

AVR-8-bit-Mikrocontroller

Arbeiten mit CodeVisionAVR C-Compiler

Teil 04 - Der Preprozessor

Teil 01 - Einführung

- 1 Eine Einführung in C
 - 1.1 Warum C ?
 - 1.2 Wie entstand C ?
 - 1.3 Der AVR-Mikrocontroller in einem eingebetteten System

Teil 02 - Erste Schritte

- 2 Grundausstattung und Installation der Werkzeuge
 - 2.1 Testboard, Mikrocontroller, Programmer
 - 2.1.1 Testboard und Mikrocontroller
 - 2.2.2 ISP-Programmer
 - 2.2 AVR Studio installieren
 - 2.3 Versteht mich der Mikrocontroller ?
 - 2.4 CodeVisionAVR C-Compiler installieren
 - 2.5 Editieren von Quell-Dateien

Teil 03 - Aufbau eines C-Projektes

- 3 Was ist ein C-Projekt ?
 - 3.1 Erzeugen eines C-Projektes
 - 3.1.1 Ein neues Projekt beginnen
 - 3.1.2 Ein C-Projekt generieren
 - 3.2 Dateistruktur eines C-Projektes
 - 3.3 Einbindung von AVR Studio in den CVAVR
 - 3.4 AVR Studio Debugger
 - 3.5 C-Compiler-Optionen

Teil 04 - Der Preprozessor

- 4 Preprozessor-Anweisungen
 - 4.1 Struktur der C-Quell-Programme
 - 4.2 `#include`-Anweisung
 - 4.3 `#define`-Anweisung (Makro)
 - 4.3.1 Makros ohne Parameter
 - 4.3.2 Makros mit Parametern
 - 4.4 `#undef`-Anweisung
 - 4.5 `#if`-, `#ifdef`-, `#ifndef`-, `#else`- und `#endif`-Anweisungen
 - 4.6 Andere Preprozessor-Anweisungen

Teil 05 - Syntax der C-Programme

- 5 Die Syntax der C-Programme
 - 5.1 C-Quell-Programme
 - 5.1.1 Kommentare
 - 5.1.2 Deklarationen (Vereinbarungen)
 - 5.1.3 Die Funktion `main`
 - 5.1.4 Schlüsselwörter (Keywords) des CodeVisionAVR C-Compilers
 - 5.2 Konstanten und Variablen
 - 5.2.1 Zahlensysteme
 - 5.2.2 Datentypen
 - 5.2.3 Konstanten
 - 5.2.4 Variablen
 - 5.3 Operatoren
 - 5.3.1 Arithmetische Operatoren
 - 5.3.2 Relationale Operatoren
 - 5.3.3 Logische und bitweise wirkende Operatoren
 - 5.3.4 Andere Operatoren und Shortcuts
 - 5.4 Komplexe Objekte in C
 - 5.4.1 Funktionen
 - 5.4.2 Funktions-Prototypen
 - 5.4.3 Pointers und Arrays
 - 5.4.3.1 Pointers
 - 5.4.3.1.1 Pointers in Verbindung mit `flash` und `eeprom`
 - 5.4.3.1.2 Pointers in Verbindung mit `typedef`
 - 5.4.3.2 Arrays

AVR-8-bit-Mikrocontroller

Arbeiten mit CodeVisionAVR C-Compiler

Teil 04 - Der Preprozessor

- 5.4.3.2.1 Ein-dimensionale Arrays
- 5.4.3.2.2 Zwei-dimensionale Arrays
- 5.4.3.2.3 Drei-dimensionale Arrays
- 5.4.3.3 Benutzen der Array-Namen als Pointers
- 5.4.3.4 Arrays von Pointers
- 5.4.3.5 Pointers auf Funktionen (Funktionszeiger)
- 5.4.3.6 Funktionen in Verbindung mit `typedef`
- 5.4.4 Strukturen und Unionen
 - 5.4.4.1 Strukturen
 - 5.4.4.2 Unionen
- 5.4.5 Komplexe Typen (eine Zusammenfassung)
- 5.5 Steuerung des Programmablaufs
 - 5.5.1 Anweisungsblöcke { . . . }
 - 5.5.2 Die Anweisung `if`
 - 5.5.3 Die Anweisung `if` in Verbindung mit `else`
 - 5.5.4 Die Fallunterscheidung `switch`
 - 5.5.5 Die Schleife `for`
 - 5.5.6 Die Schleife `while`
 - 5.5.7 Die Schleife `do` in Verbindung mit `while`
- 5.6 Arbeiten mit den Ein-/Ausgabe-Ports

Teil 06 - Modularer Aufbau der AVR-Projekte

- 6 Modularer Aufbau
 - 6.1 Das Konzept
 - 6.2 Nomenklatur
 - 6.3 Die globalen Header-Dateien
 - 6.3.1 Die globale Header-Datei `typedefs.h` (Typ- und Bit-Definitionen)
 - 6.3.2 Die globale Header-Datei `iomx.h` (Definitionen aller Register-Bits)
 - 6.3.3 Die globale Header-Datei `macros.h` (Definitionen von Makros)
 - 6.3.4 Die globale Header-Datei `switches.h` (Definitionen von Schalter-Makros)
 - 6.4 Die Module
 - 6.5 Anwendung der Module
 - 6.5.1 Das Modul `application` (Anwendungs-Steuerung)
 - 6.5.2 Das Modul `lcd_2wire` (Ausgabe auf LCD-20x4)
 - 6.5.3 Das Modul `num_conversion` (Typ-Konvertierung nach ASCII)
 - 6.5.4 Das Modul `adc_ref-1-1` (ADC mit interner Referenz 1,1 V)
 - 6.5.5 Das Modul `rc5_decoder` (RC5-IR-Fernsteuerung-Dekoder)
 - 6.5.6 Das Modul `rc5_encoder` (RC5-IR-Fernsteuerung-Enkoder)
 - 6.5.7 Das Modul `usart` (USART-Steuerung)
 - 6.5.8 Das Modul `twi_master` (I²S- bzw. TWI-Steuerung)
 - 6.5.9 Das Modul `timer0_pwm` (TIMER0-Steuerung)

Teil 07 - Anhang

- 7 Anhang
 - 7.1 Begriffe und Definitionen
 - 7.2 Eigene Bibliothek
 - 7.3 AVR-Projekte (Programmbeschreibungen)
 - 7.3.1 AVR-Projekt PB_LED
 - 7.3.2 AVR-Projekt 2_Draht_LCD
 - 7.3.3 AVR-Projekt IRDMS

AVR-8-bit-Mikrocontroller

Arbeiten mit CodeVisionAVR C-Compiler

Teil 04 - Der Preprozessor

Vorbemerkung

Nichts ist vollkommen - und nichts ist endgültig! So auch nicht dieses Tutorial! Deshalb bitte immer erst nach dem neuesten Datum schauen. Vielleicht gibt es wieder etwas Neues oder eine Fehlerbereinigung oder eine etwas bessere Erklärung. Wer Fehler findet oder Verbesserungen vorzuschlagen hat, bitte melden (info@alenck.de).

