf eine Zeile beziehen, sind entweder über dieser zu schreiben (Ausnahme bei Variablen) oder am Ende der Zeile
* C-Kommentare mit nur einer Zeile bzw. nach Code am Ende einer Zeile darf mit "**//**" gekennzeichnet werden, sonst muss "**/\***   **\*/**" verwendet werden
* Kommentare dürfen nicht geschachtelt werden

# Codeblöcke

* Zusammengehörige Textteile nach Konstrukten wie **for**, **if**, **while**, usw. werden als Blöcke implementiert. Jeder Block beginnt und endet mit geschweiften Klammern
* Die Klammern werden vom restlichen Text getrennt geschrieben, jede Klammer auf einer eigenen Zeile
  + Die Klammern haben denselben linken Einzug wie der Konstrukt **for**, **if**, **while**, etc.
* Der Blocktext hat einen linken Einzug von mindestens drei Leerzeichen von den Blockklammern
* Die Verwendung von Tabulatoren zum Einzug sind grundsätzlich zu vermeiden - stattdessen sind Leerzeichen zu verwenden

|  |  |
| --- | --- |
| **Beispiel** | /\* Comment, what to do ... \*/  **if**( Condition )  {     // Comment then branch ...     then code ...  }  **else**  {     // Comment else branch ...     then code ...  } |

* Blöcke aus einer Zeile dürfen ohne Klammern und unmittelbar nach dem Konstrukt geschrieben werden

|  |  |
| --- | --- |
| **Beispiel** | // Clear array values  **for**( i = 0; i < MAX\_VALUE; i++ ) array[i] = 0; |

# Logische Operationen

* Bei logischen Operationen muss immer berücksichtigt werden, dass man in C die Benutzung von Variablen und Rückgabewerten von Funktionen direkt, d.h. ohne Vergleich, in logischen Abfragen und Verknüpfungen einsetzen kann
* In C ist als logisch "falsch" der Wert 0 definiert. Als logisch "wahr" gilt jeder Wert ungleich Null
* Variablen, Parameter und Rückgabewerte von Funktionen, die einen logischen Zustand ausdrücken, sollten vom Datentyp **uint8\_t** sein
* Der Datentyp Bool kann verwendet werden, sollte aber auf Grund von bestimmten Compilern nicht verwendet werden.

# Ausdrücke

* Es ist sicherer bei Vergleichen mit Makrokonstanten zuerst die Makrokonstante zu verwenden. Eine Verwechslung von "**==**" und "**=**" wird dadurch vom Compiler erkannt

|  |  |
| --- | --- |
| **Beispiel** | // Check state  **if**( STATE\_OK == state ) |

* Eine Verschachtelung von Zuweisungen ist zu vermeiden

|  |  |
| --- | --- |
| **Beispiel** | // Calculate  ~~d = (a = b + c ) + r;~~ |

* **goto** Anweisungen sind nicht erlaubt – ausser in gewissen Spezialfällen, in welchen die Verwendung von **goto** für das Auffangen von Fehlern nicht verhindert werden kann.
* **break** in Schleifen ist erlaubt, wenn sich daraus eine Vereinfachung beim Lesen des Codes ergibt
* Bedingungen müssen lesbar sein
  + Doppelte Verneinung ist zu vermeiden
* Man kann sich nicht auf die automatische Typenkonvertierung verlassen. Der casting-Operator sollte bei der Typenkonvertierung immer verwendet werden

# Bedingte Übersetzungen (#ifdef … #else … #endif)

* Compilerschalter-Makros dürfen nur zur Verwendung von mehrfachem Einfügen von Header-Dateien, zur Variantenbildung und für das Debugging verwendet werden

|  |  |
| --- | --- |
| **Beispiel** | // Global directive  **#ifdef** G\_DEBUG\_ON    debug\_printf( "Program started" );  **#endif** |

# Pointer

Der Einsatz von Pointer innerhalb der modularen Programmierung sollte auf das Minimum beschränkt werden!