C-Coding Styleguide

-Programmierrichtlinien-

Hochschule Offenburg / Insa Strasbourg

Léa Bonnardot
Samuel Engel
Matthéo Bottos
Mateis Denot
Théo Portmann
Johan Maring
Thomas Eyer

Allgemeines		
A1	Sourcecode wird in **englischer Sprache** verfasst. Das gilt z.B. für Kommentare, Variablen- und Funktionsnamen.	`// User has to enter the max.` `value int iMaxUserValue; // Check against limits` `BOOL IsMaxValue(int iEnteredValue);`
A2	Faustregel: Pro 10 Programmzeilen 1-3 Zeilen Kommentar! (10-30%) Auch leere Zeilen tragen zur Lesbarkeit bei. Es sind **keine Trivialitäten** zu dokumentieren. Alles, was sich aus dem Programmcode für einen mittelmäßigen Programmierer sofort ergibt, ist nicht zu dokumentieren! .	`// increment i -> trivial, don't comment this` `i++;` `// Is source of timer interrupt match` `// register 2?` `if (LPC_TIM0->IR & (1<<2))`
A3	Kommentare beginnen mit einem //. Mehrzeilige Kommentare /* */ sind nicht erlaubt. Ausnahme: doxygen-Kommentare! *Bei projektweiter Suche ist im Suchergebnis sofort ersichtlich, welche Zeile auskommentiert ist und welche nicht.*	`// this is okay` `/* this is not okay */`
A4	Ein C-Projekt ist in viele *.c- und *.h-Dateien aufzuteilen. Dateinamen beginnen mit einem Großbuchstaben. *Die Wiederverwendbarkeit der Module wird dadurch erhöht (Baukastenprinzip).*	`Nqueensmain.c` `Recalgo.c` `Recalgo.h`
A5	Eine Funktion darf maximal nur 4-40 Programmzeilen enthalten.	
A6	Eine C-Datei darf maximal nur 4-400 Zeilen enthalten	

A7	Eine C-Datei (Modul) ist wie folgt aufgebaut: 1. Dateikopf (Doxygen Kommentar) 2. Systemheaderdateien (z.B. #include <stdio.h>) 3. User Headerdateien (z.B. #include "Eightqueen.h") 4. Modulspezifische Datentypen Defines, Konstanten, Enumerationen, Structs, Unions, Bitfields, Typedefs, Makros 5. Externe und globale Variablen (Ausnahmefall) 6. Funktionsdeklarationen für Funktionen 7. Setup function 8. Loop function 9. Functionsimplementierung *Damit ein Modul übersichtlich bleibt, werden teilweise die hier aufgelisteten Elemente in Header-Dateien ausgelagert.*	``` /*** * @file Test.c * @brief Test.c file to explain how it works * @author Thomas Eyer * @date 18.04.2024 *** #include <stdio.h> #include <conio2.h> #include "Eightqueen.h" #define SIZEOFBOARD 8 typedef unsigned int uint32_t; const double cdPI = 3.1415926535; enum eModes { continuous, stop }; extern int32_t s32SolutionsFound; int32_t s32SetNextQueen(int32_t* ps32Value); void vPrintBoard(struct nQueen* psNQueen); void setup() { //more code } void loop() { //more code } void vPrintBoard(struct nQueen* psNQueen) { } ```

A9	Keine Verwendung von Variablen mit projekt- oder modulglobaler Gültigkeit. Stattdessen z.B. in loop die verwendeten Variablen definieren und diese an andere Funktionen mit „Call by Reference“ übergeben.	`void loop()` `{` `  uint8_t u8Value;` `  vfunc1(u8Value);  // Call by value` `  vfunc2(&u8Value),  // Call by reference` `}`
A11	Für komplexe Datentypen, Variablen und Funktionen sind aussagekräftige Namen zu vergeben. Der erste Buchstaben soll der „return Value“ angeben: *Code wird lesbarer und ersetzt ggf. einen Kommentar.*	`int16_t s16GetGreatestValue(int8_t s8ValA, int_8t s8ValB);` `//function return`