Immer nach dem Motto: Das Bessere ist Feind des Guten und nichts ist so gut, dass es nicht noch verbessert werden könnte.

Gravierende Änderungen gegenüber der Vorversion

1. Neuer Abschnitt **5.6** in der Inhaltsangabe
2. Anpassung der Schrift-Typen an die **C**-Schreibweise

AVR-8-bit-Mikrocontroller

Arbeiten mit CodeVisionAVR C-Compiler

Teil 04 - Der Preprozessor

4 Preprozessor-Anweisungen

4.1 Struktur der C-Quell-Programme

Alle C-Quell-Programme (sowohl C-Dateien wie Header-Dateien) bestehen aus mehreren Teilen wie Kommentare, Preprozessor-Anweisungen, Deklarationen, Definitionen, Ausdrücke, Zuweisungen und Funktionen. Alle diese Teile - bis auf die Preprozessor-Anweisungen - werden erst im **Teil 05 - Syntax der C-Programme** beschrieben.

Preprozessor-Anweisungen sind tatsächlich kein Bestandteil der C-Syntax. Aber sie sind komfortabel und so beliebt, dass sie praktisch zum Standard geworden sind. Der Preprozessor ist - wie die Vorsilbe **Pre** (besser eingedeutscht: **Prä**) schon andeutet - ein vorgezogener Schritt am Anfang der Projekt-Erzeugung, der noch vor der aktuellen Kompilierung erfolgt. Diese Anweisungen erzeugen aus den einzelnen C- und Header-Dateien mit ihrer mehr oder weniger starken Strukturierung erst den sequentiellen Programm-Code. Oder mit anderen Worten: Der Preprozessor mit seinen diversen Anweisungen macht die für den Programmierer sinnvolle und überschaubare Strukturierung seiner Anwendung erst möglich. Das beste Beispiel dafür ist die Modularisierung der AVR-Projekte, wie sie im **Teil 06 - Modularer Aufbau der AVR-Projekte** beschrieben wird. Der Preprozessor sorgt also für die gewünschte Verbindung aller Programmteile.

Preprozessor-Anweisungen beginnen stets mit # und können wie folgt gegliedert werden:

Preprozessor-Anweisung #include

```
#include <Dateiname>
#include "Dateiname"
```

Preprozessor-Anweisungen (Makros) #define und #undef

```
#define Identifier Replacement
#define Identifier(Identifier,...,Identifier) Replacement
#undef Identifier
```

Preprozessor-Anweisungen #if, #elif, #ifdef, #ifndef, #else und #endif

```
#if Identifier
#elif Identifier
#ifdef Identifier
#ifndef Identifier
#else
#endif
```

Andere Preprozessor-Anweisungen

```
#pragma Qualifier
#line Konstante Identifier
#error Fehlermeldung
#warning Warnungsmeldung
#message Allgemeine Meldung
#asm
#endasm
```

4.2 #include-Anweisung

Eine #include-Anweisung besteht aus dem "Befehlswort" #include und der Angabe einer Header-Datei. In der C-Datei wird durch den Preprozessor diese Anweisung durch den Code der Header-Datei ersetzt, d.h. an dieser Stelle wird das Coding von der Header-Datei eingefügt.

Im Rahmen der Strukturierung gibt es auch viele Deklarationen und Funktionen, die in der Normung von ANSI (**American National Standards Institute**) der Sprache C nicht enthalten sind. Sie sind dennoch notwendig oder sehr nützlich. Man "versteckt" sie in sog. Bibliotheken (vergl. **Bild 4.2-01**). Damit der Preprozessor sie auch findet und sie noch vor dem Übersetzen des C-Quelltextes in die Quelle einfügen kann, muss man ihm mitteilen, dass er diese globalen Header-Dateien (Kopf-Dateien), mit einschließen (to include) soll. **Header-Dateien** erkennt man an der Endung **.h**

AVR-8-bit-Mikrocontroller

Arbeiten mit CodeVisionAVR C-Compiler

Teil 04 - Der Preprozessor

Beispiel 4.2-01: Aus der Datei `main.c` im AVR-Projekt `PB_LED`

```
#include "application.h"           // Definitionen für diese Applikation
```

Beispiel 4.2-02: Aus der Datei `typedefs.h` (in allen Projekten vorhanden!)

```
#include <stdio.h>                 // Funktions-Prototypen zur Ein-/Ausgabe
#include <stdlib.h>                 // Funktions-Prototypen zur Konvertierung
#include <ctype.h>                 // Funktions-Prototypen zur Typ-Abfrage
#include <delay.h>                 // Funktions-Prototypen zur Verzögerung
#include <string.h>                // Funktions-Prototypen zur String-Behandlung
#include <limits.h>                // Definitionen der min/max-Groessen der Typen
#include "iomx8.h"                 // Definition der Register-Bits
#include "macros.h"                // Definition besonderer Makros
#include "switches.h"              // Definition von Schalter-Makros
```

Header-Dateien können auch wieder `#include`-Anweisungen enthalten und diese wiederum usw., so dass Verschachtelungen unterschiedlicher Tiefen entstehen können (Kaskadierung). CVAVR lässt eine Verschachtelungstiefe bis zu 16 zu. Ob diese tiefe Verschachtelung noch sinnvoll und überschaubar ist, ist zu bezweifeln. Bei einem sinnvoll modular aufgebauten Projekt sollte eine größere Verschachtelungstiefe vermeidbar sein.

