Schneeweiss Codingstandard auf Windows und Mikrocontroller Systemen

Motivation

Diese Programmierrichtlinien beschreiben im Detail Layout, Namensgebung und Kommentierung von Softwarequellcode. Sie beziehen sich auf die Verwendung der Programmiersprachen C, C++ und C#. Sie sollten weiter eine Portierbarkeit des Quellcodes zwischen Plattformen und Systemen erleichtern.

Versionen

Version, Datum	Autor	Änderung
1.0 5.März 2007	Hülsebus	Erste Versionen dieses Dokuments
1.1 25.Juli 2007	Hülsebus	Überarbeitung der Empfehlungen

Inhalt

Namen und Bezeichner

Sprache

- Die Bezeichner sind in Deutsch zu wählen.
- Ausnahme: von den Bibliotheken vorgegebene Bezeichner.
- Ausnahme: im allgemeinen Programmierer Sprachgebrauch aus dem englischen Sprachraum stammende Bezeichner.
- Ausnahme: den Zusammenhang entstellende Bezeichner.
- Ausnahme: an anderer Stelle verwendete, aber dort nicht extra übersetzte Bezeichner.

Zeichensatz

- Multibyte (UTF) Zeichen sind unzulässig (MISRA, 8).

Groß und Kleinschreibung

- Funktionsnamen werden im ersten Buchstaben groß geschrieben: "FunktionsName" (Pascal Casing)
- Variablen werden im ersten Buchstaben klein geschrieben: "variablenName" (Camel Casing).
- Namen, die sich nur in Groß und Kleinschreibung unterscheiden, die aber eine vollkommen verschiedene Bedeutung haben, werden nicht verwendet.

Abkürzungen

- Abkürzungen werden in der Benennung von Bezeichnern vermieden, Beispiel: “GetDevice“ statt “GetDev“.
- Abkürzungen sind angemessen, um eine Modul oder Bibliothekszugehörigkeit anzudeuten, Beispiel: “uiGetDevice“ für User Interface.
- Für einfache, lokale Zähler können kurze Namen wie 'i', 'j' usw. verwendet werden.

Klassen, Module, Namespaces

- Werden keine Klassen verwendet, verdeutlicht die Wahl der Namen die Zugehörigkeit zu Modul und Datenstruktur. Funktionen und Variablen bekommen in Modulen ein Präfix im Namen, der an den Namen des Moduls angelehnt ist. Beispiel aus einer LCD Bibliothek:

```
LcdMenuClear ()
LcdMenuDraw ()
Menu *lcdMenuFileOpen
Menu *lcdMenuFileClose
```

- Wenn Klassen verwendet werden, sind Präfixe überflüssig, da diese Information aus der (Klassen)Hierarchie hervorgeht.

```
Lcd.Menu.Clear ()
```

```
Lcd.Menu.Draw ()
lcd.menu.FileOpen
lcd.menu.FileClose
```

- Namespaces werden wie folgt benannt: Firma.Bibliothek.Funktion

Instanzen

- Für Instanzen zusammengesetzter Datentypen, z.B. von Strukturen und Klassen, werden die Variablen und Funktionen nach dem Schema “[typ]+Funktion“ benannt. Dabei wird beachtet, “[typ]“ konsistent zu benennen.
- Die “[typ]“ Angabe wird erst benannt, dadurch ist die Zugehörigkeit der Instanzen sofort ersichtlich.
- Die “[typ]“ Angabe ist bei einfachen Datentypen kurz gehalten.
- Die Vorgabe von IDE’s genügt nicht und wird vermieden, da hier Unterschiede entstehen können. “[typ]“ ist nicht der Datentyp. Beispiel: in “menuLcdKontrast“ ist der Datentyp, abhängig von Sprache und Bibliothek, ein “struct Menu“, ein “System.Windows.Forms.Menu“ oder ein “System.Windows.Forms.MenuStrip“. Der Name “menuLcdKontrast“ wird nicht gewählt um auf einen Sonderfall in den Bibliotheken hinzuweisen, sondern um anzudeuten, dass ein Menüpunkt mit der Verwendung „Lcd Kontrast“ gemeint ist.

```
// Verwendung außerhalb des Moduls
Button  buttonEinschalten
Button  buttonAusschalten
Menu    lcdMenuKontrast

// Kontrastvariable LCD Modul
uchar   lcd_ucKontrast

// Zähler
uint32  lcd_ui32Count
```