Code layout		
CL1	The opening { and closing } curly brackets always start in a new line.	`for (ui = 0; ui < 10; ui++)` `{` `  // Code` `}`
CL2	The code block must be indented by **three** spaces within the curly brackets. This also applies after case labels in a switch/case control structure, when declaring complex data types and their elements, as well as for Comments. (see rule A1: Conversion of tabs to spaces)	`struct SyncMechanism` `{` `  long int liTimeStamp;` `  int iNumberOfIntervals;` `};`
CL3	The keywords if, while, for, switch are followed by a space.	`for ()` `if ()`
CL4	A function name is immediately followed by the opening round bracket ((declaration, definition, call).	`Call250usTask();  //Good` `Call250usTask  (); //Bad`
CL5	**A space** must be inserted before and after the relational (<, <= ,> ,>= ,== , !=), arithmetic (*, /, +, -, %) and logical operators (&&, ||).	`if ((s8A < 10) && (s8A != s8C))`

CL6	With switch, while, for, do/while, if, else, else/if, the curly brackets { } are placed directly under the first letter of the keyword. Also use switch as possible instead of if else	`switch (s8Selection)` `{` `  case 10:` `  while (i < 10)` `  {` `  i = GetStatusOfPort();` `  }` `  break;` `  case 20: //fall-through is ok here` `  case 30:` `  //Commands selection if s8Selection is 20 or 30` `  break;` `  default:` `  break;` `}`

<table>
<tr><th colspan="4">Data types, variables, constants and expressions</th></tr>
<tr><td>DV1</td><td>Variables and constants have self-explanatory names in English and begin with a capital letter. The data type should be added to the name as a prefix (short form).</td><td colspan="2"><pre>int8_t s8Trials = 1; //prefix is s8
const float cfPI = 3.1415f; // prefix is cf for const float</pre></td></tr>
<tr><td>DV2</td><td>The native data types are no longer used, but the typedefs from stdint.h (C99). If stdint.h is not available, this must be mapped in a separate header file. uint8_t, uint16_t, uint32_t, uint64_t for unsigned variables int8_t, int16_t, int32_t, int64_t for signed variables

Separate typedefs must be used for float, double and long double:
f32_t, f64_t, f96_t, f124_t long double is platformdependent. Floating point numbers should consist of performance on a microcontroller without FPU should be avoided.</td><td colspan="2"><pre>#include <stdint.h>

//if stdint.h not available
typedef unsigned int uint32_t;

typedef float f32_t;
typedef double f64_t;
typedef long double f96_t;</pre></td></tr>
<tr><td>DV3</td><td>The short identifier must be used as a prefix for variable names.
int16_t s16 int8_t s8
uint16_t u16 uint8_t u8
int32_t s32 f32_t f32
uint32_t u32 f64_t f64
int64_t s64 f96_t f96
uint64_t u64 _Bool (C99 native) b
The macros for bool, true and false can be found in <stdbool.h> to ensure compatibility with C++.</td><td><pre>int16_t s16Counter;
uint16_t u16Counter;
int32_t s32Counter;
uint32_t u32Counter;
int64_t s64Counter;
uint64_t u64Counter;</pre></td><td><pre>int8_t s8Byte;
uint8_t u8Byte;
f32_t f32Average;
f64_t f64Average;
f96_t f96Average;
// C99 native boolean
_Bool bErrorflag1;
// stdbool.h
bool bErrorflag2;</pre></td></tr>
</table>