Wenn die Header-Datei in spitzen Klammern (z.B. `#include <delay.h>`) aufgerufen wird, dann handelt es sich um eine globale Header-Datei aus der Standard-Bibliothek des Compilers und wird dort automatisch gesucht (in der Regel - wenn keine Änderung durch den Installateur stattgefunden hat - wird die Bibliothek bei der Installation von CVAVR im Ordner `C:\cvavr_atm18_eval\inc` angelegt). Wird die Header-Datei jedoch in Anführungszeichen (z.B. `"application.h"`) aufgerufen, so handelt es sich um eine selbst erstellte Header-Datei, die im eigens für das Projekt angelegten Ordner abgelegt sein muss. Die `#include`-Anweisungen können theoretisch an jeder Stelle im Code platziert werden, doch ist es üblich und auch sinnvoll, sie am Anfang des Programms aufzuführen. Alle im Projekt [PB_LED](#) angesprochenen Dateien und die in ihnen enthaltenen `#include`-Anweisungen sind in dem [Bild 4.2-01](#) enthalten.

Das Bild - und auch die Vielzahl der Dateien - mag im ersten Moment verwirren, aber es zeigt eigentlich nur das modulare Zusammenwirken aller Teile eines C-Projektes (man beachte den jeweiligen Text in den Kästchen), von denen die spezifischen Teile des Mikrocontrollers nur zum Anfang näher betrachtet werden. Was der Preprozessor von CVAVR daraus macht, zeigen die nachfolgenden Aufstellungen.

Eine ausführlichere Beschreibung der selbst erstellten Header-Dateien (und der selbst kreierten Module) ist im [Teil 06 Modularer Aufbau der AVR-Projekte](#) enthalten. Die globalen Header-Dateien von CVAVR sind in den **Help Topics** des Compilers unter **Library Functions Reference** beschrieben: Die Hilfe kann auch als Hilfe-Datei `CVAVR.CHM` aus dem Ordner `C:\cvavr_atm18_eval\bin` aufgerufen werden.

AVR-8-bit-Mikrocontroller

Arbeiten mit CodeVisionAVR C-Compiler

Teil 04 - Der Preprozessor

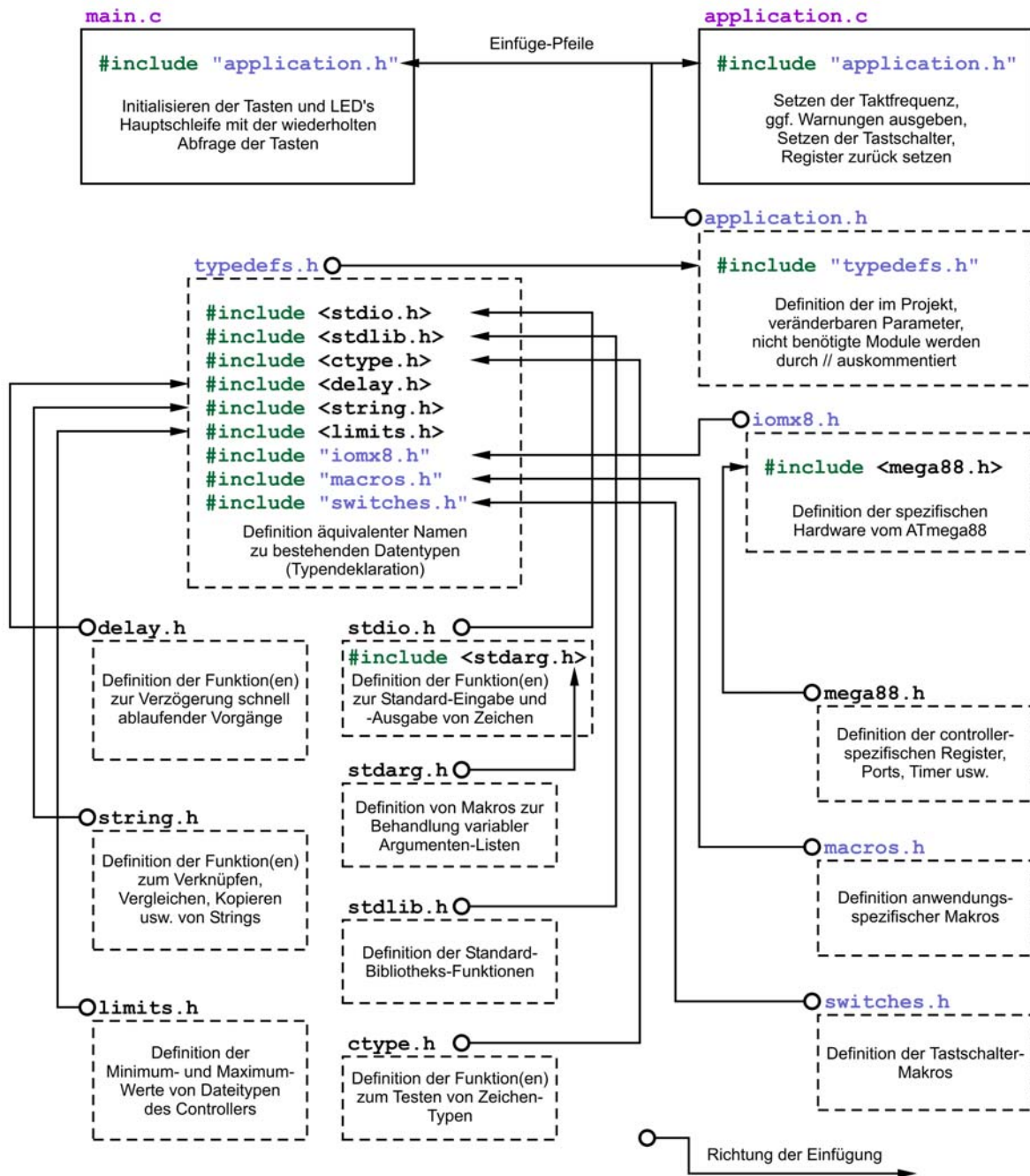


Bild 4.2-01: Struktur des Beispiel-Projektes PB_LED

Preprozessor-Ablauf (betrachtet werden hier nur die #include-Anweisungen):

Der Preprozessor erzeugt für die Datei `main.c` nacheinander die folgenden Einfügungen (für die Modul-Datei `application.c` gelten analog dieselben Einfügungen):

An der Stelle von: `#include "application.h"` in der Datei: `main.c` wird das Coding aus folgender Datei eingesetzt: `..\projekt_ordner\application.h`

Der Preprozessor setzt seine Arbeit im gerade eingesetzten Coding von `application.h` fort und trifft dort auf `#include "typedefs.h"`.

