

AVR-8-bit-Mikrocontroller

Arbeiten mit CodeVisionAVR C-Compiler

Teil 07 - Anhang

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 07 - Anhang

- 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` (I2C- 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 07 - Anhang

Vorbemerkung

Nichts ist vollkommen - und nichts ist endgültig! So auch nicht dieses Tutorial! Deshalb bitte immer erst nach der neuesten Version (neuestes Datum) schauen. Vielleicht gibt es wieder etwas Neues oder eine Fehlerbereinigung oder eine etwas bessere Erklärung. Wie im Forum gilt auch hier: 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

AVR-8-bit-Mikrocontroller
Arbeiten mit CodeVisionAVR C-Compiler
Teil 07 - Anhang

7 Anhang

7.1 Begriffe und Definitionen in der Programmiersprache C

Vorab: Es gibt eine sehr umfangreiche Hilfefunktion in englischer Sprache für den CVAVR, die mit **CVAVR.chm** aufgerufen werden kann. Darüber hinaus ist das **User Manual** des CVAVR eine sehr ausführliche Beschreibung des Compilers - aber **keine** Einführung in die **C-Programmiersprache**.

Dieser Abschnitt über Begriffe und Definitionen dient zur Orientierung und als "Nachschlagewerk". Die Begriffe werden in den betreffenden Abschnitten im **Teil 04 Der Preprozessor** und im **Teil05 Syntax der C-Programme** ausführlicher erläutert.

Argument	Das Argument ist der Wert , der bei einem Funktionsaufruf der Funktion übergeben wird.
Anweisung	Oder Statement . Das ist der Befehl, d.h. der kleinste ausführbare Einzelschritt, in einem Computer-Programm.
Anweisungsblock	Die blockweise Zusammenfassung von mehreren Anweisungen , eingeschlossen in geschweifte Klammern, bildet einen Anweisungsblock.
Array	Ein Array ist eine Kette von lückenlos aufeinander folgenden Objekten (Elementen) gleichen Typs. Arrays können ein- und mehrdimensional sein. Beispiel eines eindimensionalen Arrays: <pre>// 1-dimensionales Array für 10 ASCII-Zeichen 0 bis 9 unsigned char ziffer[10]={0,1,2,3,4,5,6,7,8,9}; // der Index in den eckigen Klammern beginnt bei 0 // und endet bei dem angegebenen Wert minus 1 // daraus folgt: ziffer[0]='0', ziffer[1]='1' usw. // bis ziffer[9]='9'</pre>
Bitfelder	Bitfelder können nur innerhalb von Strukturen und Unionen definiert werden. Sie werden häufig benutzt, wenn der Speicherplatz eng wird.
Definition/Deklaration	In C müssen alle Variablen vereinbart werden, bevor sie benutzt werden. Bei den Vereinbarungen unterscheidet man Definitionen , die die Objekte erzeugen und Deklarationen , die die Eigenschaften von Objekten nur festlegen.
Definition	Beschreibung, Vereinbarung, bei gleichzeitigem Anlegen eines oder mehrerer Objekte und Belegung von Speicherplatz. Eine Definition besteht aus einem Typ und einer Liste von Variablen .
Deklaration	(Vereinbarung) Die Deklaration legt den Namen , den Typ und die Speicherklasse einer Variablen fest.
Element	Das Element ist die einzelne Zelle eines Arrays . Lückenlos aufeinander folgende Elemente gleichen Typs bilden das Array .
Enumeratoren	Mit dem Schlüsselwort enum kann man eine Liste von Namen mit sequentiell aufsteigenden Integer-Konstanten erzeugen. Die so erzeugten Namen nennt man Enumeratoren.

