IfA - Software-Richtlinien

QUALITÄTSSICHERNDE PROGRAMMIER-REGELN FÜR C

Erstellt von: Dr.-Ing. S. Dyblenko, Dipl.-Ing. S. Reimann

Datum: 24.01.2006

Inhaltsverzeichnis

1 VORWORT

Diese Programmierregeln sind konstruktiv qualitätssichernde Maßnahmen 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 aus der Zusammenarbeit in Softwareprojekten mit Industriepartnern entstanden.

Diese Programmierregeln für die Programmiersprache C sind bei allen entsprechenden Softwareprojekten am Institut für Automatisierungstechnik zu verwenden. Abweichungen sind nur in Ausnahmefällen erlaubt.

Die allgemeinen (syntaktischen) Programmierregeln von C sind der entsprechenden Literatur zu entnehmen.

2 PROGRAMMSTRUKTUR

Das gesamte Programm wird zweckmäßig in verschiedenen C-Modulen implementiert.

Ein C-Modul besteht aus einer Implementierungsdatei und einer Headerdatei.

Zu jeder Implementierungsdatei (Erweiterung ".c") gehört eine Headerdatei mit demselben Namen und der Erweiterung ".h".

Der Modulname ist ein Substantiv. Der Modulname soll eine semantische Bedeutung tragen und soll möglichst kurz sein. Als Zeichen sind die Kleinbuchstaben `a...z`, Unterstrichzeichen "_" sowie die Ziffern `0...9` erlaubt.

Jedem Modul wird ein Kürzel aus drei Zeichen zugeordnet. Das Namenskürzel setzt sich nach Möglichkeit aus den ersten drei Zeichen des Dateinamens zusammen, soll in jedem Fall in Großbuchstaben geschrieben werden (kein Unterstrichzeichen "_", Ziffern 0..9 sind jedoch erlaubt) und soll explizit in der Dateiheader gezeigt werden. Das Kürzel wird bei der Namensgebung von Makros und zum Teil bei Funktionen und Variablen verwendet. Das Namenskürzel wird dabei mit einem Unterstrich von der Bezeichnung von Makros, Funktionen und Variablen getrennt.

Beispiel:

Modul - Steuerung der Akkuaufladung

Implementierungsdatei:	`charge_control.c`
Headerdatei:	`charge_control.h`
Modulkürzel:	`CHA`

Der Name des Moduls mit der Funktion `main()` muss unbedingt den Text `main_module` enthalten.

Die Headerdatei beschreibt nur die Schnittstellen zum C-Modul. Modulinterne Definitionen und Deklarationen werden in der Implementierungsdatei beschrieben.

In der Headerdatei sind Variablendefinitionen oder Codeteile verboten.

Um mehrfaches Einfügen einer Headerdatei zu vermeiden, ist der Hauptteil der Headerdatei in einen `#ifndef` Konstrukt einzubinden. Der Name des Compiler-Schalters (Makros) baut sich wie folgt auf: < Unterstrich ><Modulname>< Unterstrich><H>< Unterstrich >. Modulname ist dabei in Großbuchstaben zu schreiben. Beispiel: `_CHARGE_CONTROL_H_`

Es wird Datenkapselung betrieben. D.h., es darf nur über dafür vorgesehene Schnittstellenfunktionen oder Schnittstellenmakros auf Daten (Variablen, Makrokonstanten) anderer Module zugegriffen werden.

3 DATEIAUFBAU

Zum einheitlichen Layout sind die Vorlagen aus Anhang A1 zu verwenden.

4 MAKROS

Die Namen von Makros werden grundsätzlich groß geschrieben. Da durch die Großschreibung Wörter nicht mehr eindeutig getrennt sind, wird ein Unterstrichzeichen "_" eingefügt.

Die Namen von Makros müssen prägnant sein. Kürzere englische Namen sind bevorzugt.

Die Namen von sämtlichen Makros beginnen mit Modulkürzel und Unterstrichzeichen.