An der Stelle von: `#include "typedefs.h"` in der Datei: `main.c` wird das Coding aus folgender Datei eingesetzt: `..\projekt_ordner\typedefs.h`

Der Preprozessor setzt seine Arbeit im gerade eingesetzten Coding von `typedefs.h` fort und trifft dort zunächst auf die Anweisung `#include <stdio.h>`, eine globale Header-Datei des Compilers:

AVR-8-bit-Mikrocontroller

Arbeiten mit CodeVisionAVR C-Compiler

Teil 04 - Der Preprozessor

An der Stelle von: <code>#include <stdio.h></code>	in der Datei: <code>main.c</code>	wird das Coding aus folgender Datei eingesetzt: <code>C:\cvavr_atm18_eval\inc\stdio.h</code>
--	---	--

Der Preprozessor setzt seine Arbeit im gerade eingesetzten Coding von `stdio.h` fort und trifft dort auf die Anweisung `#include <stdarg.h>`, eine globale Header-Datei des Compilers:

An der Stelle von: <code>#include <stdarg.h></code>	in der Datei: <code>main.c</code>	wird das Coding aus folgender Datei eingesetzt: <code>C:\cvavr_atm18_eval\inc\stdarg.h</code>
---	---	---

Der Preprozessor setzt seine Arbeit zunächst im Coding von `stdarg.h`, dann im `stdio.h` und schließlich wieder im Coding von `typedefs.h` fort und trifft der Reihe nach auf die `#include`-Anweisungen der globalen Header-Dateien:

```
#include <stdlib.h>
#include <ctype.h>
#include <delay.h>
#include <string.h>
#include <limits.h>
```

An der Stelle von: <code>#include <stdlib.h></code> <code>#include <ctype.h></code> <code>#include <delay.h></code> <code>#include <string.h></code> <code>#include <limits.h></code>	in der Datei: <code>main.c</code> <code>main.c</code> <code>main.c</code> <code>main.c</code> <code>main.c</code>	wird das Coding aus folgender Datei eingesetzt: <code>C:\cvavr_atm18_eval\inc\stdlib.h</code> <code>C:\cvavr_atm18_eval\inc\ctype.h</code> <code>C:\cvavr_atm18_eval\inc\delay.h</code> <code>C:\cvavr_atm18_eval\inc\string.h</code> <code>C:\cvavr_atm18_eval\inc\limits.h</code>
---	---	---

Der Preprozessor durchläuft diese Codings von `stdlib.h` bis `limits.h` und trifft auf die `#include`-Anweisung der selbst erstellten globalen Header-Datei

```
#include "iomx8.h"
```

Der Preprozessor setzt seine Arbeit im gerade eingesetzten Coding von `iomx8.h` fort und trifft dort auf die Anweisung `#include <mega88.h>`, ebenfalls eine globale Header-Datei des Compilers:

An der Stelle von: <code>#include <mega88.h></code>	in der Datei: <code>main.c</code>	wird das Coding aus folgender Datei eingesetzt: <code>C:\cvavr_atm18_eval\inc\mega88.h</code>
---	---	---

Nachdem der Preprozessor das eingesetzte Coding von `mega88.h` und `iomx8.h` durchlaufen hat, werden noch die selbst erstellten globalen Header-Dateien, deren `#include`-Anweisungen noch im Coding von `typedefs.h` enthalten sind eingefügt:

An der Stelle von: <code>#include "macros.h"</code> <code>#include "switches.h"</code>	in der Datei: <code>main.c</code> <code>main.c</code>	wird das Coding aus folgender Datei eingesetzt: <code>..\projekt_ordner\macros.h</code> <code>..\projekt_ordner\switches.h</code>
---	--	--

Wenn alle diese Einfügungen abgeschlossen sind, müssten alle Einstellungen für das Projekt getätigt worden sein, so dass die Kompilierung stattfinden kann.

4.3 #define-Anweisung (Makro)

Eine **#define-Anweisung** kann als eine einfache Art der Text-Ersetzung (**Replacement**) aufgefasst werden und bildet in der **C-Programmierung** ein **Makro**.

Die Standard-Form ist

```
#define Identifier_NamespaceReplacement_TextCR/LF
```

Der Preprozessor wird auf Grund dieser Anweisung vom Auftreten der Anweisung bis zum Ende der Quell-Datei und seinen Einfügungen überall wo der Begriff **Identifier_Name** erscheint, diesen durch den **Replacement_Text** (Makro-Rumpf) ersetzen. Natürlich nicht, wenn er innerhalb von Zeichenketten oder innerhalb von Namen auftaucht. Bei einer `#define`-Anweisung endet der **Replacement_Text** gewöhnlich mit **CR/LF**. Normalerweise ist der Ersatztext also mit dem Ende der Zeile abgeschlossen. Sollte er jedoch länger sein müssen, so wird am Ende der Zeile, wo die Fortsetzung stattfinden soll, ein Backslash (\) gesetzt.

AVR-8-bit-Mikrocontroller

Arbeiten mit CodeVisionAVR C-Compiler

Teil 04 - Der Preprozessor

Beim CVAVR wird - entgegen vielen anders lautenden Beschreibungen - ein Zeilen-Kommentar **nicht mit in den Replacement_Text eingebunden**, sondern der Preprozessor interpretiert // ebenfalls als Ende des Replacements!

Im Allgemeinen werden **#define**-Anweisungen benutzt, um Konstanten zu deklarieren (Makros ohne Parameter). Jedoch können im **Identifizier_Name** und in dem **Replacement_Text** Parameter aufgeführt sein, so dass der Ersatztext vom Aufruf des Makros abhängt. Makros werden erst expandiert und dann wird der Ausdruck bewertet.

4.3.1 Makros ohne Parameter

Beispiel 4.3.1-01: Aus der Datei `main.c` im AVR-Projekt `PB_LED`

#define-Anweisungen **ohne** Parametern: Es werden bei dieser Ersetzungskette durch den Preprozessor die **#define**-Anweisungen sowohl für `LED1_DDR`, `LED2_DDR` und `LED3_DDR` als auch für `LED1_BIT`, `LED2_BIT` und `LED3_BIT` behandelt. Diese sind zuständig für das Datenrichtungsregister **DDRC** (Bit2, Bit3 und Bit4).