- In C, C++ und C# wird dieselbe Konvention verwendet. C und C++ stellen Zeiger mit ’*’ Zeichen dar, hier wird man also schreiben:

```
// Verwendung außerhalb des Moduls
Button  *buttonEinschalten
Button  *buttonAusschalten
Menu    *menuLcdKontrast

// Kontrastvariable LCD Modul
uchar   lcd_ucKontrast

// Zähler
uint32  lcd_ui32Count
```

Callbacks

- Callbacks und Funktionstypen, also beispielsweise Delegaten in C# werden wie alle Funktionen groß geschrieben und nach ihrer Aufgabe benannt, nicht nach ihrem Kontext, konkret also: OnErrorEventCallback statt SomeHandler, Beispiel C und C++:

```
// Deklaration
void (* OnErrorEventCallback) (int errorNumer)
void InvokeError (int errorNumer)
{
  ...
}
OnErrorEventCallback = &InvokeError;

// Verwendung
(*InvokeError) (42)
```

- Beispiel C#:

```
// Deklaration
public delegate void ErrorEventHandler (int errorNumer);
public event ErrorEventHandler OnErrorHandler;
public void InvokeError (int errorNumer)
{
  ...
}

// Verwendung
InvokeError (42)
```

Namen

- Namen sind kürzer als 31 Zeichen um Kompatibilität zwischen Compilern zu gewärleisten. (MISRA, 11)
- Funktions-, Variablen- und Datenstrukturnamen sind selbsterklärend; Funktion oder Verwendungsweise gehen aus dem Namen hervor.
- Im Sachzusammenhang stehende Namen werden entsprechend gewählt:

```
// falsch:
#define MAX 42
for (int i = 0; i < MAX; i++)
{
    liste[i]
}

// richtig:
const int liste_Count = 42;
for (int i = 0; i < liste_Count; i++)
{
    liste[i]
}
```

- Gewählte Namen in einem inneren Scope überdecken auf keinen Fall Bezeichner in einem äußeren Scope. (MISRA, 21)

- Gewählte Namen überdecken keine Bezeichner die im System bereits vorhanden, beispielsweise durch “typedef“ deklariert sind. (MISRA, 17) , Beispiel:

```
// falsch:
typedef unsigned long ulong;
...
for (ulong = 0; ulong < 42; ulong++)
...
```

- Funktions- und Methodenparameter sind nach ihrer Funktion benannt, Beispiel:

```
// falsch
dist (o1, o2)

// richtig
BerechneDistanz (ortStart, ortZiel)
```

- Funktionen, die Werte zurückgeben, werden so benannt, dass der Name den Rückgabewert impliziert, Beispiel:

```
// gut
if (isString (““))
{
  ...
}

// falsch
if (NumberOrString (““))
{
  ...
}

// besser
if (isNumber (““) || isString (““))
{
  ...

// auch besser
if (NumberOrString (““) == NUMBER)
{
  ...
}
else if (NumberOrString (““) == STRING)
{
  ...
}
```

Formatierung

Zeilenumbruch

- Zeilen, die nicht auf eine Bildschirmbreite passen, werden sinnvoll umgebrochen.
- Insbesondere findet der Umbruch vor Operatoren und nach Kommata statt.
- Der Zeilenumbruch erhält die logische Struktur des Ausdrucks.

```
if  ((foo == 1)
  && (bar == 2)
  && (yuk == 3 * (c + 5)))
  ...
```

Leerzeilen

- Statements werden optisch durch eine Leerzeile sinnvoll gruppiert.
- Zwischen Funktionen und Datenstrukturen werden zwei Leerzeilen eingefügt.

```
void Foo ()
{
}


void Bar ()
{
}
```

- Zwischen Gruppen von Statements wird eine Leerzeile eingefügt; zwischen Statements mit klarem Sachzusammenhang keine.

```
uint32 i = 0;
while (i < 42)
{
    ...
}

uint32 n = 0;
while (n < 42)
{
    ...
}
```

Regionen

- Regionen unterteilen Quellcode in Funktionsgruppen
- In C# werden die “#region“ und “#endregion“ Direktiven verwendet.
- In C werden Kommentare verwendet.
- Ein aussagekräftiger Name und eine kurze Beschreibung der Funktionsgruppe ist an den Anfang einer Region gestellt

Leerzeichen, Tab

- Nach Kommata wird ein Leerzeichen gesetzt.

```
void LcdContrast (uchar display, ui16 value)
```

- Tab Breite wird auf vier Leerzeichen festgelegt. Tabs selbst werden nicht verwendet, die entsprechende Einstellung wird in der IDE vorgenommen. Das ist sinnvoll um zwischen verschiedenen Entwicklungsumgebungen wechseln zu können ohne den Quellcode vollkommen unleserlich werden zu lassen.

