ANSI-C, eigene Definitionen

Eigene Defines im C-Quelltext

Folgende Defines im Quelltext sind hilfreich und verbessern die Typensicherheit und die Lesbarkeit des Programmquelltexts.

```
#ifndef FALSE           // Boolscher Wert FALSE
#define FALSE  0
#endif

#ifndef TRUE            // Boolscher Wert TRUE
#define TRUE   !FALSE
#endif

#ifndef NULL            // Null-Zeiger für Pointer-Initialisierung
#define NULL   0
#endif
```

Pointer müssen initialisiert sein, damit keine unliebsamen Effekte entstehen.
Pointer-Initialisierung mit NULL, falls noch nicht klar ist, auf welches Objekt der Pointer zeigen soll.
Beispiel:

```
char * pText = NULL;    // Pointer auf einen Textstring
```

Beachte auch das **ASCII Nullzeichen NUL**. Dies ist die ASCII-Entsprechung des Zeichens '\0', bei dem alle Bits auf 0 gesetzt sind (Byte-Wert == 0). NUL oder '\0' wird als Abschlusszeichen in C-Strings verwendet, siehe: C-String-Terminierung.

Eigene Datentypen UBYTE und BOOL

Die Datentypen UBYTE für vorzeichenlose Binärdaten und BOOL für TRUE/FALSE-Bedingungen fehlen in ANSI-C, darum definieren wir sie jeweils im Programmkopf selbst.

```
typedef unsigned char UBYTE;   // unsigned byte
typedef unsigned char BOOL;    // bool datatype
```

Sprechende Namen für BOOL-Variablen: z.B. *isReady, isOverflow, hasChanged, canBegin* etc.
Verwendungsbeispiele von BOOL-Variablen:

```
If (isReady) {
   canBegin = TRUE;
}
...
If (isOverflow) {
   OutputErrorMesssage("OverflowError");
   isOverflow = FALSE;
}
...
if (hasChangedColor) {
   GetNewColor();
}
```

Datentyp BIT

ANSI-C kennt keinen Datentyp für Einzelbits.

Es gibt jedoch Compiler-spezifische Deklarationen und Bezeichner wie z.B. "sbit" in der KEIL-Umgebung, die allerdings *nicht ANSI-kompatibel* sind und die Portabilität des Codes einschränken.

```
sbit P12 = P1^2;  // KEIL-spezifische Bit-Deklaration
```

Werden in der Applikation mehrere Bits z.B. als Flags benötigt, kann eine **struct** deklariert und Einzelbits definiert werden (siehe Literatur über ANSI-C). Dies macht aber nur Sinn, falls sehr viele Einzelbits benötigt werden und der Speicherplatz der Anwendung knapp ist.
Eine einfachere, portable Lösung wäre, für jedes Bit ein Byte zu "verschwenden", dem die Werte 0 oder 1 zugewiesen werden.
Alternativ könnten in einem Byte mehrere Bits untergebracht werden, indem auf sie mittels C-Logikoperationen zugegriffen wird (ANSI-C Logikoperatoren: | & ^ ~).

Bitmanipulation mit Logikoperatoren, Beispiele:

```
unsigned char Bits;

Bits &= 0x7F;  // MSBit (Bit 7) löschen, ohne restliche Bits zu verändern

Bits |= 0x10;  // Bit 4 setzen, ohne restliche Bits zu verändern

while (Bits & 0x04) { ... }  // Bit 2 testen mit Maske 0000‘0100 binär
```

***