Im Hauptprogramm `main.c` treten die folgenden Zuweisungs-Anweisungen auf:

```
// Initialisieren der LED's
LED1_DDR |= LED1_BIT;           // Setze LED1-Pin auf Ausgabe
LED2_DDR |= LED2_BIT;           // Setze LED2-Pin auf Ausgabe
LED3_DDR |= LED3_BIT;           // Setze LED3-Pin auf Ausgabe
...
```

die in "normaler Schreibweise" wie folgt aussehen würden (vergl. **Shortcuts** im Abschnitt **5.3.4**):

```
// Initialisieren der LED's
LED1_DDR = LED1_DDR | LED1_BIT; // Setze LED1-Pin auf Ausgabe
LED2_DDR = LED2_DDR | LED2_BIT; // Setze LED2-Pin auf Ausgabe
LED3_DDR = LED3_DDR | LED3_BIT; // Setze LED3-Pin auf Ausgabe
...
```

Die Makros für `LED1_DDR` usw. sowie für `LED1_BIT` usw. sind in der Header-Datei `application.h` definiert:

```
...
//-----
// LED's jeweils fuer Datenrichtungsregister, Port- und Bit-Nummer definieren,
// Anwendung im Projekt (direkt ueber main.c):
//-----

#define LED1_DDR      DDRC      // LED1 an Port C Data Direction Register
#define LED1_PRT      PORTC     // LED1 an Port C Input Pins Address
#define LED1_BIT      PC2       // LED1 an PC2 = PINC2 = Bit2

#define LED2_DDR      DDRC      // LED2 an Port C Data Direction Register
#define LED2_PRT      PORTC     // LED2 an Port C Input Pins Address
#define LED2_BIT      PC3       // LED2 an PC3 = PINC3 = Bit3

#define LED3_DDR      DDRC      // LED3 an Port C Data Direction Register
#define LED3_PRT      PORTC     // LED3 an Port C Input Pins Address
#define LED3_BIT      PC4       // LED3 an PC4 = PINC4 = Bit4
...
```

Damit ersetzt der Preprozessor die Anweisungen wie folgt:

```
// Initialisieren der LED's
DDRC |= PC2;           // Setze LED1-Pin auf Ausgabe
DDRC |= PC3;           // Setze LED2-Pin auf Ausgabe
DDRC |= PC4;           // Setze LED3-Pin auf Ausgabe
```

Jetzt müssen auch `PC2`, `PC3` und `PC4` durch ihre Replacements ersetzt werden. Die dafür zuständigen **#define**-Anweisungen findet der Preprozessor in der Header-Datei `iomx8.h`:

AVR-8-bit-Mikrocontroller

Arbeiten mit CodeVisionAVR C-Compiler

Teil 04 - Der Preprozessor

```
//-----  
// Register PINC (Port C Input Pins Address)                               Adresse (0x26)  
//-----  
...  
#define PINC4      BIT4           // Port-Input-Pin-Adress-Register PINC  
#define PC4        BIT4           // Bit4  
#define PINC3      BIT3           // Port-Input-Pin-Adress-Register PINC  
#define PC3        BIT3           // Bit3  
#define PINC2      BIT2           // Port-Input-Pin-Adress-Register PINC  
#define PC2        BIT2           // Bit2  
...
```

und ersetzt die Anweisungen in der Datei `main.c` wie folgt:

```
// Initialisieren der LED's  
DDRC |= BIT2;           // Setze LED1-Pin auf Ausgabe  
DDRC |= BIT3;           // Setze LED2-Pin auf Ausgabe  
DDRC |= BIT4;           // Setze LED3-Pin auf Ausgabe
```

Schließlich müssen auch noch BIT2, BIT3 und BIT4 durch ihre Replacements ersetzt werden, die jetzt konkrete Konstanten enthalten sollen. Die dafür zuständigen Makros (und viele andere) sind so allgemein, dass ihre `#define`-Anweisungen in der globalen Header-Datei `typedefs.h` für das AVR-Projekt abgelegt sind:

```
//-----  
// Definitionen der Bit-Nummern  
//-----  
...  
#define BIT2      0x04  
#define BIT3      0x08  
#define BIT4      0x10  
...
```

Der letzte Schritt des Preprozessors in dieser Angelegenheit ergibt die realen Anweisungen:

```
// Initialisieren der LED's  
DDRC |= 0x04;           // Setze LED1-Pin auf Ausgabe  
DDRC |= 0x08;           // Setze LED2-Pin auf Ausgabe  
DDRC |= 0x10;           // Setze LED3-Pin auf Ausgabe
```

Das sind also bitweise ODER-Verknüpfungen von Bit2, Bit3 und Bit4 im Datenrichtungsregister `DDRC` mit den vorher definierten Konstanten; das Ergebnis wird wieder im Datenrichtungsregister abgespeichert und sorgt so für die Initialisierung dieser Bits für die Ausgabe.

Ergebnis in dieser Anwendung für das Datenrichtungsregister `DDRC` ist schließlich (Ursprungswert von `DDRC` ist nach dem Restart immer binär `00000000`):

```
DDRC = DDRC | 00001100 = 00001100      Bit2 wird als Ausgang konfiguriert  
DDRC = DDRC | 00001000 = 00001100      Bit2 und Bit3 werden als Ausgänge konfiguriert  
DDRC = DDRC | 00010000 = 00011100      Bit2, Bit3 und Bit4 werden als Ausgänge konfiguriert
```

Anmerkung: Damit kann über die Bit2 bis Bit4 am Port C eine Ausgabe (Schalten der LEDs) stattfinden. Eine LED leuchtet erst dann, wenn das betreffende Bit des Port C logisch **1** aufweist (5 V = **High**). Die Zählung der Bits erfolgt von rechts nach links beginnend mit Bit0, Bit1, Bit2 usw.

4.3.2 Makros mit Parametern

Makros mit Parametern (Argumenten) spielen für die Programmierung der AVR-Mikrocontroller eine große Rolle. Sie haben mit ihrem Flash-Speicher relativ viel Speicherplatz jedoch vergleichsweise nur wenig RAM und nur eine geringe CPU-Leistung. Zusätzlich ist der Befehlssatz eher auf geringen Platzbedarf als auf höhere Geschwindigkeit optimiert. Für eine Funktion müssen beispielsweise bis zu 100 Taktzyklen für ihre Abarbeitung ohne Berücksichtigung des Funktionskörpers aufgewendet werden, während für jeden Befehl nur rund 3 bis 4 Takte gebraucht werden.

