C Coding Style, Conventions & Rules

Alexander Rössler

4. Mai 2013

Inhaltsverzeichnis

1 Formatierung

1.1 Einrückung

- 4 Leerzeichen werden zum Einrücken verwendet
- Leerzeichen!!!

1.2 Whitespace

- Verwende Leerzeilen um zusammenhängende Befehle zu strukturieren
- Es wird immer nur eine Leerzeile benutzt
- Nach einem Schlüsswort und vor einer geschwungenen Klammer wird ein Leerzeichen verwendet

```
1     // Falsch
2     if(foo)
3     {
4     }
5
6     // Richtig
7     if (foo)
8     {
9     }
```

- Bei Funktionen wird hinter jeden Parameter ein Leerzeichen gemacht.

```
1     // Falsch
2     myFunction(parameter1,parameter2,parameter3);
3
4     // Richtig
5     myFunction(parameter1, parameter2, parameter3);
```

- Ausdrücke werden nicht in einer Wurst geschrieben.
- Leerzeichen werden eingefügt so das der Ausdruck leicht leserlich ist.

```
1     // Falsch
2     result=2*pi+x/3;
3
4     // Richtig
5     result = 2*pi + x/3;
```

1.3 Geschwungene Klammern

- Geschwungene Klammern kommen immer in eine eigene Zeile
- Geschwungene Klammern werden auch bei Bedingungen mit nur einer Zeile gemacht

```
1     // Falsch
2     if (foo) {
3     }
4
5     if (bla)
6         x++;
7
8     // Richtig
9     if (foo)
10    {
11    }
12
13    if (bla)
14    {
15        x++;
16    }
```

- Geschwungene Klammern werden auch bei leeren Schleifen verwendet

```
1     // Falsch
2     while (1);
3
4     // Richtig
5     while (1) {}
```

1.4 Runde Klammern

- Ausdrücke werden durch runde Klammern gruppiert

```
1     // Falsch
2     if (a && b || c)
3
4     // Richtig
5     if ((a && b) || c)
```

1.5 Switch

- Die case Labels sind auf gleicher Höhe wie das switch Labels
- Jeder case muss einen break haben oder anzeigen das er keinen break hat. Ausnahme: Ein weiterer case folgt direkt.

```
    switch (myEnum) {
    case Value1:
        doSomething();
        break;
    case Value2:
    case Value3:
        soSomethingElse();
        // fall through
    default:
        defaultHandlin();
        break;
```

1.6 Zeilenumbrüche

- Halte Zeilen kürzer als 100 Buchstaben
- Zeilenumbrüche werden überall verwendet wo es den Code übersichtlicher macht
- Beistriche kommen ans Ende einer umgebrochenen Zeile. Operatoren kommen am Anfang der neuen Zeile. Ein Operator ist leicht zu übersehen wenn der Editor zu schmal ist.

```
    // Falsch
    if (longExpression +
        otherLongExpression +
        otherOtherLongExpression)
    {
    }

    // Richtig
    if (longExpression
        + otherLongExpression
        + otherOtherLongExpression)
    {
    }
```

1.7 Kommentare

- Verwende nur Zeilenkommentare
- Kommentiere überall wo es Sinn macht
- Versuche Kommentare durch vernünftige Namensgebung überflüssig zu machen
- Schreibe Kommentare so das sich auch Andere auskennen
- Verwende ein Leerzeichen hinter den //
- Rücke Kommentare so ein das sie leicht leserlich sind

```
// Falsch
someExpression; //BlaBlaBla
someOtherExpression; //BlaBlaBla

// Richtig
someExpression;             // BlaBlaBla
someOtherExpression;        // BlaBlaBla
```

1.8 Dokumentation

- Zur Dokumentation des Codes wird Doxygen verwendet.
- Funktionen werden dokumentiert.
- Enumeratoren werden dokumentiert.
- Es wird alles Dokumentiert was notwendig erscheint.

```
/** Short Description
  * @param width The width of the new rectangle.
  * @param height The height of the new rectangle
  * @return Returns 0 on success or -1 on failure.
  */
createRectangle(int width, int height);
```