AVR-8-bit-Mikrocontroller

Arbeiten mit CodeVisionAVR C-Compiler

Teil 07 - Anhang

Beispiele zur Erzeugung von Enumeratoren:

```
enum {null, eins, zwei, drei, vier, fuenf};
```

Die Enumeratoren null bis fuenf sind jetzt der Reihe nach mit den konstanten Integer-Werten 0 bis 5 belegt worden und können anstelle dieser Werte eingesetzt werden.

```
enum {red_led_on = 1, green_led_on, both_leds_on};  
#define leds PORTA
```

```
/* wo in einer der folgenden Anweisung "leds" auftaucht,  
wird es durch "PORTA" ersetzt.  
*/
```

```
PORTA = 0x1;           //bringt die rote LED zum Leuchten  
leds = red_led_on;    //ist eine identische Anweisung
```

Der Wert von red_led_on wird auf 1, der von green_led_on auf 2 und der von both_leds_on auf 3 gesetzt.

Funktion

Die Steuerung des Programmablaufs wird allein durch **Funktionen** vorgenommen, d.h. nur innerhalb der Funktionen sind Algorithmen zugelassen. Die **Funktionen** machen mit Hilfe von überschaubaren, in sich logisch abgeschlossenen Handlungen die C-Anwendungen generell übersichtlicher, leichter veränderbar und portierbar. Im Allgemeinen bestehen C-Programme aus vielen kleinen und nicht aus wenigen großen **Funktionen**.

Jede Funktion hat folgende allgemeine Form:

```
Rückgabety Name (Parameter-Liste) // Deklaration  
{ // Beginn  
    Vereinbarungen; // des  
    Anweisung1; // Funktionskörpers  
    Anweisung2;  
    usw.  
    return (Rückgabewert); // und sein  
} // Ende
```

Funktionskörper

Die Steuerung des Programmablaufs wird innerhalb von **Funktionskörpern** wahrgenommen. Ein **Funktionskörper** wird von geschweiften Klammern eingeschlossen

Funktions-Prototypen

Der **Prototyp** einer **Funktion** weist den Compiler vor dem eigentlichen Aufruf der **Funktion** an, welche Aufrufargument-Typen und/oder welcher Rückgabety der **Funktion** akzeptiert werden sollen. Die Nennung der **Variablen** entfällt. Mit der Aufzählung von **Funktions-Prototypen** zu Beginn eines Quell-Codes kann die Reihenfolge des Auftretens der **Funktionen** beliebig sein.

Beispiel zur Anwendung eines Funktions-Prototyps:

```
int funktion_a(int, int); // Funktions-Prototyp  
// fuer funktion_a  
  
void funktion_b(void); // Deklaration funktion_b  
  
int var1, var2, var3; // Deklaration der Variablen  
  
void main(void) // Deklaration main
```

AVR-8-bit-Mikrocontroller

Arbeiten mit CodeVisionAVR C-Compiler

Teil 07 - Anhang

```

{
    var3 = funktion_a(var1, var2); // Aufruf funktion_a
    funktion_b; // Aufruf funktion_b
}

int funktion_a(int x, int y); // Deklaration funktion_a

```

Header

Kopf-Datei <abc.h> bzw. "**xyz.h**" mit **Deklarationen** als Inhalt.

Kommentar

Kommentare sind Erklärungen innerhalb eines Quellprogramms, um den Programmablauf zu kommentieren. Mit

```

/* beliebiger Text
über mehrere Zeilen */

```

können ganze Absätze in das Coding eingefügt werden, die jeweils mit **/*** beginnen und mit ***/** enden. Alle Zeichen dazwischen werden vom Compiler ignoriert. Sollen nur einzelne Befehls-Zeilen kommentiert werden, so wird mit **//** der erläuternde Text eingeleitet. Dieser endet stets mit dem Zeilenende.

Beispiele für verschiedene Kommentare:

```

01 // Module..: REACTION_LED, MAIN.C
02 // Version.: 1.0
03 // Compiler: CodeVisionAVR
04 // Chip....: ATmega88
05 // Date....: May 2008
06 // Author..: Udo Juerss
07 //-----
08
09 /*
10 Testboard wiring for REACTION_LED application
11
12 COMPONENT                      Testboard-PIN
13 -----
14 *** LEDs ***
15 LED1 (left most)                PD2
16 LED2                            PC5
17 LED3                            PC4
18 LED4                            PC3
19 LED5                            PC2
20 LED6                            PC1
21 LED7 (right most)              PC0
22
23 *** PUSHBUTTONS ***
24 S2 (middle)                      PBO
25 */
26 //-----
27 #include "typedefs.h" //h-Datei fuer eigene Datentypen
28 #include "led.h" //h-Datei zur Definition der LEDs
29 #include "stop_watch.h" //h-Datei zur Definition des Stop
30 #include "switches.h" //h-Datei zur Def. der Tasten
31 //-----