AVR-8-bit-Mikrocontroller

Arbeiten mit CodeVisionAVR C-Compiler

Teil 04 - Der Preprozessor

Das Missverhältnis ist nur dadurch zu verbessern, indem man gleich in Assembler programmiert oder eben Makros mit Parametern verwendet. Ein einfaches Beispiel für ein Makro mit Parametern ist die Maximums-Bildung aus 2 Größen, die auch wieder aus mehreren Variablen zusammengesetzt sein können.

a und b können im folgenden Beispiel beliebig gewählt werden, dürfen aber nicht mit anderen Variablen-Namen identisch sein. Der etwas kryptische 2. Klammerausdruck ist der Bedingungsoperator `?:` (siehe auch: Abschnitt **5.3.4 Andere Operatoren und Shortcuts**), der besagt: Wenn der Ausdruck a größer ist als der Ausdruck b, dann wähle den 1. Ausdruck nach dem `?` und andernfalls den 2. Ausdruck.

```
...
#define MAXIMUM(a, b) ((a) > (b) ? (a) : (b))
...
```

Im Quellprogramm steht irgendwo der Befehl

```
...
    ergebnis = MAXIMUM(par1+par2, par3+par4);
...
```

den der Preprozessor durch folgende Zeile ersetzt:

```
...
    ergebnis = ((par1+par2) > (par3+par4) ? (par1+par2) : (par3+par4));
...
```

Beispiel 4.3.2-01: Aus der Datei `main.c` im AVR-Projekt `PB_LED`

#define-Anweisungen mit Parametern: Bei dieser Ersetzungskette durch den Preprozessor wird das Datenrichtungsregister `DDRB` auf Null gesetzt (alle Pins werden als Eingänge konfiguriert) und die Eingabe-Bits Bit3 bis Bit5 von `PINB` auf logisch **1**.

Im Hauptprogramm `main.c` treten die folgenden Initialisierungs-Anweisungen auf:

```
// Initialisieren der Tasten (Pushbuttons)
S1_INIT();
S2_INIT();
S3_INIT();
```

Sie korrespondieren mit den Definitionen in der Header-Datei `switches.h`

```
//-----
// Definitionen von Tasten-Makros
//-----
#define S1_INIT() {S1_DDR &= ~S1_BIT; S1_PRT |= S1_BIT;} // Setze S1-Pin (K8.1)
// der Taste 1 auf Eingabe
#define S2_INIT() {S2_DDR &= ~S2_BIT; S2_PRT |= S2_BIT;} // Setze S2-Pin (K8.2)
// der Taste 2 auf Eingabe
#define S3_INIT() {S3_DDR &= ~S3_BIT; S3_PRT |= S3_BIT;} // Setze S3-Pin (K8.3)
// der Taste 3 auf Eingabe
```

so dass der Preprozessor daraus zunächst die folgenden Zeilen im Hauptprogramm erzeugt:

```
// Initialisieren der Tasten (Pushbuttons)
{S1_DDR &= ~S1_BIT; S1_PRT |= S1_BIT;} ;
{S2_DDR &= ~S2_BIT; S2_PRT |= S2_BIT;} ;
{S3_DDR &= ~S3_BIT; S3_PRT |= S3_BIT;} ;
```

`S1_DDR`, `S2_DDR` und `S3_DDR` sowie `S1_BIT`, `S2_BIT` und `S3_BIT` sowie `S1_PRT`, `S2_PRT` und `S3_PRT` korrespondieren mit den Definitionen in der Header-Datei `application.h`:

```
...
//-----
// Tasten S1, S2 und S3 definieren
// Anwendung in den Projekten (indirekt ueber S1_INIT(), S2_INIT(), S3_INIT() in
// main.h => switches.h)
//-----
```

AVR-8-bit-Mikrocontroller

Arbeiten mit CodeVisionAVR C-Compiler

Teil 04 - Der Prozessor

```

#define S1_DDR DDRB // S1 an Port B Data Direction Register
#define S1_PRT PINB // S1 an Port B Input Pins Address
#define S1_BIT BIT3 // S1 an Bit3 von PINB

#define S2_DDR DDRB // S2 an Port B Data Direction Register
#define S2_PRT PINB // S2 an Port B Input Pins Address
#define S2_BIT BIT4 // S2 an Bit4 von PINB

#define S3_DDR DDRB // S3 an Port B Data Direction Register
#define S3_PRT PINB // S3 an Port B Input Pins Address
#define S3_BIT BIT5 // S3 an Bit5 von PINB

```

so dass auch diese ersetzt werden:

```

// Initialisieren der Tasten (Pushbuttons)
{DDRB &= ~BIT3; PINB |= BIT3;} ;
{DDRB &= ~BIT4; PINB |= BIT4;} ;
{DDRB &= ~BIT5; PINB |= BIT5;} ;

```

und jetzt auch noch die Ersetzung von BIT3, BIT4 und BIT5 durch die Definitionen aus der Header-Datei `typedefs.h`:

```

//-----
// Definitionen der Bit-Nummern
//-----
...
#define BIT3 0x08
#define BIT4 0x10
#define BIT5 0x20
...

```

Ergebnis des Preprozessors:

```

// Initialisieren der Tasten (Pushbuttons)
{DDRB &= ~0x08; PINB |= 0x08;} ;
{DDRB &= ~0x10; PINB |= 0x10;} ;
{DDRB &= ~0x20; PINB |= 0x20;} ;

```

Die Anweisungen besagen, dass das Datenrichtungsregister `DDRB` (Port B Data Direction Register) der Reihe nach mit `~0x08`, mit `~0x10` und `~0x20` UND-verknüpft wird und dass das Register `PINB` (Port B Input Pins Address) ebenfalls der Reihe nach mit `0x08`, mit `0x10` und `0x20` ODER-verknüpft wird. `~` bedeutet Negation oder Einerkomplement, d.h. alle Bits werden "umgedreht".

