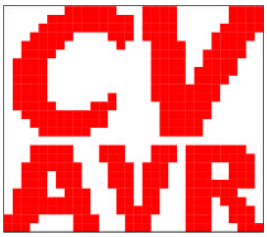
AVR-8-bit-Mikrocontroller
Gruppe 500 - CodeVisionAVR C-Compiler
Teil 503 - Der Preprozessor



Teil 501 - Einführung

1 Eine Einführung in **C**
- 1.1 Warum **C** ?
- 1.2 Wie entstand **C** ?
- 1.3 Der AVR-Mikrocontroller in einem eingebetteten System
- 1.4 Werkzeuge (siehe Gruppe 200)

Teil 502 - Aufbau eines C-Projektes

2 Was ist ein **C**-Projekt ?
- 2.1 Erzeugen eines **C**-Projektes
  - 2.1.1 Ein neues Projekt beginnen
  - 2.1.2 Ein **C**-Projekt generieren
- 2.2 Dateistruktur eines **C**-Projektes
- 2.3 Einbindung von AVR Studio in den CVAVR
- 2.4 AVR Studio Debugger
- 2.5 **C**-Compiler-Optionen

Teil 503 - Preprozessor

3 Preprozessor-Anweisungen
- 3.1 Struktur der **C**-Quell-Programme
- 3.2 `#include`-Anweisung
- 3.3 `#define`-Anweisung (Makro)
  - 3.3.1 Makros ohne Parameter
  - 3.3.2 Makros mit Parametern
- 3.4 `#undef`-Anweisung
- 3.5 `#if`-, `#ifdef`-, `#ifndef`-, `#else`- und `#endif`-Anweisungen
- 3.6 Andere Preprozessor-Anweisungen

Teil 504 - Syntax der C-Programmiersprache

4 Die Syntax der **C**-Programmiersprache
- 4.1 **C**-Quell-Programme
  - 4.1.1 Kommentare
  - 4.1.2 Deklarationen (Vereinbarungen)
  - 4.1.3 Die Funktion `main`
  - 4.1.4 Schlüsselwörter (Keywords) des CodeVisionAVR **C**-Compilers
- 4.2 Konstanten und Variablen
  - 4.2.1 Zahlensysteme
  - 4.2.2 Datentypen
  - 4.2.3 Konstanten
  - 4.2.4 Variablen
- 4.3 Operatoren
  - 4.3.1 Arithmetische Operatoren
  - 4.3.2 Relationale Operatoren
  - 4.3.3 Logische und bitweise wirkende Operatoren
  - 4.3.4 Andere Operatoren und Shortcuts
- 4.4 Komplexe Objekte in **C**
  - 4.4.1 Funktionen
  - 4.4.2 Funktions-Prototypen
  - 4.4.3 Pointers und Arrays
    - 4.4.3.1 Pointers

4.4.3.1.1 Pointers in Verbindung mit **flash** und **eeprom**
4.4.3.1.2 Pointers in Verbindung mit **typedef**
4.4.3.2 Arrays
4.4.3.2.1 Ein-dimensionale Arrays
4.4.3.2.2 Zwei-dimensionale Arrays
4.4.3.2.3 Drei-dimensionale Arrays
4.4.3.3 Benutzen der Array-Namen als Pointers
4.4.3.4 Arrays von Pointers
4.4.3.5 Pointers auf Funktionen (Funktionszeiger)
4.4.3.6 Funktionen in Verbindung mit **typedef**
4.4.4 Strukturen und Unionen
4.4.4.1 Strukturen
4.4.4.2 Unionen
4.4.5 Komplexe Typen (eine Zusammenfassung)
4.5 Steuerung des Programmablaufs
4.5.1 Anweisungsblöcke **{ . . . }**
4.5.2 Die Anweisung **if**
4.5.3 Die Anweisung **if** in Verbindung mit **else**
4.5.4 Die Fallunterscheidung **switch**
4.5.5 Die Schleife **for**
4.5.6 Die Schleife **while**
4.5.7 Die Schleife **do** in Verbindung mit **while**
4.6 Arbeiten mit den Ein-/Ausgabe-Ports