```

In den Zeilen 01 bis 07 und in den Zeilen 26 bis 31 werden Zeilen-Kommentare eingefügt. Die Zeilen 09 bis 25 bilden einen Block-Kommentar zur Beschreibung des Programms.

Konstanten

Einige **Konstanten** sind bereits Teile des kompilierten Programms selbst andere werden per **Deklaration** durch das **reservierte Wort `const`**, der Festlegung des **Typs** und der Vergabe eines **Namens** vereinbart.

Beispiel:

```

const float pi = 3.141592F; //F steht für Floating Point

```

AVR-8-bit-Mikrocontroller

Arbeiten mit CodeVisionAVR C-Compiler

Teil 07 - Anhang

Makros Ein Name, dem per **#define**-Anweisung ein Text-String (Makro-Rumpf) zugeordnet ist, bildet in der C-Programmierung ein **Makro**. Nach Konventionen werden Makro-Namen, die **Konstanten** repräsentieren, groß geschrieben. Im AVR-Projekt werden **Makros** generell groß geschrieben.

Jedes Vorkommen eines Makro-Namens, also bei seinen Aufrufen im Programm, wird vom **Preprozessor** durch den Makro-Rumpf ersetzt, in die Eingabe für den **Preprozessor** zurückgestellt und ggf. erneut bearbeitet.

Mitglied Oder **Member**, das sind die Komponenten in **Strukturen** und **Unionen**.

Namen Oder **Identifizier** sind die eindeutigen Bezeichnungen (Worte) für **Variable**, **Konstante** oder **Funktionen**. Ein **Name** besteht aus einer Folge von Buchstaben und Ziffern. Das erste Zeichen muss ein Buchstabe sein. Das Zeichen **_** (Unterstrich) zählt zu den Buchstaben. Große und kleine Buchstaben werden unterschieden. **Schlüsselwörter** werden stets mit kleinen Buchstaben gekennzeichnet.

Als Vereinbarung im AVR-Projekt gilt:

- Alle **Variablen** und **Funktionen** werden generell kleingeschrieben,
- **Konstante**, **Makros** und **Namen** für **Datentypen** werden großgeschrieben,
- zusammengesetzte **Namen** werden mit einem Unterstrich **_** verbunden.

Objekt Sämtliche **Variablen** und adressbehafteten **Konstanten** jeglichen **Typs**, **Funktionen**, **Arrays**, **Strukturen**, **Unionen**, **Bitfelder** und **Enumerationen** sind **Objekte** der Sprache **C**. **Funktionen** sind **Code-Objekte** und die übrigen sind **Daten-Objekte**. Jedem Objekt wird bei seiner Definition Speicherplatz an einer unveränderbaren Position im Speicher zugewiesen.

Operatoren Symbole aus ein oder zwei Zeichen, die dem Compiler den Typ der auszuführenden Operation anzeigen. Eine Anweisung (Statement), in der ein Operator ein oder zwei Operanden verbindet, resultiert entweder in einer Aussage **TRUE** oder **FALSE** oder in einem numerischen Wert.