Operatoren

- Um Operatoren, ausgenommen '(', ')' und '[', ']', werden Leerzeichen gesetzt.

```
ui32count = ui32basiswert + (42 * ui32schwellwert);
```

Klassen und Strukturen

- In der Deklaration von Records und Klassen werden erst die publiken, globalen Members und danach die privaten, lokalen Members definiert.
- Dabei werden jeweils zuerst Funktionen, danach Variablen benannt.
- In C werden zwischen den Deklarationen entsprechende Kommentare eingefügt:

```
struct display
{
   // public
   void (* Clear) (void)
   void (*Contrast) (uchar display, ui16 value)
   uint16 width;
   uint16 height

   // private
...
}
```

Deklarationen

- Deklarationen, die einen Sachzusammenhang bilden, werden in “Tabellenform“ gebracht:

```
int32   basisWert;
uchar8  kuerzel;
```

Konstanten, Aufzählungen

- Im Quellcode verteilte, hart codierte konstante Zahlwerte (“magic Numbers“) werden nicht verwendet. Stattdessen werden der Präprozessor, also “#define“, mit “static const“ definierte Variablen oder Aufzählungen verwendet.
- Konstanten werden global definiert und beschrieben. In C und C++ wird pro Modul die Datei “const.h“ verwendet, in C# “const.cs“.

- Konstanten werden nach ihrer Funktion benannt.
- Für Konstanten werden Grossbuchstaben verwendet.

```
#define PI 3.141592
const float PI 3.141592
```

- Falls möglich, werden Aufzählungen verwendet:

```
enum Farbe
{
    ROT,
    GELB,
    BLAU;
}
```

- Oktale Konstanten werden nicht verwendet. (MISRA, 19).

Statements

- Jede Zeile enthält nur ein Statement.
- Gleiche Verschachtelungsebenen werden gleich weit eingerückt. Gleiche Konstrukte wie "`if`", "`while`", "`case`" usw. werden nach dem gleichen Schema eingerückt.
- Ausdrücke werden sinnvoll geklammert um die Lesbarkeit zu verbessern.
- Statements werden kurz und einfach gehalten.

Fallunterscheidungen

- Fallunterscheidungen werden sinnvoll eingerückt. Die Einrückweite beträgt 4 Leerzeichen.

```
if (a == b)
{
    foo ()
}

switch
{
    case ROT:
    {
    }
    case GELB:
    {
    }
    case BLAU:
    {
    }
}
```

Schleifen

- Schleifen werden sinnvoll eingerückt. Die Einrückweite beträgt 4 Leerzeichen.

```
while (a == b)
{
    foo ()
}
```

Return

- Der auf “return“ folgende Ausdruck braucht nicht geklammert zu werden.
- In “return“ Anweisungen werden keine komplexen Ausdrücke verwendet, da das Debuggen dadurch schwieriger wird, Beispiel:

```
int a = 42 * b;
return a;
```

Kommentare

Allgemein

- Kommentare im Quellcode beschreiben die Aufgabe einer Datenstruktur oder Funktion.
- Module, die die Datenstruktur oder Funktion verwenden (Abhängigkeiten), werden benannt. Das stellt die die Datenstruktur oder Funktion in einen größeren Kontext.
- Kommentare werden in Deutsch geschrieben. Kommentare aus verwendeten Bibliotheken bleiben in der originalen Sprache.
- Kommentare werden nicht geschachtelt. (MISRA, 9)
- Das Offensichtliche wird nicht kommentiert, Beispiel:

```
...
// result zurückgeben
return result;
```

- Kommentare sind ständig aktualisiert, da die Gründe für Diskrepanzen zwischen dem Kommentar und dem Quellcode seltenst auf gute Software deuten. Diskrepanzen zwischen Kommentar und Quellcode können bedeuten, dass der Quellcode falsch ist, dass der Kommentar falsch ist oder dass beide falsch sind. Konsistenz und Sorgfalt in den Kommentaren bedeutet Konsistenz mit der Projektbeschreibung und den Tests.