Werden Makros im Sinne von Funktionen verwendet, werden sie genauso wie Funktionen benannt. Sie erhalten dann grundsätzlich (auch wenn sie nicht parametrisiert sind) Klammern.

Beispiel:

Modul - Steuerung der Akkuaufladung, Modulkürzel: `CHA`.

Makro als eine Konstante:	`CHA_VOLTAGE_U0`
Makro als eine Funktion ohne Parameter:	`CHA_GET_VOLTAGE()`
Makro als eine Funktion mit Parameter:	`CHA_SET_VOLTAGE(x)`

Die Verwendung von Makros in Makros ist grundsätzlich zu vermeiden.

Makros mit mehr als einer einfachen Zuweisung sind nicht erlaubt. Stattdessen sind Inline-Funktionen zu verwenden.

Makros dürfen als "Funktionsersatz" nur dann verwendet werden, wenn es aus Code Optimierungsgründen unumgänglich ist.

Makros mit einer Anweisung müssen in Klammern eingeschlossen werden. Makros mit Parametern müssen die Parameter in der Ersetzung klammern, sowie zusätzlich den gesamten Ersetzungsausdruck.

Beispiel:

```
#define   CHA_CHECK_VOLTAGE_LEVEL(u,l) ((u) > (l))
```

5 VARIABLEN

Variablen bestehen lediglich aus einem oder mehreren Substantiven und Eigenschaftswörtern. Es darf Groß- und Kleinschreibung verwendet werden. Großbuchstaben kennzeichnen den Beginn eines neuen Wortes. Nutzung von nur Großbuchstaben ist wegen der Verwechslung mit Makros nicht erlaubt.

Die Namen von Variablen müssen prägnant sein. Kürzere englische Namen sind bevorzugt.

Die Namen von sämtlichen modulglobalen Variablen beginnen mit Modulkürzel und Unterstrichzeichen. Dies gilt auch für die Schnittstellenvariablen, obgleich sie nur von Schnittstellenfunktionen benutzt werden.

Zur Kennzeichnung des Datentyps wird ein Datentypbezeichner der Variablen vorangestellt. Mögliche Kennzeichner sind der Tabelle 4.1 zu entnehmen.

Variablen von positiven ganzzahligen Datentypen (z. B. `unsigned int`, `unsigned char`) mit nur einem Buchstabe im Namen (z.B. `i, j, k`) dürfen ohne Datentypbezeichner verwendet werden.

Tabelle 4.1: Datentypbezeichner und Datentypen

Datentyp	Datentypbezeichner
`unsigned char`	`uc`
`signed char`	`sc`
`char`	`c`
`int`	`i`
`unsigned int`	`ui`
`unsigned long`	`ul`
`long`	`l`
`unsigned char *`	`puc`
`signed char *`	`psc`
`char *`	`pc`
`unsigned int *`	`pui`
`int *`	`pi`
`unsigned long *`	`pul`
`long *`	`pl`
`float`	`f`
`double`	`d`
`long double`	`ld`
`float *`	`pf`
`double *`	`pd`
`long double *`	`pld`
`struct *`	`ps`
`function-ptr`	`pfn`
`void`	`v`
`void *`	`pv`

Beispiel:

Modul - Steuerung der Akkuaufladung, Modulkürzel: `CHA`.

Modulglobale Variable: `CHA_uiVoltageLevel`
Modullokale Variable: `uiIndex`

Ob der Typbezeichner „`char`“ mit oder ohne Vorzeichen definiert ist, hängt häufig von Compiler ab und ist somit nicht eindeutig. Deshalb darf der Typbezeichner „char“ nur für alphanumerische Zeichen verwendet werden. Für ganze Zahlen im 8-Bit-Format dürfen nur „`signed char`“ oder „`unsigned char`“ verwendet werden.

Der Pointer-Qualifizierer (`*`) muss beim Namen und nicht bei Datentypen stehen.

