

A decorative graphic on the left side of the slide, consisting of a network of blue dots connected by thin lines, forming a complex web-like structure.

Creating safety.
With passion.

NewTec

Software-Engineering

- Projekt-Kodierstandard

**Die folgenden Regeln sind im Rahmen
der SW-Einführungsschulung einzuhalten!**

1. Code Style

1.1 Englische Sprache


- Quellcode und Kommentierung *müssen* in Englisch gehalten werden.

1.2 Namensgebung

- Namen für Bezeichner (Module, Includes, Konstanten, Makros, Typen Variablen und Funktionen) *sollen* aussagekräftig und damit leicht zu merken sein.
- Bei der Benennung *sollte* die natürlichen Sprache Vorbild sein.
- Schleifenvariablen *dürfen nicht* nur mit „i“, „j“, „ii“ oder ähnlichen Kurzbezeichnungen benannt werden.

- Module:

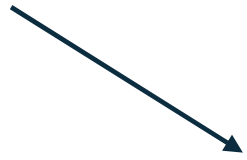
- Selbsterklärend & spezifisch, erster Buchstabe groß, Rest klein.
- Ganze Worte, Zusammensetzung durch Großschreibung („CamelCase“).



```
LineSensor.c  
LineSensor.h
```

- Includes:

- Bei der Reihenfolge der Includes werden zuerst Standardbibliotheken aufgeführt, gefolgt von einer Leerzeile.
- Danach die Header-Datei des eigenen Moduls.
- Dann alle weiteren Header-Dateien.



```
#include <avr/io.h>  
  
#include "LineSensor.h"  
#include "Gpio.h"
```



- Konstanten :


- Nur Großbuchstaben. Zur Abgrenzung von Wörtern werden Unterstriche verwendet.
- Standardpräfixe: MIN_, MAX_, DEFAULT_, ...



```
#define DEFAULT_DURATION (200)
```


- Makros:

- Nur Großbuchstaben.
- Zusammengesetzte Worte werden mit „_“ getrennt.
- Makros sollen immer geklammert werden.
- Makros sollen immer kommentiert werden.




```
/** Determines the minimum of two comparative values */  
#define MIN(a,b) ((a)<(b)?(a):(b))
```

- Typen :
 - Selbsterklärend & spezifisch, erster Buchstabe groß, Rest klein.
 - Ganze Worte, Zusammensetzung durch Großschreibung („CamelCase“).



```
typedef enum
{
    GPIO_RET_OK = 0,    /**< Ok. */
    GPIO_RET_ERROR    /**< Error. */
} GpioRet;
```

- Aufzählungstypen: nur Großbuchstaben, Wort Trennung mit „_“.
- Strukturkomponenten beginnen mit einem Kleinbuchstaben.



```
typedef struct
{
    UInt8 options;    /**< Options. */
    UInt8 index      /**< Index. */
} DemoStruct;
```


- Variablen :
 - Selbsterklärend & spezifisch, beginnen mit einem Kleinbuchstaben.
 - Ganze Worte, Zusammensetzung durch Großschreibung („camelCase“).
 - Bildet die Variable eine physikalische Einheit ab, soll die Variable mit der Einheit abschließen.
 - Modulvariablen beginnen mit einem kleinen „g“ mit lokaler Gültigkeit für das Modul → Deklaration als „static“.

UInt32 sensorValues;

UInt32 gTickCountMs;

- Funktionen :

- Selbsterklärend & spezifisch, beginnen mit einem Kleinbuchstaben.
- Ganze Worte, Zusammensetzung durch Großschreibung („CamelCase“).
- Externe Funktionen beginnen mit dem Modulnamen gefolgt von einem “_”.



```
void initDemoStruct(void)
```



```
void Pwm_setDutyCycle(PwmID id, UInt8 percent)
```

1.3 Einrückung

- Funktionelle Blöcke *müssen* eingerückt werden.
- Jede Einrückungsebene *muss* 4 Leerzeichen tief sein.
- Tabs *dürfen nicht* verwendet werden.