DV5	When defining a pointer variable/constant, do not insert a space between the data type and the *.	`int8_t * ps8Val1;  // not okay` `int16_t* ps16Val2;  // okay`
DV6	For constants, add a c in front of the prefix. Pointers must be preceded by p and arrays must be preceded by a. A void pointer must be preceded by a pv as the entire prefix (special role, no void variable possible)	`const f64_t cf64PI = 3.1F;  int16_t as16Semester[7];` `int32_t*  ps32Sum;` `char* pacSymbols[];`
DV7	As an exception, i, j, k, l, m and n can be used for running variables.	`int i;`
DV10	**Hexadecimal constants** always start with 0x (no capital O!!!). Capital letters must be used for A-F**.**	`const uint16_t cu16Value = 0x1AF3;`
DV11	**Octal constants** (leading 0) should not be used.	`uint16_t us16Val = 072; //072 is octal constant`
DV13	Implicit **casts** (casts by compiler) for assignments are to be avoided! Only use explicit casts to show that this is intended.	`s32Val = f16Val; // implicit cast - not okay` `s32Val = (int32_t)f16Val; // explicit cast - okay`
DV14	**Pointers must be** initialised with NULL before they are used for the first time, unless a valid address is assigned to the pointer in an initial definition. *If this is done consistently (with rule DV16), it can always be determined whether a pointer is valid. NULL is defined as ((void*)0): #define NULL ((void*)0)*	`int32_t* ps32Memory = NULL;` `// check if pointer is valid if (ps32Memory != NULL)`
DV15	Always use the sizeof operator (no absolute values) for **malloc.** *malloc returns a pointer of type **void*.** Explicit cast required in C++*	`ps32Memory = (int32_t*)malloc(100 * sizeof(int32_t));`
DV16	After a free, the pointer must also be set to NULL in order to mark it as invalid again.	`free(ps32Memory);` `ps32Memory = NULL;`
DV4	Only one variable/constant must be declared/defined per line.	`int8_t* ps161, ps162; // not okay`

DV17	**Embedded assignments should be** avoided. Embedded assignments must be split accordingly. *This makes the programme code easier to read and less prone to errors.*	`//not okay if ((s32Total = GetTotal()) == 10) { printf("do it\n"); }`	`// okay s32Total = GetTotal(); if (s32Total == 10) { printf("do it\n"); }`
DV18	Signed and unsigned variables/constants may not be used in an instruction/condition.	`if (u32Val1 > s32Val2) //possible wrong behaviour`	
DV19	The bit-oriented operators (>> and <<) may not be used on signed variables.	`s32Val2 = s32Val2 >>1U; //not okay if iVal2 is negative` `u32Val3 = u32Val3 >>1U; //okay`	

Complex data types	
Variables of complex data types enum, union, struct, bitfield are prefixed: enum e union u struct s bitfields bf	`enum States eProgrammStates; union MeasureTime uLastTime; struct Address sCustomerAddress; struct Flags bfNewFlags;`
The same applies to complex data types: For constants, add a c in front of the prefix. For arrays, precede the prefix with an a and for pointers with a p.	`const enum States ceProgrammStates; struct Address asCustomerAddress[100]; struct Address* psCustomerAddress;` `const struct Name csDefaultName = {"Hans", "Mustermann"};`

Functions		
F3	The optional variable names must be specified in the function definition if parameters are passed. *This makes it easier to understand when reading the function definition.*	`ints32_t s32GetNumberOfTrials(void); //Okay int32_t GetNumberOfTrials(void); //Not Okay`
F4	Only one return per function is permitted, and this must be the last statement.	
F5	A function that does not return a value (void) does not have a return statement.	
F7	Formal parameters that are only used read-only within a function must be passed with the keyword const	`uint16_t u16CalcSpeed(const uint16_t cu16Val)` `{` `  return (10 * cu16Val);` `}`
F8	Structures must never be transferred by value. *These would otherwise require too much memory on the stack.*	`int32_t CalcMedian(struct VeryBig smybig); //not okay int32_t CalcMedian(struct VeryBig* const pcsmybig); //okay`

