Merkblatt: C - Referenz für Mikrocontroller (1)

Kommentare:

/* This is a comment */

/* This is a
two line comment */

// This is also a comment (to end of line)

*Verschachtelungen sind **nicht** erlaubt!*

Groß- und Kleinschreibung

Wird bei Variablennamen, Funktionen, etc. unterschieden!

Buffer ≠ buffer !

Anweisungen mit Semikolon (**;**) abschließen!

*Anweisungsblock: **{ .... }** fasst mehrere Anweisungen zus.*

Keywords in C (reservierte Ausdrücke): *(nicht erlaubt für z.B. Variablen- oder Funktionsnamen!)*

break, bit, case, char, const, continue, default, do, double, eeprom, else, enum, extern, flash, float, for, funcused, goto, if, int, interrupt, long, register, return, short, signed, sizeof, sfrb, sfrw, static, struct, switch, typedef, union, unsigned, void, while.

Zuweisung: a = b * c; // von rechts nach links

Vergleich: if (c == 255) bzw. if (a != c)

Datentypen:

Type	Size (Bits)	Range
bit	1	0 , 1
char	8	-128 to 127
unsigned char	8	0 to 255
signed char	8	-128 to 127
int	16	-32768 to 32767
short int	16	-32768 to 32767
unsigned int	16	0 to 65535
signed int	16	-32768 to 32767
long int	32	-2147483648 to 2147483647
unsigned long int	32	0 to 4294967295
signed long int	32	-2147483648 to 2147483647
float	32	1.175e-38 to 3.402e38
double	32	1.175e-38 to 3.402e38

Bei Mikrocontrollern möglichst nur **bit, char, (**ev. **int**) verwenden, wenn es ausreichend ist. Das spart Platz im (meist knappen) RAM. Bei vielen Mikrocontrollern ist ein Teil des RAMS auch bitweise adressierbar, d.h. einzelne Bits können direkt angesprochen werden.

Variablendeklarationen:

Globale Variable:

Werden außerhalb jeder Funktion deklariert, sind durch alle Funktionen überschreibbar. Beispiel::

```
unsigned char a=8;     // stehen außerhalb jeder Funktion, werden
float zahl3 = 109.72;  // hier gleichzeitig auch initialisiert!
 void main(void)
    { .... }
```

Lokale Variable (auto oder static):

*Die Variable wird innerhalb (am Anfang) einer Funktion deklariert, nach Verlassen der Funktion wird normalerweise der Speicher freigegeben (Standard=auto), außer wenn als **static** deklariert: dann immer gleicher Speicherort; der Wert bleibt erhalten, bis die Variable (in dieser Funktion) wieder überschrieben wird.*

```
void funktion_keys( )
{  static int Neuwert;       // Reservierung von 1 x 2 Bytes
   unsigned char a,b,c;      // Reservierung von  3 x 1 Bytes
   … }
```

Auch **externe** Deklarationen sind möglich (außerhalb der Quelldatei, sie müssen aber dann mit #include eingebunden werden)

Arrays:

```
char  warnung[ ] = „Achtung! Fehler!“;     //  Array of characters, initialisiert mit String,  Länge durch String festgelegt
int sollwerte[3] = (0, 750, 0xFFF);       //  Array von 3 int-Variablen,  erstes Element ansprechbar durch „sollwerte[0]“ !
```

Typenkonversion („casting“):

```
void main(void) {
int  a,c;
long b;
/* The long integer variable b will be treated here as an
   integer:  */
c = a + (int) b; }
```

Arithmetische und logische Operationen:

+	Addition	-	Subtraktion
*	Multiplik.	/	Division
%	Modulo-Division	++	Inkrement (+ 1)
--	Decrement (-1)	=	Zuweisung nach links
==	Vergleich	~	Bitweise Negation
!	Logische Negation		
<	kleiner als	>	grösser als
<=	kleiner gleich	>=	grösser gleich
&&	Log. UND	&	Bitweises UND
||	Log. ODER	|	Bitweises ODER
^	EXOR		
<<n	im Akku um n Stellen nach links schieben		
>>n	im Akku um n Stellen nach rechts schieben		

```
z.B.:  c = c << 4;  // c um 4 Stellen nach links
                    // verschieben,  Nullen nachfüllen
```

Spezielle Kurzschreibweise in „C“:

„+=“ z.B. a += 3; // Kurzschreibung für a = a + 3;
(diese Schreibweise ist auch für alle anderen obigen Operationen gültig)

Der Aufbau einer Funktion:

```
float  mittelwert (int a, int b);        // „Prototyp“ wird vor
...                                      //  1. Aufruf deklariert!
ergebnis = mittelwert(360, 500);  // Aufruf der Funktion
          // mit den Übergabeparametern a und b, Rückgabe
          // des berechn. Wertes in die float-Variable ergebnis
...
      // Eigentlicher Funktionsteil: (Definition der Funktion):
float  mittelwert (int a, int b)   // int a und b als
{                                  // Übergabeparameter
  return ((a * b) / 2);            // liefert  Ergebn. als float-Var.
}
```

Bei mehreren Übergabewerten wird nicht der Wert, sondern die Adresse übergeben („call by reference“ mit &-Operator))