Variablen vom gleichen Typ müssen einzeln deklariert und kommentiert werden:

kein
```
int i, uiBtIndex;
```
sondern
```
int i;                      // Kommentar
int uiBtIndex;              // Kommentar
```

Alle Variablen müssen kommentiert werden (siehe Kommentierung).

Werden die Modulschnittstellen als Funktionen ausgeführt und nicht als Makros, müssen alle globalen Variablen im Modul mit `static` gekennzeichnet werden. Dadurch erhält man eine erhöhte Sicherheit bei der Datenkapselung.

6 STRUKTUREN

Strukturen werden grundsätzlich über `typedef` definiert. Der Strukturname wird identisch zu Variablen benannt. Als Kennzeichnung wird "`Struct`" vorausgeschrieben.

Beispiel:

Deklaration eines Strukturtypen:

```
typedef struct {
        float fVol;    /*[V], Ladespannung*/
        float fCur;    /*[A], Ladestrom   */
        } typeStructChargeParams;
```

Definition einer Struktur (lokale Variable):

```
typeStructChargeParams StructChargeParams;
```

Definition einer Struktur (globale Variable):

```
typeStructChargeParams CHA_StructChargeParams0;
```

Alle Strukturfelder müssen kommentiert werden (siehe Kommentierung).

7 KONSTANTEN

Die Verwendung von Zahlen als Konstanten ist grundsätzlich zu vermeiden. Stattdessen sind Makros zu verwenden.

Die Verwendung von Konstanten mit Typqualifizierer `const` ist zu vermeiden. Alle Konstanten sind grundsätzlich über Makros (`#define`) zu implementieren.

8 FUNKTIONEN

Funktionen beschreiben eine Aktivität. Die Namen von Funktionen bestehen deshalb aus einem Verb und einem Substantiv.

Schnittstellenfunktionen erhalten zudem den 3-Zeichen-Kürzel des Modulnamens (+Unterstrich).

Es darf Groß- und Kleinschreibung verwendet werden. Großbuchstaben kennzeichnen den Beginn eines neuen Wortes. Nutzung von nur Großbuchstaben ist wegen Verwechslung mit Makros nicht erlaubt.

Dem Funktionsnamen vorangestellt wird eine Abkürzung für den zurückgelieferten Datentyp – siehe Tabelle 4.1.

Funktionen die keinen Rückgabewert haben müssen mit "`void`" deklariert werden.

Es sind Funktionsprototypen zu verwenden (in Headerdatei). Schnittstellenfunktionen erhalten zudem den Typqualifizierer "`extern`" in ihrer Deklaration.

Tabelle 8.1: Einige bevorzugte englische Verben für Aktionsbezeichnung (anstelle von „Start“ im unteren Beispiel)

Verb	Mögliche Bedeutung der Aktion
`Adjust`	Abgleichen eines Wertes
`Apply`	Zuweisen eines neuen Wertes
`Calc`	Einen Algorithmus zur Berechnung von etwas ausführen
`Check`	Überprüfen
`Clear`	Zurücksetzen
`Cntrl`	Eine Kontrollaktivität durchführen
`Cvrt`	Konvertieren
`Decr`	Dekrementieren
`Get`	Daten abfragen aus einem anderen Modul
`Read`	Daten abfragen aus Hardware
`Init`	Initialisieren
`Lookup`	Wert, Datum über eine Tabelle erfragen
`Manage`	Andere Aktivitäten anstoßen
`Reset`	Zurücksetzen Daten oder Zustände
`Set`	Setzen von Daten in einem anderen Modul
`Start`	Start eines Prozesses
`Stop`	Stop eines Prozesses
`Test`	Durchführen eines Tests
`Update`	Aktualisierung von Daten
`Is`	Abfrage eines logischen Zustandes

Beispiel:

Modul - Steuerung der Akkuaufladung, Modulkürzel: `CHA`.