F11	Self-explanatory function names must be used. The function name is composed as follows: <short code return value><module name><_><verb><other specifying words>. The module name is the name of the C file in which the function is implemented. This is followed by a subscore, a verb and further specifying words in English. The verb and specifying words each begin with a capital letter. Typical verbs here are "Get", "Set", "Init", "Delete", "Copy", "GetNext", "GetLast", "Print", "Is", ... Use the short identifier for the return values: see DV3 If pointers are returned, the short identifier must be preceded by a p. If your own signed data types (typedefs) are returned, an **x must be** used as a prefix. If it is an unsigned data type, a **ux must be** used as the prefix. *Function names are structured accordingly in FreeRTOS, which is used in Embedded Systems 2*. **Exception:** Names of the ISR (interrupt service routines). Their names are already specified in the startup.s in the interrupt vector table.	`// File PWM.c` `uint32_t u32PWM_GetSignalGPIO (uint8_t u8SignalSource);` `psPWMConfig pxPWM_GetConfig (uint8_t u8SignalSource);` `void SysTick_Handler(void)`
F13	If pointers are passed to a function, they must be checked for != NULL at the start of the function (**sanity check**) and correct error handling must be implemented.	`void DoCallByReference(int32_t* ps32A)` `{` `  if (ps32A!= NULL)` `}`

<table>
<tr><th colspan="4">Control structures</th></tr>
<tr><td>K1</td><td>Jumps with goto are not permitted. Accordingly, labels are not necessary except for switch/case control structures.</td><td colspan="2"><pre>goto Label1; // not okay</pre></td></tr>
<tr><td>K3</td><td>A switch/case control structure always has a default as the last case (error output etc.). This last case also contains a break.</td><td colspan="2"><pre>default: break;</pre></td></tr>
<tr><td>K4</td><td>A fallthrough (missing break) in a switch/case control structure must be documented in detail.</td><td colspan="2"><pre>case 1: //
fallthrough to
case 2,
because same
behaviour case
2:</pre></td></tr>
<tr><td>K5</td><td>Omitting parts of the loop body using continue is not permitted. This can be achieved using a customised control flow.</td><td colspan="2"><pre>//not okay
while (iStatus < 10)
{
   iStatus = GetStatus();
   if (iStatus == 4)
   {
      continue; // not okay
   }
   // other statements
}</pre></td></tr>
<tr><td>K6</td><td>Cancelling the program with exit() is only possible in justified exceptional cases (safety or emergency). Otherwise, exit should generally be avoided.</td><td><pre>if (iEmergency == 1)
{
exit(0);
}</pre></td><td><pre>//okay
while (iStatus < 10)
{
   iStatus = GetStatus();
   if (iStatus != 4) //okay
   {
      // other statements
   }
}</pre></td></tr>
<tr><td>K7</td><td>The conditional assignment ?: (ternary operator) should be avoided if this can be expressed more clearly (as an if-else query). Otherwise, use a comment to describe what is to be achieved with the conditional assignment.
Conditional assignment is often found in function macros.</td><td colspan="2"><pre>iZ = (iA > iB) ? iA : iB; // z = max(a, b)</pre></td></tr>
</table>

K8	**elseif** is **not** allowed for reasons of better readability! *This can simply be replaced by brackets {} and an if.*	`// not okay if` `(iVal == 1)` `{` `  iVal = 42;` `}` `elseif (iVal == 2)` `{` `  iVal = 79;` `}`

K9	A code block that consists of only one line must also be bracketed. The outermost code block of a case in a switch/case control structure must not be bracketed.	`if (iVal == 1)` `{` `  iVal = 42;` `}`	`// okay if` `(iVal == 1)` `{` `  iVal = 42;` `}` `else` `{` `  if (iVal == 2)` `  {` `  iVal = 79;` `  }` `}`
K10	The rules of "**structured programming"** must be observed. §1 A function has one input and one output. §2 Do not jump to queries. §3 Do not jump out of queries. §4 Do not jump into loops. §5 Do not jump out of loops. *Only structured functions can be realised with structure diagrams (NassiShneiderman diagrams).*		

K11	Short forms (C standard: != 0 for true, 0 for false) are to be avoided. This implicit knowledge *(!= 0 for true)* should not be assumed. *The C89 standard does not yet have true and false.*	`if (iVal) // better: if (iVal != 0)`
K12	If the number of loop passes before the loop is known, the for loop should be used instead of the while loop. *While loops are generally more error-prone and somewhat more difficult to read.*	
K13	Loop variables (run variables) must not be changed in the loop body of a for loop.	`for (i = 0; i < 100; i++)` `{` `  //more code` `  i = s32UserInput; // not okay` `  //more code` `}`