1.4 Quellcode-Struktur

- Zu jeder .c Datei *muss* es eine zugehörige .h Datei geben.
- Innerhalb einer .c Datei *muss* folgende Reihenfolge eingehalten sein:
 - Includes
 - Konstanten
 - Makros
 - Typen
 - Prototypen
 - Externe Funktionen
 - Lokale Funktionen

1.5 Kontrollstrukturen

- `if / else if / else`
 - Funktionelle Blöcke nach `if`-Anweisungen *müssen* geklammert sein, auch wenn die bedingte Anweisung einzeilig ist.
 - Nach der Anweisung `if` bzw. `else if` *soll* immer ein Leerzeichen folgen, zur besseren Lesbarkeit.

```
Gut:  
if (Bedingung)  
{  
    methode();  
}
```

```
Schlecht:  
if (Bedingung)  
    methode();
```

```
if (Bedingung)  
{  
    methode();  
}  
else if (Bedingung)  
{  
    ...  
}  
else  
{  
    ...  
}
```

- `for` / `while` / `do while`
 - Nach der Anweisung *soll* immer ein Leerzeichen folgen.

```
for (index = 0; index < 10; ++index)
{
    ...
}
```

```
while (Bedingung)
{
    ...
}
```

```
do
{
    ...
} while (Bedingung);
```

- `switch / case`
 - Nach der `switch`-Anweisung *soll* immer ein Leerzeichen folgen.
 - Die `case` Anweisungen, sowie die `default` Anweisung *sollen* dieselbe Einrückungstiefe wie die `switch`-Anweisung haben, um unnötige Verschachtelungstiefe zu vermeiden.
 - Leere `case`-Anweisungen *müssen* mit dem Kommentar `/* Fall through */` versehen sein.

```
switch (Bedingung)
{
case option1:
    ...
    break;

case option2:
    /* Fall through */

case option3:
    ...
    break;

default:
    ...
    break;
}
```

1.6 Prüfung auf Gleichheit

- Bei Prüfung auf Gleichheit *müssen* Literale das erste Argument sein.

```
/** ... */  
if (0 == counter)  
{  
    good = true;  
}
```

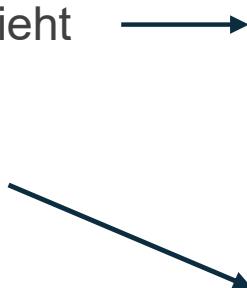
```
/** ... */  
if (counter == 0)  
{  
    bad = true;  
}
```



„Yoda condition“

1.7 Kommentare

- Die Kommentierung *muss* in der Doxygen-Syntax erfolgen, sodass aus ihr eine Dokumentation erzeugt werden kann.
- Folgende Stile *können* zur Kommentierung von Doxygen relevanten Kommentaren verwendet werden:
 - Vorwärts-Kommentar, d. h. der Kommentar bezieht sich auf das nachfolgende Element.
 - Rückwärts-Kommentar, d. h. der Kommentar bezieht sich auf das voranstehende Element.



```
/** ... */  
bool myVariable;  
  
/**  
 * ...  
 * ...  
 */  
void myMethod(void);
```

```
bool myVariable; /**< ... */
```

- Kommentierung von Funktionen *muss* die Beschreibung der Argumente beinhalten.
- Kommentierung von Funktionen *muss* bei der Deklaration erfolgen.
- Kommentierung von Funktionen *muss* die Beschreibung der Rückgabewerte beinhalten (sofern nicht `void`).

```
/**  
 * This ...  
 *  
 * @param[in] myArgument The ...  
 *  
 * @return If it is successful, it will return true otherwise false.  
 */  
bool myMethod(uint8_t myArgument);
```

- Folgende Stile *können* zur Kommentierung von Funktionen verwendet werden:
 - Kurz-Kommentierung von Funktionen *muss* im imperativ erfolgen

```
/** @brief Start / Stop / etc. ... */
```
 - Ausführliche Kommentierung von Funktionen *muss* aus ganzen Sätzen bestehen

```
/**  
 * This function starts / stops / etc. ...  
 */
```
- Achtung, Kommentare im Algorithmus, d. h. in Methodenrümpfen müssen ohne Doxygen Einleitung erzeugt werden.

```
/* ... */
```

2.3 Magic Numbers

Sog. „Magische Zahlen“^{*)} *dürfen nicht* verwendet werden.

- Davon ausgenommen sind Index- und Koordinatenzugriffe.

Falsch:

```
setMotorSpeed(255);
```

Richtig:

```
#define MAXIMUM_MOTOR_SPEED (255)

[...]

setMotorSpeed(MAXIMUM_MOTOR_SPEED);
```

Richtig:

```
lcd.gotoXY(2,3);
```

^{*)} „Magische Zahlen“ (auch „hart kodierte Zahlen“ genannt) sind im Quellcode auftauchende Zahlenlitterale, deren Bedeutung sich nicht unmittelbar erkennen lässt – die Bedeutung ist somit „magisch“.