Datenstrukturen

- Für Datenstrukturen oder, im einfachsten Fall, globale Variablen, wird die Bedeutung der gespeicherten Daten beschrieben, Beispiel:

```
// Die Stärke des Luftstromes wird in [...] gemessen und in [...]
// verwendet, um den Verbrauch an [...] zu berechnen
uint32  iLuftStromWert;
```

Funktionen

- Die Aufgabe der Funktion wird beschrieben.
- Die Parameter, die Menge gültiger Argumente und ihre Bedeutung werden genannt.
- Die Bedeutung des Ausgabeparameters wird beschrieben.
- Um eine Verwendung von Quellcodes in automatischen Kommentierungs- Tools wie NDoc oder Intellisense zu ermöglichen, verwenden wir XML – Kommentare an den Funktionsköpfen Beispiel:

```
/// <summary>
/// Berechnen der Fakultät eines Wertes n durch Rekursion
/// Eingabe: integer Zahl n, Ausgabe: n!
/// sizeof (uint32) ist 4 Bytes
/// Eingabekontrolle findet nicht statt.
/// </summary>
/// <param name="n">integer Zahl n,
/// maximaler Eingabewert: 16</param>
/// <returns>n!, maximaler Ausgabewert: 2004189184</returns>
```

- Innerhalb von Funktionen erklären normale (also nicht XML konforme) Kommentare Zusammenhang und Hintergrund der Konstrukte.
- Spezielle Seiteneffekte oder Abhängigkeiten werden durch Kommentare benannt und beschrieben.

```
int Fak (uint32 n)
{
    if (n == 0)
    {
        // 0! = 1, also Abbruch der Rekursion
        return (1);
    }
    else
    {
        // n! = n * (n - 1)! für n > 0
        return (n * Fak (n - 1));
    }
}
```

- Besondere, also unnötig schwer verständliche Software Tricks, ungewöhnliche Softwarekonstrukte sowie exotische oder nicht portable Funktionen, werden soweit möglich, vermieden. Ist das nicht möglich, werden sie durch Kommentare benannt und beschrieben.
- Design Patterns werden durch Kommentare benannt und ggf. auch beschrieben um das Verständnis der Programm Logik zu erleichtern.
- Bei Algorithmen, die in engem Zusammenhang mit dem Projekt stehen, werden diese Vorgaben benannt, Beispiel:

```
// Berechnung der Strömung nach der Formel ...
float BerechneStroemung (uint32 querschnitt, float druck)
{
     ...
}
```

- Bei Standardalgorithmen, die nicht offensichtlich verständlich sind, kann ein Hinweis auf weitere Informationsquellen, beispielsweise im Internet, aufgenommen werden.

```
// Bresenham algorithm, see
// www.cs.helsinki.fi/group/goa/mallinnus/lines/bresenh.html
// draw a line from point x1, y1 to point x2, y2
// c == 0 draws no pixel: "white"; c != 0 draws a pixel: "black"

void lcd_drawline (uint16 x1, uint16 y1, uint16 x2, uint16 y2,
uchar c)
{
  ...
}
```

Präprozessor

- Präprozessor Anweisungen werden auf jeden Fall kommentiert um die Verschachtelung anzudeuten.

```
// Version mit Reifen
#define LASTFAHRZEUG 1
  ...
  ...

#ifdef LASTFAHRZEUG
  ...
#else // LASTFAHRZEUG
  ...
#endif // LASTFAHRZEUG
```

Empfehlungen

Präprozessor

- Präprozessor Anweisungen vom Typ “#define“ sind zugunsten von als “const ...“ definierten Variablen zu vermeiden, da sie keine Typprüfung erlauben. Werden sie dennoch verwendet, erhalten sie eine Begründung, warum eine syntaktische Textersetzung an dieser Stelle sinnvoll ist.
- Ausnahme: spezielle Speicheradressen, in der Regel also Funktionsregister, werden mit “#define“ in eine lesbare, einheitlich benannte Form gebracht.

```
// Output LED 1
#define OUT_LED1_WRITEMODE  P1OUT
#define OUT_LED1_SELECT     P1SEL
#define OUT_LED1_DIRECTION  P1DIR
#define OUT_LED1_OUTPUT     BIT7
```

- Präprozessor Anweisungen vom Typ “#ifdef“ sind zugunsten von der Verwendung geeigneter Module (z.B. Treiber, Ressourcen) zu vermeiden, da sie semantische

Abhängigkeiten durch syntaktische Konstrukte ersetzen. Werden sie dennoch verwendet, erklärt ein Kommentar, warum.