Tabelle: Arithmetische Operatoren

Vorrang	Operator	Operation	Beispiele
1	*	Multiplikation	y = a * b
1	/	Division	y = a / b
2	%	Restwert-Division (Modulo)	rest_von = a % b
3	+	Addition	y = x + 3
3	-	Subtraktion oder Negation	y = x - z

Tabelle: Relationale Operatoren

Operator	Operation	Beispiele	Annahme	Ergebnis
==	... ist gleich ...	(x == y)	x = 3 y = 5	FALSE
!=	... ist nicht gleich ...	(x != y)		TRUE
<	... ist kleiner als ...	(x < y)		TRUE
<=	... ist kleiner oder gleich ...	(x <= y)		TRUE
>	... ist größer als ...	(x > y)		FALSE
>=	... ist größer oder gleich ...	(x >= y)		FALSE

AVR-8-bit-Mikrocontroller

Arbeiten mit CodeVisionAVR C-Compiler

Teil 07 - Anhang

Tabelle: Logisch und bitweise wirkende Operatoren

Operator	Operation	Beispiele	Annahme	Ergebnis
~	Einerkomplement bitweise	<code>x = ~y;</code>	unsigned char y = 201 = 0xC9 = 0b11001001	<code>x = 0b00110110</code>
<<	Links-Shift bitweise	<code>x = y << 3;</code>		<code>x = 0b01001000</code>
>>	Rechts-Shift bitweise	<code>x = y >> 4;</code>		<code>x = 0b00001100</code>
&	UND bitweise	<code>x = y & 0x3F;</code>		<code>x = 0b00001001</code>
^	Exklusives ODER bitweise	<code>x = y ^ 1;</code>		<code>x = 0b11001000</code>
	ODER bitweise	<code>x = y 0x10;</code>		<code>x = 0b11011001</code>
&	UND bitweise	<code>(x & y)</code>	x = 0b00001001 y = 0b00000110	<code>0b00000000 = FALSE</code>
&&	UND logisch	<code>(x && y)</code>		<code>TRUE && TRUE = TRUE</code>
^	Exklusiv ODER bitweise	<code>(x ^ y)</code>	x = 0b00001001 y = 0b00001001	<code>0b00000000 = FALSE</code>
	ODER bitweise	<code>(x y)</code>		<code>0b00001001 = TRUE</code>
	ODER logisch	<code>(x y)</code>		<code>TRUE TRUE = TRUE</code>

Tabelle: Alle Operatoren nach Vorrang (z.B. Punktrechnung geht vor Strichrechnung)

Vorrang (Level)	Gruppierung von/nach	Operator	Anzahl Operanden	Operation (Wirkungsweise)
1	links nach rechts	()	beliebig	Mehrfachbedeutung, Funktions-Operator
1	links nach rechts	[]	1 (monadisch)	Array-Element
1	links nach rechts	->	2 (dyadisch)	Zugriffsoperator auf ein Struktur-Mitglied
1	links nach rechts	.	2 (dyadisch)	Punkt-Operator für Struktur-Mitglieder
2	rechts nach links	!	1 (monadisch)	Logisches NICHT liefert TRUE oder FALSE
2	rechts nach links	~	1 (monadisch)	Einerkomplement (bitweise)
2	rechts nach links	-	1 (monadisch)	Änderung des Vorzeichens
2	rechts nach links	*	1 (monadisch)	Verweis-Operator (Indirection-Operator)
2	rechts nach links	&	1 (monadisch)	Adress-Operator (Address Operator)
2	rechts nach links	++	1 (monadisch)	Prä- oder Post-Inkrement
2	rechts nach links	--	1 (monadisch)	Prä- oder Post-Dekrement
2	rechts nach links	(Type)	1 (monadisch)	Temporäre Typzuweisung
2	rechts nach links	sizeof	1 (monadisch)	Anzahl Bytes eines Operanden
3	links nach rechts	*	2 (dyadisch)	Multiplikation
3	links nach rechts	/	2 (dyadisch)	Division
3	links nach rechts	%	2 (dyadisch)	Restwert-Division (Modulo; nur mit int)
4	links nach rechts	+	2 (dyadisch)	Addition
4	links nach rechts	-	2 (dyadisch)	Subtraktion
5	links nach rechts	<<	2 (dyadisch)	Links-Shift (bitweise)
5	links nach rechts	>>	2 (dyadisch)	Rechts-Shift (bitweise)
6	links nach rechts	<	2 (dyadisch)	Vergleichsoperator: Kleiner als
6	links nach rechts	<=	2 (dyadisch)	Vergleichsoperator: Kleiner oder gleich als
6	links nach rechts	>=	2 (dyadisch)	Vergleichsoperator: Größer oder gleich als
6	links nach rechts	>	2 (dyadisch)	Vergleichsoperator: Größer als
7	links nach rechts	==	2 (dyadisch)	Vergleichsoperator: Ist identisch mit
7	links nach rechts	!=	2 (dyadisch)	Vergleichsoperator: Ist nicht identisch mit
8	links nach rechts	&	2 (dyadisch)	UND (bitweise)
9	links nach rechts	^	2 (dyadisch)	Exklusives ODER (bitweise)
10	links nach rechts		2 (dyadisch)	ODER (bitweise)
11	links nach rechts	&&	2 (dyadisch)	Logisches UND liefert TRUE oder FALSE
12	links nach rechts		2 (dyadisch)	Logisches ODER liefert TRUE oder FALSE
13	rechts nach links	?:	3 (triadisch)	Bedingungsoperator: ähnlich if()-else
14	rechts nach links	=	2 (dyadisch)	Zuweisungsoperator