K14	Floating point variables (double and float) may not be used as loop variables (run variables). *Floating point numbers are subject to rounding inaccuracies (IEEE754).*	`for (f32Val = 0.0F; f32Val < 10.0F; f32Val += 0.01F) //not okay` `for (s32Val = 0; s32Val < 1000; s32Val ++) // okay`
K15	Floating point variables (double and float) must never be checked for equality. *Floating point numbers are subject to rounding inaccuracies (IEEE754).*	`if (dVal == 3.5517) // not okay` `const double cdTolerance = 0.0001;` `double dDifference;` `dDifference = fabs(dVal - 3.5517);` `if (cdTolerance > dDifference) // okay`

Preprocessor		
P1	Each header file (e.g. PWM.h) must be protected against multiple inclusion using include-Guard. The name of the header file must be used as shown on the right. *This means that each h-file is only included once when translating **a** C-file.* *Often h-files are included several times via other h-files.*	`#ifndef _PWM_H` `#define _PWM_H` `// all code` `#endif //_PWM_H`
P3	**#defines should be** capitalised. *This makes it easier to distinguish them from variables.*	`#define NUMBEROFFILES 5`
P4	If defines are used for conditional compilation, SW_ must be used as a prefix in the name.	`#define SW_LINUX` `#define SW_GLCD`
P5	To minimise side effects, macros should be implemented as follows • The entire macro must be bracketed • Each macro parameter must be bracketed • Macro parameters and the entire macro must be cast to the correct type	`//Swap the Bytes of a 16-Bit Value` `#define SWAPBYTES16(x) ((uint16_t)(((uint16_t)(x) << 8) | \` `((uint16_t)(x) >> 8)))`
P6	More extensive macros can be divided into several lines for easier reading. A backslash separates the macro lines from each other.	`see P5`

P7	In order to rule out further side effects, inline functions should be preferred to macros. These offer better type safety and avoid unwanted side effects. *Although inline functions appear as functions in the source code, the function body is usually copied directly to the location of the function call by the compiler. This increases the scope of the machine code - performance increases. In order to be able to call the inline functions outside the module, a "correct" implementation is provided. However, inline is only a hint to the compiler. Extensive functions are not "inlined" despite the inline keyword.*	`// MACRO: Sideeffect with char cval = 'Z' AND` `// usage with ISUPPER(cval++)` `#define ISUPPER(c) (((c) >= 'A') && ((c) <= 'Z'))` `// Inline function: No Sideeffect with char cval = 'Z' AND` `// usage with isUpper(cval++)` `#include <stdbool.h>` `inline bool isUpper(char cval)` `{` `  bool bret = false;` `  if ((cval >= 'A') && (cval <= 'Z'))` `{ bret = true;` `  }` `  return bret;` `}`
P8	If possible, symbolic constants should be replaced by constant variables.	`#define PI 3.1415 //avoid const f64_t` `= 3.1415; //better`

Doxygen		
D1	Right from the start, **doxygen** is to be used for documentation for File headers, functions, structures, enumerations, unions and bit fields.	see www.doxygen.org
D2	Each file contaioldest a **file header** that can be interpreted by doxygen. @file @letter Name of the file @author Brief description of the module @date Name of the module creator Creation date	/*** * @file MyFirstCFile.c * @brief File contains the main programme * @author Daniel Fischer * @date 01.08.2010 * * Additional Infos: This module is certified... ***/
D3	Any function contains a **function header** that is interpreted by doxygen. can be. @fn Function declaration @letter Brief description of the function @param Name and description of the formal parameter and only if available @return Data type and description of the return value, and only if not equal to void @author Name of the programmer @date Creation date	/** * @fn int CalcGreatest(int iA, int iB) * @brief Calculates the greatest value * @param iA first value * @param iB second value * @return int greatest value of iA and iB * @author Daniel Fischer * @date 01.08.2010 * * The greatest value is calculated based on the * given two parameters. If both are equal... */
D4	Doxygen must be configured so that *call graphs* and *include dependency graphs* are displayed graphically. *The GraphViz tool must be installed for this purpose. See the corresponding instructions.*	