- Funktionsmakros werden nicht verwendet, sie haben keinerlei Vorteile und können zu schwer findbaren Fehlern leiten, Beispiel:

```
// Funktionsmakro
#define isupper (c) ((c) >= 'A' && (c) <= 'Z')

// Falsch für c >= 'A'
while (isupper (c = getchar ())) ...
```

Register

- Beim Setzen werden im Normalfall immer alle Bits des Registers gesetzt, und zwar vom hochwertigsten Bit (MSB) herunter zum niederwertigsten (LSB). Also „MSB first“.
- Ausnahme sind solche Register, in denen mehrere voneinander unabhängige Komponenten des Controllers gemeinsam konfiguriert werden, z.B. die Timer. Hier werden die zu verändernden Bits entweder einzeln angesprochen:

```
REGISTER |= (1 << BIT) | (0 << BIT)
```

oder das Register passend maskiert:

```
REGISTER = (REGISTER & MASKE) | (1 << BIT)
```

In jedem Fall wird dieser Umstand mit einem Kommentar erwähnt.

- Maskierungen auf Registern werden über einen nachvollziehbaren binären Wert gemacht, nicht hexadezimal.
- Es werden keine Werte, weder in Hexadezimaler noch in Binärer Form, direkt für Zuweisungen an Status- und Steuerregister verwendet, also kein REGISTER = 0xA3 und REGISTER = 0b00011101.
- Auch wenn der Default-Wert eines oder mehrerer Bits zufällig dem gewünschten Wert entsprechen sollte, wird es / werden sie trotzdem immer gesetzt.
- In Fällen wo sich Einstellungen des Controllers über Bits in mehreren Registern erstrecken (z.B. Control-Register A Bits 4-2 und Control-Register B Bits 6 und 5 stellen XY ein) wird dies per Kommentar dokumentiert.
- Wenn eine Komponente (z.B. PWM - Ausgang) über drei Steuerregister verfügt, für die gewünschte Funktionalität aber nur die ersten beiden Änderungen erfordern, werden trotzdem alle drei aufgeführt, ggf. auskommentiert, aber der Name wird genannt.
- Enthält ein Register Bits ohne Funktion, die aber für die zukünftige Kompatibilität auf einen definierten Wert gesetzt sein müssen, so wird dies gemacht, kommentiert und das Bit über seine Nummer angesprochen.
- Bits werden immer über ihren Konstanten-Namen angesprochen, also (1 << SCLK0) und nicht (1 << 5). Ausnahme sind die oben erwähnten Bits ohne Funktion für „future compatibility“.
- Register für Datenrichtung (DDR), zur Umschaltung zwischen Ein- und Ausgang (PORT) sowie Datenregister sind mit binären Konstanten zu manipulieren, nie mit hexadezimalen. Ausnahme können 0x00 und 0xFF sein, z.B. um alle Pins eines Ports als Ausgang bzw. Eingang zu schalten, etc.
- Maßstab ist, dass alle Operationen nachvollziehbar sein müssen. Sofern das ohne genaue Kenntnisse der Zusammenhänge nicht möglich ist, hilft ein Kommentar weiter, ggf. mit Verweis auf die Seite(n) im Handbuch.

Daten und Variablen

- Pro Zeile wird eine Variable deklariert.
- Initialisierungen und ggf. Deklarationen aller Funktionen und Variablen werden kurz vor und im Zusammenhang mit der Verwendung der Variablen durchgeführt. (MISRA, 20)
- Variablen werden im kleinstmöglichen Scope verwendet.
- Systemeigene Datentypen werden nicht direkt verwendet, da ihr Speicherbedarf von System zu System unterschiedlich sein kann. Stattdessen werden Datentypen wie "uint32", "uchar8" usw. definiert, aus deren Namen Typ und Größe klar hervorgehen. (MISRA, 13). Die Größe eines systemeigenen Datentypen wird nicht vermutet sondern mit "sizeof" festgestellt.
- Insbesondere ist der Datentyp “char“ konsistent definiert als entweder “signed char“ oder “unsigned char“ (MISRA, 14)