2 Namensgebung

2.1 Variablen

- Jede Variable in einer neuen Zeile
- Kurze sinnlose Namen wie zum Beispiel “rbar” vermeiden
- Einzelne Buchstaben sind nur in Ordnung für Zähler und temporäre Variable wo der Zweck klar ist
- Variablen werden dort deklariert wo sie verwendet werden, zum Beispiel in einer Schleife, Funktion, ...

```
// Falsch
int a, b;
char *c, *d;

// Richtig
int height;
int width;
char *nameOfThis;
char *nameOfThat;
```

- Variablen starten mit einen kleinen Buchstaben, jedes Wort im Variablennamen startet mit einem großen Buchstaben
- Keine Abkürzungen wie zum Beispiel “rstButten” verwenden
- Anbkürzungen werden klein geschrieben, zum Beispiel setXml statt setXML

```
// Falsch
short Cntr;
char ITEM_DELIM = '\t';

// Richtig
short counter;
char itemDelimiter = '\t';
```

2.2 Konstanten und Enumeratoren

- Werden UpperCamelCase geschrieben.
- Jede Enumerator Konstante enthält den Enumerator Namen.
- Enumerator werden zur besseren Lesbarkeit des Codes verwendet.
- Konstanten werden zur einfachen Bearbeitung von zur Laufzeit statischen Parametern verwendet.
- Enumeratoren werden mit typedef deklariert.

```
// Falsch
//
void myFunction(int option); // 0 = enable, 1 = disable

// Richtig
typedef enum {
    MyFunctionOption_Enable = 0,
    MyFunctionOption_Disable = 1
} MyFunctionOption;

void myFunction(MyFunctionOption option);
```

- Globale Enumeratoren und Konstanten müssen den Bibliotheksnamen enthalten.

```
typedef enum {
    Timer_Clock_1MHz = 0b00,
    Timer_Clock_2MHz = 0b01,
    Timer_Clock_5MHz = 0b10,
    Timer_Clock_10MHz = 0b11
} Timer_Clock;
```

2.3 Funktionen

- Funktionsnamen werden lowerCamelCase geschrieben.
- Bei globalen Deklarationen wird der Bibliotheksname vorangestellt.
- Der Bibliotheksname wird groß geschrieben.

```
1     int8 Timer_initialize();
2     void Timer_setClock(uint32 clock);
```

2.4 Defines

- Defines werden in die Header-Datei geschrieben.
- Defines werden in Großbuchstaben mit _ geschrieben.
- Defines werden nach Möglichkeit vermieden und durch Konstanten ersetzt (Achtung funktioniert in C auf globaler Ebene nicht).

3 Regeln

3.1 Aufbau einer Header-Datei

1. Lizenzkommentar
2. Doxygen Library Header
3. #pragma once
4. Defines
5. Includes
6. Globale Enumeratoren und Konstanten
7. Globale Funktionsdeklarationen
8. Doxygen Library Footer

3.2 Aufbau einer Source-Datei

1. Lizenzkommentar
2. Include der Header-Datei
3. Private Enumeratoren, Konstanten und Variablen
4. Private Funktionsdeklarationen
5. Funktionsdefinitionen

3.3 Modularisierung

- Oft verwendeter Code soll gekapselt und in Funktionen verpackt werden
- Für Peripherals und Devices werden Bibliotheken erstellt
- Bibliotheken sollen keinen plattformspezifischen Code enthalten, dieser wird in eine Treiber-Datei ausgelagert
- Bibliotheken werden so geschrieben das sie möglichst vielseitig verwendbar sind, auch wenn nur eine Untermenge an Funktionen benötigt wird. Dies verhindert das Jemand der die Bibliothek nutzen will, sich nochmals in das Themengebiet einlesen muss.