AVR-8-bit-Mikrocontroller

Arbeiten mit CodeVisionAVR C-Compiler

Teil 07 - Anhang

14	rechts nach links	+=	2 (dyadisch)	Zusammengesetzte Zuweisung mit +
14	rechts nach links	-=	2 (dyadisch)	Zusammengesetzte Zuweisung mit -
14	rechts nach links	*=	2 (dyadisch)	Zusammengesetzte Zuweisung mit *
14	rechts nach links	/=	2 (dyadisch)	Zusammengesetzte Zuweisung mit /
14	rechts nach links	%=	2 (dyadisch)	Zusammengesetzte Zuweisung mit %
14	rechts nach links	&=	2 (dyadisch)	Zusammengesetzte Zuweisung mit &
14	rechts nach links	^=	2 (dyadisch)	Zusammengesetzte Zuweisung mit ^
14	rechts nach links	 =	2 (dyadisch)	Zusammengesetzte Zuweisung mit
14	rechts nach links	<<=	2 (dyadisch)	Zusammengesetzte Zuweisung mit <<
14	rechts nach links	>>=	2 (dyadisch)	Zusammengesetzte Zuweisung mit >>
15	links nach rechts	,	beliebig	Aufzählung, Kettenbildung

Parameter Ein **Parameter** ist die **Variable**, die den Wert des **Argumentes** in der aufgerufenen **Funktion** repräsentiert.

Parameter-Liste Die Angabe eines oder mehrerer **Parameter**, die in runden Klammern eingeschlossen dem **Funktionsnamen** folgen, nennt man **Parameter-Liste**.

Pointer Zeiger, Adresse (indirekte Adressierung).

pragma Die Preprozessor-Anweisungen **#pragma** (Kurzform für **Pragmatic Information**) ermöglicht es, besondere Anweisungen und logische Schalter vor der eigentlichen Kompilierung zu verwenden. Sie sind extrem Compiler-abhängig. Eine **#pragma**-Anweisung übermittelt gewöhnlich unwesentliche Information, häufig in der Absicht, den Compiler bei der Programm-Optimierung zu helfen.

Schlüsselwörter Die folgenden **Namen** haben für den **C-Compiler** besondere Bedeutungen und dürfen deshalb nicht als Worte für Variable usw. benutzt werden (Tabelle von **CVAVR**):

break	bit	bool	_Bool	case
char	const	continue	default	defined
do	double	eeprom	__eeprom	else
enum	extern	flash	__flash	float
for	goto	if	inline	int
interrupt	__interrupt	long	register	return
short	signed	sizeof	sfrb	sfrw
static	struct	switch	typedef	union
unsigned	void	volatile	while	

Speicherklasse **Speicherklassen** legen fest, wie die **Variable** behandelt werden soll, insbesondere in welchem **Geltungsbereich** sie angelegt werden soll.

Strukturen Die Strukturierung ist eine Methode, um ein einheitliches Daten-Objekt zu erzeugen, das aus mehreren **Variablen** besteht. Im Unterschied zu **Arrays** können **Strukturen** als Ganzes zugewiesen werden und können somit als Ganzes an Funktionen übergeben und zurückerhalten werden.