Code

- Software wird so in Funktionen und Methoden zerlegt, dass jede eine klare, einfache Aufgabe erledigt, dadurch werden Spezifikationen und API’s klarer und Tests einfacher.
- Für die Mikrocontroller Programmierung empfehlen wir eine Dreischichtarchitektur. In der oberen Schicht liegen die Anwendungsprogramme oder Module und die von ihnen verwendeten Bibliotheken, die die Funktionalität der Software implementieren. Diese sind hardwareunabhängig gehalten und können daher einzeln auch auf anderen Systemen oder im Simulator getestet werden. Die mittlere Schicht bildet eine Datenaustauschschicht, hier liegen global alle relevanten Datenstrukturen. Die dritte, unterste Schicht wird aus hardwarenahen Treibern aufgebaut. Bei einem Wechsel der Hardware muss diese Schicht als Einzige umprogrammiert werden.
- Für die Programmierung von Client Server Anwendungen empfehlen wir den Einsatz üblicher Design Patterns.
- Die Länge einer Funktion und Methode sollte eine Bildschirmseite nicht wesentlich überschreiten.
- Für C Programme wird der ISO 9899 C Standard eingehalten. (MISRA, 1) Dieser Standard beschreibt, Zitat: “This International Standard specifies the form and establishes the interpretation of programs written in the C programming language“. Bei C Compilern kommen eigenmächtige und daher inkompatible Sprachumdeutungen und Erweiterungen von Compilerherstellern so inkonsequent vor, dass man sich auf so was gar nicht erst verlassen sollte. Der Standard hingegen sollte und wird auch meist gut implementiert.
- Für C# Programme existiert die entsprechende Standard ECMA-334: C# Language Specification (ISO/IEC 23270:2006), Zitat: “This International Standard specifies the form and establishes the interpretation of programs written in the C# programming language“.
- Softwarekonstrukte sollen einfach lesbar sein. Beispiel:

```
// falsch
while (*(str + i++)); i--;

// richtig
i = strlen (str);
```

- Die Qualität des Codes und die Einhaltung der Codingstandards werden mit einem geeigneten Tool geprüft.

Fehlerbehandlung

- Funktionen führen eine Eingabe und Ausgabekontrolle durch.
- Rückgabewerte von Bibliotheksfunktionen werden immer überprüft.
- Stellt das Betriebssystem einen systemgenerierten Fehlertext zur Verfügung, so wird der in die Fehlermeldung mit aufgenommen.
- Das Auftreten eines Fehlers wird an den Benutzer rückgemeldet.
- Die Fehlerbehandlung von Softwareinternen Fehlern stellt mindestens die fehlergenerierende Funktion und deren Parameter zur Verfügung, falls möglich einen Callstack.
- Die Fehlerbehandlung von Benutzerführungsfehlern erzeugt im Gegensatz zu gängiger Praxis keine Statusmeldung, sondern gibt konstruktive Hinweise wie das Problem lösbar ist.
- Ist eine ausführliche Rückmeldung nicht möglich, so wird versucht, wenigstens einfachste Statusmeldungen zu erzeugen, wie Beispielsweise Statuscodes im LCD oder farbcodierte Leuchtdioden.

Semantik

- Die rechte Seite von Operatoren darf keine Seiteneffekte enthalten. (MISRA, 33) Beispiel: “while (i < j++) ...“ ist unzulässig.
- In logischen Operatoren werden nur primäre Ausdrücke verwendet. (MISRA, 34) Beispiel: “while (i && j++) ...“ ist unzulässig.
- Insbesondere werden in Ausdrücken, die “true“ oder “false“ zurückgeben, keine Anweisungen verwendet. (MISRA, 35) Beispiel: “while (i < j = k) ...“ ist unzulässig, Grund: die korrekte Funktion dieses Codefragmentes basiert auf Annahmen über den Compiler, die stimmen können, aber nicht müssen, und die aus diesem Codefragment nicht ersichtlich sind.
- Implizites Konvertieren ist unzulässig. (MISRA, 43). Casten von Typen auf void Zeiger ist unzulässig. (MISRA, 45)
- Labels werden nicht verwendet. (MISRA, 55) Ausnahme: innerhalb eines “switch“.
- “goto“ Konstrukte werden nicht verwendet. (MISRA, 56)
- Identische Funktionszeiger zeigen immer auf Funktionen mit identischen Prototypen. (MISRA, 105)
- Zeiger auf dynamischen Speicher haben denselben Scope wie der Speicher selbst. (MISRA, 106)
- Variablen, die in “for“ Ausdrücken als Schleifenzähler verwendet werden, werden in der Schleife nicht verändert. (MISRA, 67)
- Verwendung von “++“ usw. Operatoren wird nicht empfohlen, Beispiel: array[i++] = i; ist nicht eindeutig und möglicherweise auch nicht zwischen Compilern und Architekturen portabel.