Modulinterne Funktion: `double dGetVoltage()`
Schnittstellefunktion: `void CHA_vStartCharging()`

9 KODIERUNG

Um ein einheitliches Layout von Quelldateien zu erreichen, werden folgende Kodierregeln vorgeschrieben. Ein einheitliches Layout ermöglicht eine leichte Einarbeitung, verhindert Fehler und erleichtert die Wiederverwendung von Code.

9.1 Codeblöcke

Zusammengehörende Textteile nach Konstrukten wie `for`, `if`, `while` usw. werden üblicherweise 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` usw.

Der Blocktext hat einen linken Einzug von mindestens drei Leerzeichen von den Blockklammern.

Die Verwendung von Tabulatoren zum Einzug ist grundsätzlich zu vermeiden. Stattdessen sind Leerzeichen zu verwenden.

Beispiel:

```
if (fCurrent == 0)
{
   /* Berechnung von ...*/
   iOut[] = iIn[i+k];

   /* Berechnung von ...*/
   for(i=0;i<7;i++)
   {
      iIn[i] = 0;
      iOut[] = iIn[i+k];
   }
}
```

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

Beispiel:

```
/* Zurücksetzen aller Werte in einem Feld*/
for(i=0;i<7;i++) iIn[i] = 0;
```

9.2 Logische Operationen und Operanden

Variablen, Parametern und Rückgabewerten von Funktionen, die einen logischen Zustand ausdrücken, müssen vom Datentyp `unsigned char` sein. Solche Variablen und Funktionen bekommen den Datentypbezeichner `b`.

Als logisch "falsch" ist 0 definiert. Als logisch "wahr" gilt jeder Wert ungleich Null.
Hintergrund: Die Sprache C erlaubt, die Benutzung von Variablen und Rückgabewerten von Funktionen direkt, d.h. ohne Vergleich, in logischen Abfragen und Verknüpfungen einzusetzen. C interpretiert den Wert 0 als "falsch" und einen Wert ungleich 0 als "wahr".

Beispiel:

```
unsigned char bIsPortOpen;
```

9.3 Ausdrücke

Es ist sicherer, Vergleiche mit einer Makrokonstante umzudrehen:
`if (CHA_KONSTANTE == dVoltage).`
Ein Verwechseln von “==” mit “=” wird dadurch vom Compiler sicher erkannt.

Eine Verschachtelung von Zuweisungen ist zu vermeiden: `d = ( a=b+c ) + r;`

`goto` Anweisungen sind nicht erlaubt.

“`break`” in Schleifen ist erlaubt, wenn sich daraus eine Vereinfachung beim Lesen des Codes ergibt.

Es darf mehr als ein “`return`” in Funktionen geben, wenn dadurch das Fehlerhandling der Funktion übersichtlicher wird.

Bedingungen müssen lesbar sein. Doppelte Verneinung ist zu vermieden. Wird nur ein Zweig einer `if` Bedingung genutzt muss dies der „then“ Zweig sein.

Es darf sich nicht auf automatische Typenkonvertierung verlassen werden. Es ist Gebrauch vom casting-Operator zu machen.

Zur Verbesserung der Lesbarkeit sind Leerzeilen einzufügen.

9.4 Operatoren

Runde Klammern müssen zu einer besseren Lesbarkeit verwendet werden. So können auch Fehler vermieden werden, die entstehen können, wenn die Priorität von Operatoren falsch berücksichtigt wurde.

Bei komplizierten Ausdrücken müssen die Operatoren und Operanden für eine bessere Lesbarkeit durch Leerzeichen separiert werden.

Der Komma Operator (`,`) darf nicht zur Anweisungsschachtelung verwendet werden. Ausnahme: `for`-Konstrukt.

Beispiel:

```
/* Berechnung von ...*/
for(i=0, j=0;i<7;i++,j++) ia[i] = ib[j];
```

9.5 Hardwarezugriffe

Hardwarezugriffe sind in Funktionen, Makros, eigene Module zu kapseln.