Das Ergebnis in dieser Anwendung für das Datenrichtungsregister `DDRB` und die Eingabepins von `PINB` ist schließlich (Ursprungswerte von `DDRB` und `PINB` sind nach dem Restart immer binär `00000000`):

<code>DDRB = DDRB & 11110111 = 00000000</code>	alle Bits sind als Eingänge konfiguriert
<code>DDRB = DDRB & 11101111 = 00000000</code>	alle Bits sind als Eingänge konfiguriert
<code>DDRB = DDRB & 11011111 = 00000000</code>	alle Bits sind als Eingänge konfiguriert
<code>PINB = PINB 00001000 = 00001000</code>	Bit3 ist auf Eingabe aktiviert
<code>PINB = PINB 00010000 = 00011000</code>	Bit3 und Bit4 sind auf Eingabe aktiviert
<code>PINB = PINB 00100000 = 00111000</code>	Bit3, Bit4 und Bit5 sind auf Eingabe aktiviert

Anmerkung: Das entspricht einer Spannung von jeweils +5 V (**High**) an den Pins `PB3` bis `PB5`, die erst beim Betätigen einer Taste `s1` bis `s3` auf Masse (**Low** = 0 V) geschaltet werden.

4.4 #undef-Anweisung

Falls ein bereits verwendeter Identifier ein anderes Replacement erhalten soll, muss er zuvor "ent"-definiert werden. Ein Identifier kann logischerweise keine zwei oder mehr verschiedene Replacements haben. Im folgenden Beispiel soll der Identifier `LED1_BIT` von `PC2` auf `PC4` geändert werden:

```

#define LED1_BIT PC2

```

AVR-8-bit-Mikrocontroller

Arbeiten mit CodeVisionAVR C-Compiler

Teil 04 - Der Preprozessor

```
...
...
#undef LED1_BIT
#define LED1_BIT PC4
```

Mehrere Definitionen mit demselben Replacement sind dagegen ohne weiteres möglich:

```
#define PINB7 BIT7 // Port-Input-Pin-Adress-Register PINB
#define PB7 BIT7 // Bit7
```

4.5 #if-, #elif-, #ifdef-, #ifndef-, #else- und #endif-Anweisungen

Wenn diese Anweisungen verwendet werden, so spricht man auch von "bedingter Kompilierung" (Conditional Compilation - es wird in der Literatur häufig die Arbeit des Preprozessors als Kompilation aufgefasst), denn mit diesen Preprozessor-Anweisungen ist es elegant möglich - noch vor der eigentlichen Kompilierung - Teile des Codes weg- oder zuzulassen. Auf Grund "logischer Schalter" die entweder vorher vom Programmierer definiert wurden oder die während der Arbeit des Preprozessors gesetzt werden, werden Teile des Codes übersprungen oder in die Kompilierung einbezogen.

In den Modulen von [Teil 06](#) wird von dieser Methode rege Gebrauch gemacht, um einmal erzeugte ausführliche Header-Dateien (Dateien mit möglichst umfassender Definition aller gebräuchlichen Ports usw.) einfach durch Setzen der "logischen Schalter" das Kompilat der jeweiligen Anwendung anzupassen.

Die "logischen Schalter" werden einfach durch Definition von Makro-Namen oder durch logische Verknüpfungen gebildet. Ein "logischer Schalter" ist immer **TRUE**, wenn sein numerischer Wert von Null verschieden ist, andernfalls ist er **FALSE**.

Allgemeine Syntax:

```
#if LOGISCHE_VERKNUEPFUNG1
    Hier folgt ein
    Anweisungs-Block1
#elif LOGISCHE_VERKNUEPFUNG2
    Hier folgt ein
    Anweisungs-Block2
#else
    Hier folgt ein
    Anweisungs-Block3
#endif
```

Die allgemeine Syntax besagt:

- Wenn die `LOGISCHE_VERKNUEPFUNG1` den Wert **TRUE** angenommen hat, dann wird der **Anweisungs-Block1** kompiliert.
- Wenn die `LOGISCHE_VERKNUEPFUNG2` den Wert **TRUE** angenommen hat, dann wird der **Anweisungs-Block2** kompiliert.
- Andernfalls wird der **Anweisungs-Block3** kompiliert.
- **#endif** schließt die "bedingte Kompilierung" ab.

Beispiel 4.5-01: Aus der Datei `main.c` im AVR-Projekt `PB_LED`

Vorbemerkung zum Verständnis der Einstellung des Prozessor-Taktes:

`_MCU_CLOCK_FREQUENCY_` ist ein vom CVAVR vordefiniertes Makro, das den in der Projekt-Konfiguration eingestellten Takt übernimmt. `CLKPCE` ist das Bit7, welches das **CLoCK Prescale Register** `CLKPR` des ATmega88 aktiviert, damit in ihm in den Bits 0 bis 3 ein Teiler eingetragen werden kann (siehe Seite 38 in der **Atmel-Druckschrift 8025D-AVR**). Um in dieses Register den Teiler (**0000** steht für Divisor 1, **0001** für Divisor 2, **0010** für Divisor 4, **0011** für Divisor 8 usw.) schreiben zu können, muss zunächst das Bit `CLKPCE` gesetzt und innerhalb von 4 Zyklen muss dann der Wert einschließlich

AVR-8-bit-Mikrocontroller

Arbeiten mit CodeVisionAVR C-Compiler

Teil 04 - Der Preprozessor

einer 0 für das Bit CLKPCE zurück geschrieben werden. Dass das Bit CLKPCE den Wert 10000000 (0x80) erhält, wird in den Header-Dateien `iomx8.h` und `typedefs.h` definiert:

In der Header-Datei `iomx8.h`

```
#define CLKPCE BIT7
```

und in der Header-Datei `typedefs.h`

```
#define BIT7 0x80
```

In der Quell-Datei `application.c` treten die folgenden "bedingten Kompilierungen" und Zuweisungs-Anweisungen auf. Die `#if`- und `#elif`-Abfragen bestimmen letztlich nur **eine** auf die eingestellte Frequenz abgestimmte **Einstellung**, die vom Compiler übernommen werden soll:

```
// Set selected CPU clock
#if (_MCU_CLOCK_FREQUENCY_ == 16000000)
    CLKPR = CLKPCE;    // Enable Clock Prescaler mit CLKPR = 0x80 (CLKPCE ist Bit7 = 0x80)
    CLKPR = 0;        // Setze Clock Prescaler, Division durch 1 (= 16 MHz bei 16MHz-Quarz)
#elif (_MCU_CLOCK_FREQUENCY_ == 8000000)
    CLKPR = CLKPCE;    // Enable Clock Prescaler mit CLKPR = 0x80 (CLKPCE ist Bit7 = 0x80)
    CLKPR = 1;        // Setze Clock Prescaler, Division durch 2 (= 8 MHz bei 16MHz-Quarz)
#elif (_MCU_CLOCK_FREQUENCY_ == 4000000)
    CLKPR = CLKPCE;    // Enable Clock Prescaler mit CLKPR = 0x80 (CLKPCE ist Bit7 = 0x80)
    CLKPR = 2;        // Setze Clock Prescaler, Division durch 4 (= 4 MHz bei 16MHz-Quarz)
#elif (_MCU_CLOCK_FREQUENCY_ == 2000000)
    CLKPR = CLKPCE;    // Enable Clock Prescaler mit CLKPR = 0x80 (CLKPCE ist Bit7 = 0x80)
    CLKPR = 3;        // Setze Clock Prescaler, Division durch 8 (= 2 MHz bei 16MHz-Quarz)
#elif (_MCU_CLOCK_FREQUENCY_ == 1000000)
    CLKPR = CLKPCE;    // Enable Clock Prescaler mit CLKPR = 0x80 (CLKPCE ist Bit7 = 0x80)
    CLKPR = 4;        // Setze Clock Prescaler, Division durch 16 (= 1 MHz bei 16MHz-Quarz)
#else
    // Andernfalls Fehleranzeige
    #error *** Invalid clock selected! ***
#endif
```

Wenn keine der Bedingungen erfüllt wird, wird die `#error`-Anweisung erreicht: Der Preprozessor bricht ab und es wird die angegebene Fehlermeldung ausgegeben.

Beispiel 4.5-02: Vermeidung mehrfacher Einfügungen durch `#include`-Anweisungen

Im Abschnitt 4.2 `#include`-Anweisungen traten in dem Beispiel keine wiederholten Einfügungen auf. Aber auf Grund verschachtelter `#include`-Anweisungen und Einbindung zahlreicher Quell-Programme ist es leicht möglich, dass dieselben Header-Dateien mehrfach auftreten. Nur drei Preprozessor-Anweisungen zur "bedingten Kompilierung" machen es möglich, dieses zu verhindern. In jeder Header-Datei werden jeweils am Anfang (beispielsweise in der Header-Datei `typedefs.h`) eine Abfrage und eine nachfolgende Definition über die Einbindung der Header-Datei als "logischer Schalter" eingebaut:

```
#ifndef __TYPEDEFS_H
#define __TYPEDEFS_H
```

und am Ende der "bedingten Kompilierung" steht:

```
#endif
```

Die erste Anweisung fragt ab, ob das Makro `__TYPEDEFS_H` noch **nicht definiert** ist (if not defined = `#ifndef`). Die Frage ist mit Sicherheit beim ersten Anlauf zu bejahen (`TRUE`; `__TYPEDEFS_H` ist noch nicht definiert) und der Code wird für die Kompilierung übernommen, da die Definition ja erst nach dem ersten Ansprung mit `#define` vollzogen wird. Das Makro `__TYPEDEFS_H` tritt nur an dieser Stelle auf und symbolisiert die Beispiels-Header-Datei `typedefs.h`. In den anderen Header-Dateien steht jeweils das entsprechende Symbol in Form eines Makros.

Wenn die Header-Datei erneut angesprochen wird, ist die Bedingung `#ifndef __TYPEDEFS_H` `FALSE` und alle Anweisungen bis zur `#endif`-Anweisung werden für die Kompilierung ignoriert, d.h. die gesamte Header-Datei wird nicht noch einmal eingebunden.

AVR-8-bit-Mikrocontroller

Arbeiten mit CodeVisionAVR C-Compiler

Teil 04 - Der Preprozessor

Jede "bedingte Kompilierung" muss mit einer `#endif`-Anweisung abgeschlossen werden.

Wie man sieht, werden keine Header-Dateien doppelt eingefügt, doppelte werden ausgespart. Die Annahme, dass auf Grund der Reihenfolge aller kaskadierten `#include`-Anweisungen auch **alle** Header-Dateien der **Reihe nach** eingefügt werden, ist also logischerweise für mehrfaches Auftreten von Header-Dateien nicht zutreffend.

4.6 Andere Preprozessor-Anweisungen

Weitere detaillierte Angaben über die Möglichkeiten und den Einsatz der hier erwähnten Preprozessor-Anweisungen können dem **CodeVisionAVR User Manual (Vers. 2.03.4; CVARMAN2.pdf) Seite 72 ff** und der **CodeVisionAVR Help-Datei (CVAVR.CHM)** entnommen werden.

Die `#pragma`-Anweisungen

Die `#pragma`-Anweisungen sind extrem Compiler-spezifisch und ermöglichen es, besondere Anweisungen und "logische Schalter" zu verwenden. Sie dienen der Programm-Optimierung und als Hilfe bei der Fehlersuche.

Die `#line`-Anweisung

Mit einer `#line`-Anweisung kann man die vom CVAVR vordefinierten Makros `__LINE__` und `__FILE__` modifizieren. Diese Makros gehören (wie das schon oben in **Beispiel 4.5-01** angewendete Makro `__MCU_CLOCK_FREQUENCY__`) in die Liste der speziell für diesen Compiler erzeugten vordefinierten Makros (Predefined Macros). Beispiel:

```
// Die Anweisung in der naechsten Zeile setzt das Makro __LINE__ auf 50
// (50 ist dann die laufende Zeilennummer des kompilierten Programms)
// und das Makro __FILE__ auf application.c
// (application.c ist in diesem Fall die gerade kompilierte Quell-Datei)
#line 50 "application.c"
```

Die `#error`-Anweisung

Die `#error`-Anweisung wurde schon in **Beispiel 4.5-01** angewendet. Wenn diese Anweisung vom Preprozessor erreicht wird, so beendet er seine Interpretations-Arbeit mit dem angegebenen Fehler-text.

Die `#warning`-Anweisung

Die `#warning`-Anweisung dient wie die `#error`-Anweisung zur Ausgabe eines vordefinierten Textes mit dem Unterschied, dass sie keinen Abbruch des Preprozessors verursacht

Die `#asm`- und `#endasm`-Anweisung

Die `#asm`- und `#endasm`-Anweisungen dienen zum Einfügen von Assembler-Coding. Mit `#asm` wird der Beginn des Assembler-Codings eingeleitet und mit `#endasm` abgeschlossen.