Teil 505 - Modularer Aufbau der AVR-C-Projekte

5 Modularer Aufbau
5.1 Das Konzept
5.2 Nomenklatur
5.3 Die speziellen Header-Dateien
5.3.1 Die spezielle Header-Datei **typedefs.h** (Typ- und Bit-Definitionen)
5.3.2 Die spezielle Header-Datei **iomx.h** (Definitionen aller Register-Bits)
5.3.3 Die spezielle Header-Datei **macros.h** (Definitionen von Makros)
5.3.4 Die spezielle Header-Datei **switches.h** (Definitionen von Schaltern)
5.4 Die AVR-C-Module
5.5 Anwendung der angepassten AVR-C-Module
5.5.1 Das AVR-C-Modul **application** (Anwendungs-Steuerung)
5.5.2 Das AVR-C-Modul **lcd_2wire** (Ausgabe auf LCD-20x4)
5.5.3 Das AVR-C-Modul **num_conversion** (Typ-Konvertierung nach ASCII)
5.5.4 Das AVR-C-Modul **adc_ref_1_1** (ADC mit interner Referenz 1,1 V)
5.5.5 Das AVR-C-Modul **rc5_decoder** (RC5-IR-Fernsteuerung-Dekoder)
5.5.6 Das AVR-C-Modul **rc5_encoder** (RC5-IR-Fernsteuerung-Enkoder)
5.5.7 Das AVR-C-Modul **usart** (USART-Steuerung)
5.5.8 Das AVR-C-Modul **twi_master** (IR2- bzw. TWI-Steuerung)
5.5.9 Das AVR-C-Modul **timer0_pwm** (TIMER0-Steuerung)

Teil 506 - Anhang

6 Anhang
6.1 Begriffe und Definitionen
6.2 Bibliothek

AVR-8-bit-Mikrocontroller
Gruppe 500 - CodeVisionAVR C-Compiler
Teil 503 - 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.

Bild-, Beispiel-, Form- und Tabellen-Nummern sind nach folgendem Schema aufgebaut, damit bei Einfügungen/Löschungen nicht alle Nummern wieder geändert werden müssen (hier bunt dargestellt):

Darstellungsart Abschnitt-LfdNummer: Beschreibung	**allgemeines Schema**
• **Bild 5.1.4-02: Daten-Adress-Raum**	Benummerung eines Bildes
• **Beispiel 5.1.4-03: EEPROM-Speicherung**	Benummerung eines Beispiels
• **Form 5.1.3-01: Die `main`-Funktion**	Benummerung einer Formdarstellung
• **Tabelle 5.1.4-01: Schlüsselwörter vom CVAVR**	Benummerung einer Tabelle

Gravierende Änderungen gegenüber der Vorversion

1.
Völlig neue Strukturierung in **Gruppen** und **Teile**, um das Tutorial umfassend ordnen zu können. Die **Abschnitte** in den **Teilen** sind weitgehend erhalten geblieben.

Gruppenbezeichnung	Kurzbezeichnung
Gruppe 100: Technologie der AVR-8-Bit-Mikrocontroller	**Technologie**
Gruppe 200: Einsetzen von AVR-Tools	**Tools**
Gruppe 300: Arbeiten mit AVR-Assembler **3xx_Programm_yyyyy**	**ASM-Programmierung** **ASM-Programm-Beispiel**
Gruppe 400: AVR-ASM-Projekte **4xx_Projekt_yyyyy**	**ASM-Projekte** **ASM-Projekt-Bezeichnung**
Gruppe 500: CodeVisionAVR C-Compiler **5xx_Programm_yyyyy**	**C-Programmierung** **C-Programm-Beispiel**
Gruppe 600: AVR-C-Projekte **6xx_Projekt_yyyyy**	**C-Projekte** **C-Projekt-Bezeichnung**

xx steht für die laufende Nummer innerhalb des **Teils**, in dem das Programm bzw. das Projekt erscheint und **yyyyy** steht für die Programm- bzw. Projekt-Kurz-Bezeichnung.

2.
Notwendige Änderungen auf Grund Neuinstallation von **Windows 7**.

3.
Windows 7 machte eine Installation von **CodeVisionAVR V2.60** als Vollversion notwendig. Daraus leiten sich auch viele Änderungen im Detail für die C-Programmierung (**Gruppe 500**) ab.

4.
Neu-Installation von **AVR Studio Vers. 4.19** unter **Windows 7**

5.
Zur Demonstration des Tools **AVR Studio** ist in **Gruppe 200** eine Trennung in **Teil 205 - Assembler und AVR Studio** und **Teil 206 - C-Compiler und AVR Studio** vorgenommen worden.

6.
ASM- und **C-Projekte** werden jeweils in eigenen Gruppen gesammelt (**Gruppe 400** für Assembler- und **Gruppe 600** für C-Projekte).