Typ Der (Daten-)**Typ** einer **Variablen** legt die Struktur der **Variablen** fest, z.B. ob sie ganzzahlige Werte oder Gleitkomma-Werte annehmen. Die folgende Tabelle zeigt die verschiedenen **Typen** und ihre zugewiesene Größe bei **Variablen** (für **CVAVR**):

AVR-8-bit-Mikrocontroller

Arbeiten mit CodeVisionAVR C-Compiler

Teil 07 - Anhang

Tabelle: Die Größen verschiedener Typen

Schlüsselwörter	Typ		Größe in Bits	Werte-Bereich	
	AVR-Synonyme			von	bis
bit	-		1	0	1
bool, _Bool	-		8	0	ungleich 0
char	S8, S08, tS08		8	-128	+127
unsigned char	U8, U08, tU08		8	0	255
signed char	S8, S08, tS08		8	-128	+127
int	S16, tS16		16	-32768	+32767
short int	S16, tS16		16	-32768	+32767
unsigned int	U16, tU16		16	0	65535
signed int	S16, tS16		16	-32768	+32767
long int	S32, tS32		32	-2147483648	+2147483647
unsigned long int	U32, tU32		32	0	4294967295
signed long int	S32, tS32		32	-2147483648	+2147483647
float	-		32	±1.175e-38	±3.402e38
double	-		32	±1.175e-38	±3.402e38

Typ-Deklarationen

In **C** werden sehr vielfältige **Typ-Deklarationen** (komplexe Typen) benutzt, die sehr genau unterschieden werden müssen. Um Typen korrekt zu erkennen, muss man vom **Namen** ausgehen und links und rechts davon befindliche **Operatoren** gemäß ihrem Vorrang interpretieren. Dabei muss man von Innen nach Außen vorgehen und Paare mit runden Klammern untersuchen, ob sie lediglich einen Vorrang erzwingen sollen oder aber eine Funktion anzeigen. In den folgenden Beispielen steht der Typ **int** stellvertretend für ein beliebiges Typen-Schlüsselwort:

```

int i; // i ist eine Integer-Variable.

int *p; // p ist ein Pointer auf eine
        // Integer-Variable.

int **pp; // pp ist ein Pointer auf den
          // Pointer p, der auf eine
          // Integer-Variable zeigt.

int ***ppp; // ppp ist ein Pointer, der auf
            // den Pointer pp zeigt, der auf
            // den Pointer p zeigt, der auf
            // eine Integer-Variable zeigt.

int *a[3]; // a ist ein Array von 3 Pointers
           // auf Integer.

int * f(); // f ist eine Funktion, die einen
           // Pointer auf Integer als
           // Ergebnis liefert.

int (*p)(); // p ist ein Pointer auf eine
            // Funktion mit Integer-Ergebnis.

int (*p)(int x); // p ist ein Pointer auf eine
                // Funktion mit Integer-Ergebnis,
                // die als Parameter einen
                // Integer-Wert uebernimmt.

int (*p)(int[]); // p ist ein Pointer auf eine
                 // Funktion mit Integer-Ergebnis,
                 // die als Parameter ein Integer-
                 // Array unbestimmter Groesse
                 // uebernimmt.

int (*p)(int *a[]); // p ist ein Pointer auf eine

```

AVR-8-bit-Mikrocontroller

Arbeiten mit CodeVisionAVR C-Compiler

Teil 07 - Anhang

```
// Funktion mit Integer-Ergebnis,  
// die als Parameter ein Array a  
// unbestimmter Groesse auf  
// Integer uebernimmt.  
  
int *f(int *a[3]); // f ist eine Funktion, die einen  
// Pointer auf Integer als  
// Ergebnis liefert und als  
// Parameter ein Array a mit  
// 3 Pointers auf Integer  
// uebernimmt.
```