9.6 Kommentierung

Allgemein gilt: die Quelltexte sind zu kommentieren. C ist jedoch 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.

Kommentare müssen mit der Codierung übereinstimmen und gepflegt werden.

Am Beginn jeder Datei ist ein Kommentar-Dateiheader nach Doxygen-Richtlinien einzufügen.

Alle Modulschnittstellen sind nach Doxygen-Richtlinien zu kommentieren:
- Kommentare bei Prototypen sämtlicher Funktionen und „Makros als Funktionen“ mit Angabe der abgedeckten Funktionalität und Beschreibung der Übergabe- bzw. Rückgabeparameter
- Kommentare bei Deklaration sämtlicher globalen Variablen und bei „Makros als Konstanten“

Herkömmliche C-Kommentare sind erforderlich vor allem bei:
- Deklarationen von Variablen (wenn es keinen Doxygen-Kommentar gibt)
- Abfragen von Daten
- Schleifen, Verzweigungen
- Aufwendigen Berechnungen

C-Kommentare, die sich auf eine Zeile beziehen, sind entweder über diese zu schreiben (Ausnahme - Variablen) oder am Ende der Zeile.

C-Kommentar mit nur einer Zeile bzw. nach Code am Ende einer Zeile darf mit “`//`” gekennzeichnet werden, sonst muss “`/* */`” verwendet werden.

Der C-Kommentar einer Variable muss entweder direkt nach der Variablen in der gleichen Zeile stehen, oder er muss als Blockkommentar `/* */` in der nächsten Zeile beginnen.

Wenn eine Variable eine physikalische Größe darstellt, ist deren Messeinheit unbedingt anzugeben.

Deklarationen von Feldern einer Modulschnittstellen-Struktur werden einzeln und gemäß Doxygen-Richtlinien kommentiert. Deklarationen von Feldern von modulinternen Strukturen werden als

modulinterne Variablen kommentiert.

Kommentare dürfen nicht geschachtelt werden.

Alter Code, der nicht gelöscht wird, darf nicht durch Compiler Schalter deaktiviert werden. Er ist grundsätzlich mittels Kommentaranweisungen zu deaktivieren.

9.7 Bedingte Übersetzung

Compilerschalter-Makros dürfen nur zur Verhinderung von mehrfachem Einfügen von Headerdateien, zur Variantenbildung und für das Debugging verwendet werden.

Compilerschalter-Makros werden in globale und lokale Schalter unterteilt.

Die globalen Schalter gelten für alle Module und werden in einer speziellen Headerdatei allen Modulen zur Verfügung gestellt. Die globalen Schalter bekommen Präfix `GSW_` (global switch).

Die lokalen Compilerschalter gelten nur für ein Modul und werden nur dort definiert. Die lokalen Compilerschalter bekommen Präfix: `LSW_` (local switch). Das Modulkürzel ist auch zu verwenden.

10 ANHANG A1

Als Beispiel für Modul Steuerung der Akkuaufladung.

10.1 Struktur einer Headerdatei

Datei: charge_control.h

```
************************************
Dateiheader nach Doxygen-Richtlinien
************************************

#ifndef _CHARGE_CONROL_H_
#define _CHARGE_CONROL_H_

//=====================================================================
// includes
#include


//=====================================================================
// data types


//=====================================================================
// interface macros and variables


// =====================================================================
// module functions

#endif // _CHARGE_CONROL_H_

/*********************************************************************/
```

10.2 Struktur einer Implementierungsdatei

Datei: charge_control.c

```
************************************
Dateiheader nach Doxygen-Richtlinien
************************************

// =====================================================================
// includes

#include "charge_contol.h"


// =====================================================================
// external module variables


// =====================================================================
// internal module variables


// =====================================================================
// module functions



// = eof: charge_control.c
/*********************************************************************/
```