3 Preprozessor-Anweisungen

3.1 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 504 - 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 505 - Modularer Aufbau der AVR-C-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      Warungsmeldung
#message      Allgemeine Meldung
#asm
#endasm
```

3.2 `#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 (vergleiche auch **Bild 3.2-01**). 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 die Header-Dateien (Kopf-Dateien), mit einschließen (to **include**) soll. **Header-Dateien** erkennt man an der Endung **`.h`**

Beispiel 3.2-01: Einfügung in die Datei `main.c` im Projekt AVR_PB_LED

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

Beispiel 3.2-02: Einfügungen in die 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 Verzoegerung
#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 Vers. 2.60** im Ordner **C:\cvavr2\inc** angelegt). Wird die Header-Datei jedoch in Anführungszeichen (z.B. **`"application.h"`**) aufgerufen, so handelt es sich um eine selbst erstellte **spezielle 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. Zum Beispiel sind alle im AVR-C-Projekt **601_AVR_PB_LED** angesprochenen Dateien und die in ihnen enthaltenen **`#include`**-Anweisungen im **Bild 3.2-01** 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, zeigt die nachfolgende Aufstellung.

Eine ausführlichere Beschreibung der selbst erstellten Header-Dateien (und der selbst kreierten Module) ist im **Teil 505 - Modularer Aufbau der AVR-C-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:\cvavr2\bin** aufgerufen werden.

Anmerkung: Die globalen Haeder-Dateien stehen im Ordner **C:\cvavr2\inc** in der Form ***.h** und sind ein Pool von Funktions-Prototypen für ähnliche Aufgaben.

Beispiel 3.2-03: `#include <delay.h>  // Funktions-Prototypen zur Verzoegerung`

Die Haeder-Datei `delay.h` enthält die beiden Funktions-Prototypen, um programmtechnische Abläufe entweder um n **Mikro**sekunden oder n **Milli**sekunden zu verzögern:

```
void delay_us(unsigned int n);
void delay_ms(unsigned int n);
```

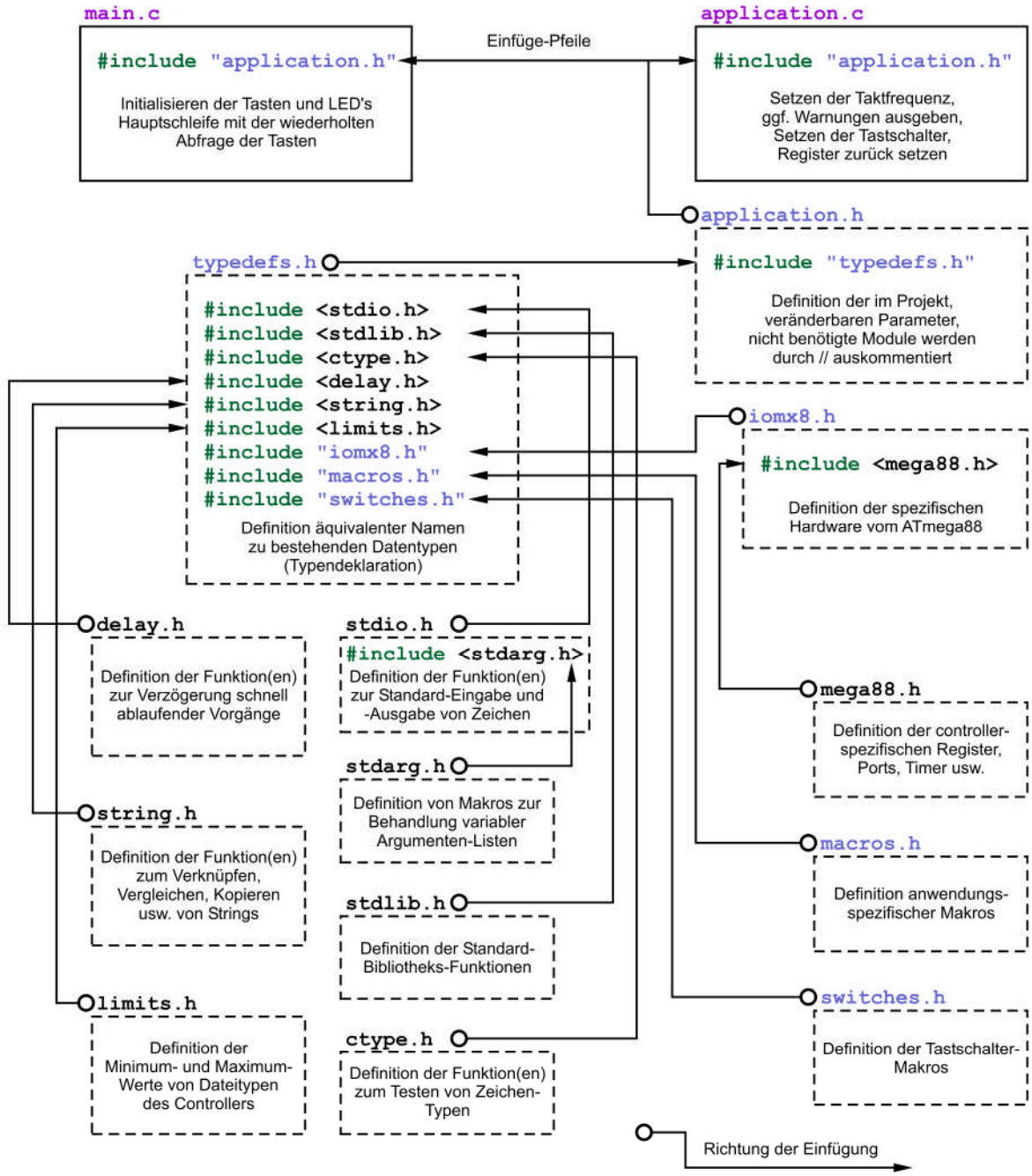


Bild 3.2-01: Struktur des Beispiel-Projektes AVR_PB_LED (Bildvergrößerung)

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

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

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:	in der Datei:	wird das Coding aus folgender Datei eingesetzt:
#include <stdarg.h>	main.c	C:\cvavr_atm18_eval\inc\stdarg.h

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

Der Preprozessor durchläuft diese Codings von **stdlib.h** bis **limits.h** und trifft auf die **#include**-Anweisung der selbst erstellten speziellen 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:	in der Datei:	wird das Coding aus folgender Datei eingesetzt:
#include <mega88.h>	main.c	C:\cvavr_atm18_eval\inc\mega88.h

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

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

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

3.3 #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_Name**$_{Space}$**Replacement_Text**$_{CR/LF}$

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.

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 **`Identifier_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.

3.3.1 Makros ohne Parameter

Beispiel 3.3.1-01: Einfügung in die Datei `main.c` im Projekt AVR_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**:

```
//---------------------------------------------------------------------------
// 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 speziellen 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 | 00000100 = 00000100`	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 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.

3.3.2 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.

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 mehren 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 **4.3.4 Andere Operatoren und Shortcuts** im **Teil 504**), 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 3.3.2-01: Einfügung in die Datei `main.c` im Projekt AVR_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)
//-------------------------------------------------------------------------------
#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**):

```
DDRB = DDRB & 11110111 = 00000000        alle Bits sind als Eingänge konfiguriert
DDRB = DDRB & 11101111 = 00000000        alle Bits sind als Eingänge konfiguriert
DDRB = DDRB & 11011111 = 00000000        alle Bits sind als Eingänge konfiguriert
PINB = PINB | 00001000 = 00001000        Bit3 ist auf Eingabe aktiviert
PINB = PINB | 00010000 = 00011000        Bit3 und Bit4 sind auf Eingabe aktiviert
PINB = PINB | 00100000 = 00111000        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.

3.4 #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
...
...
#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
```

3.5 #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 505 Modularer Aufbau der AVR-C-Projekte** wird von dieser Methode rege Gebrauch gemacht, um einmal erzeugte ausführliche Header-Dateien (Dateien mit möglichst umfassender Definition aller gebräuchlichen Ports usw.) einfach durch Setzten 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 3.5-01: Einfügung in die Datei `main.c` im Projekt AVR_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 **CL**oc**K** **P**rescale **R**egister CLKPR des ATmega88 aktiviert, damit in ihm in den Bits 0 bis 3 ein Teiler eingetragen werden kann (siehe Seite 36ff in der **Atmel-Druckschrift** Rev. 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 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 3.5-02: Vermeidung mehrfacher Einfügungen durch `#include`-Anweisungen

Im Abschnitt **3.2 `#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.

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.

3.6 Andere Preprozessor-Anweisungen

Weitere detaillierte Angaben über die Möglichkeiten und den Einsatz der hier erwähnten Preprozessor-Anweisungen können dem **CodeVisionAVR User Manual** (Vers. 2.60) **Seite 100 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 3.5-01** 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 3.5-01** angewendet. Wenn diese Anweisung vom Preprozessor erreicht wird, so beendet er seine Interpretations-Arbeit mit dem angegebenen Fehlertext.

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.