typedef	Schlüsselwort, um Synonyme (unterschiedliche Namen gleicher Bedeutung) für die Typen-Namen zu bilden (vergl. 2. Spalte in der Typ-Tabelle).
Unionen	Unionen sind mit den Strukturen bis auf einem Punkt identisch: Alle Mitglieder einer Union teilen sich ein und denselben Speicherplatz.
Variable	Eine Variable ist eine (im Gegensatz zur Konstante) Größe, die wie in der Algebra veränderbar ist.
Variable, Definition	Die Variable wird definiert durch ein reserviertes Wort, welches den Typ und die Größe festlegt, durch den für sie eindeutig festgelegten Namen (Identifier) und der Zuweisung von Speicherplatz.
Variable, Deklaration	Die Variable wird deklariert durch ein reserviertes Wort, welches den Typ und die Größe festlegt, und durch den für sie eindeutig festgelegten Namen (Identifier) .
	Beispiele: <pre>unsigned char text_anzeige; int lfd_nr, schuh_groesse; long int population;</pre>
Variablen, Lokale	Variablen , die lediglich in Funktionen deklariert werden, sind privat, intern oder lokal , d.h. keine andere Funktion hat auf sie einen direkten Zugriff. Jede lokale Variable in einer Funktion wird nur erzeugt, wenn die Funktion aufgerufen wird und sie verschwindet wieder, wenn die Funktion verlassen wird. Ausnahme bei der Speicherklasse static
Variablen, Globale	Variablen , die mit einem Namen außerhalb aller Funktionen definiert werden, sind extern oder global. D.h. sie sind in jeder Funktion per Namen beliebig verfügbar, in der sie deklariert wurden. Man kann mit ihnen anstelle von Argumenten Daten zwischen Funktionen übergeben. Sie sind permanent verfügbar, auch nach Abschluss einer Funktion behält eine globale Variable ihren Wert bei, der ihr innerhalb der Funktion zugewiesen wurde.

AVR-8-bit-Mikrocontroller

Arbeiten mit CodeVisionAVR C-Compiler

Teil 07 - Anhang

7.2 Eigene Bibliothek

Da der Abschnitt **7.2 Eigene Bibliothek** eine Sammlung von diversen Projekt-Beschreibungen werden soll, werden die Beschreibungen als eigenständige Teile, d.h. als selbständige Dateien ausgeführt.

In diesen Teilen der eigenen Bibliothek wird jeweils ein ausgewähltes Projekt wegen seiner allgemeinen Anwendbarkeit ausführlich beschrieben und in deutscher Sprache kommentiert. Die Dateien stehen gesondert zum Download bereit.

Die extrem kommentierten Anweisungen sollen Neulingen 1. die praktische **C**-Programmierung nahe bringen und 2. den Einstieg in die AVR-Projekte erleichtern. Die Hinweise auf die allgemeine Anwendbarkeit der dargestellten Projekte auf andere Anwendungen sollen darüber hinaus den Einstieg in die "Modul-Welt" von [Teil 06 Modularer Aufbau der AVR-Projekte](#) fördern. Die Funktionen und ihre grundlegenden Anwendungen werden bei den Modulen nach und nach beschrieben, so dass man die verschiedenen Module nutzen und ihre Wirkung erkennen und anwenden kann, ohne genau zu wissen, wie sie funktionieren. Wer sich allerdings durch das Coding der Projekte "durchbeißt", der lernt die Funktionen bis in die Feinheiten kennen.

Mit anderen Worten: Letztlich sollte man sich grundsätzlich mit den AVR-Modulen befassen, da sie ein Optimum an Laufzeit, an Speicherbedarf, an Übersicht und an Portierbarkeit gewährleisten. Manchmal ist der Weg dorthin aber steinig und mühsam - **bitte nicht aufgeben!**

7.3 AVR-Projekte (Programmbeschreibungen)

Die Programme werden in besonderen Teilen beschrieben. Bisher stehen folgende AVR-Projekte zur Verfügung:

[7.3.1 AVR-Projekt PB LED](#)

[7.3.2 AVR-Projekt 2 Draht LCD](#)

[7.3.3 AVR-Projekt IRDMS](#)