Pointer „* “ und Adressoperator „&“ :

```
char *cptr;                 // Deklaration  eines Pointers (Zeigers) auf eine char - Variable
cptr = &string[0];          // cptr bekommt die Adresse des ersten Zeichens des char-Arrays „string“ zugewiesen
cptr++;                     //  der Pointer zeigt jetzt auf das nächste Element von string[ ]
```

Merkblatt: C - Referenz für Mikrocontroller (2)

Kontrollstrukturen:

Mit „break“ kann man Kontrollstrukturen unmittelbar verlassen (Vorsicht!)

```
if (x==9)
    x=1;
else
 { x++;
   a = b - c;
 } // auch ohne “else” möglich
```

```
for (n=1; n<3; n++)
   { ; }  // läuft 2x durch die
          // (Leer)-Schleife
In der Klammer:
- Initialisierung des Zählers
- Laufbedingung
- Erhöhen / Erniedrigen des
  Schleifenzählers
```

```
while (x < 100)
   { x++;  …
   }


while(1)  // Endlosschleife!
   { … }
```

```
do
 { ….
 }
 while (x < 100);

Wird mindestens 1x
durchlaufen, da
Überprüfung am Ende!
```

```
switch(Taste)
  {  case ‘L’ :
       {  ... ; break; }
     case ‘R’ :
       {  ... ; break; }
     default: break;
  }
```

Verwendung von Interrupts in C-Programmen:

Beispiel für einen Timer-Interrupt mit dem HITEC-PICL-Compiler und den Mikrocontroller Microchip PIC 16F84:

```
void interrupt timer0_int(void)     // Wird bei Interrupt angesprungen
{  sroutine_t0();                   // Aufruf der eigentlichen Interrupt-Service-Routine bei Ueberlauf von Timer 0 (FF->00)
}                                   // hier darf nur der eigentliche Aufruf der ISR stehen, sonst keine Befehle !!!
```

Beispiel für einen Timer-Interrupt mit dem CVAVR-Compiler und den Mikrocontroller ATMEL AVR ATMega8:

```
interrupt [TIM0_OVF] void timer0_ovf_isr(void)  // Timer 0 overflow interrupt service routine
{ TCNT0=0x06;  // 256 - 250 = 6 ! (250 x 4 us base time = 1 ms). Preload Timer 0 value for 1 ms - Interrupt
  counter1++;      // simply count up the software-counters every 1ms
  counter2++;      // no overflow control is done here!
  counter3++;
}
```

Einbauen von Assembleranweisungen in den C-Code:

Beispiel für den HITEC-Compiler PICL und PIC 16F84:

```
asm("NOP");  // 1us delay at  4MHz  XTAL frequency
```

Beispiel für den CVAVR und den ATMEL AVR:

```
#asm(“NOP\NOP\NOP”)   // ohne Strichpunkt,
// da als “Precompiler”-Anweisung gehandhabt
```

Preprozessoranweisungen:

englisch auch „Precompiler-statements“, steuern den Übersetzungsvorgang bzw. definieren „Makros“ oder Konstanten

#define	Makrodefinition, z.B. #define pi 3.1415	#include	Einschließen von ganzen Dateien mit Makrodefinitionen, Funktionsprototypen, usw. in den Übersetzungsvorgang z.B. #include <atmega8.h>
#undef	löschen eines Makros		
#if	Steuerung des Übersetzungs-		
#endif	vorganges (bedingte Compilierung)		
#ifdef	Abfrage, ob Makro bereits definiert ist	#pragma	zur Einstellung von compilerspezifischen Optionen („switches“)
#ifndef	Abfrage, ob Makro nicht definiert ist		
#else	Alternative, wenn (nicht) definiert		

Makros:

Der Ausdruck links wird vor dem Compilieren vom Preprozessor durch den Ausdruck rechts ersetzt, z.B.:

```
#define   quadrat(a)   a*a        // muss vor einer erster Verwendung (z.B. in einer Funktion) im Quelltext stehen!
….
c = quadrat(3);                   // „quadrat(3)“ wird hier vom Preprozessor durch  „3 * 3“ ersetzt!
                                  // zur „Laufzeit“, d.h. bei Ausführung des Programmes erhält c damit dann den Wert 9
```

Der Übersetzungsvorgang:

Quellcode (ASCII-Textdatei)		**(Übersetzer)**	(ev. Linker) **Objektdatei** andere obj. Dateien werden eingebunden)	**HEX-Datei** Maschinencode im HEX-Format (z.B. INTEL8-HEX) für Programmiergeräte
Programm.c	→	(COMPILER)		
oder			→ **Programm.obj** (ev. *.bin)	→ **Programm.hex**
Programm.asm	→	(ASSEMBLER)	+ Programm.lst (ASCII-„List“-Datei, zur Kontrolle)	

Oft werden auch eigene Dateien mit DEBUG-Informationen für Simulations und Testzwecke generiert (z.B. *.cof - Dateien)

Beim Übersetzen generiert der Compiler selbsständig den **„Startup“-Code,** den er vor dem User-Programm einfügt: Zweck: Definieren des Stackbereiches, setzen des Stackpointers, initialisieren der Variablen, Interruptvektoren setzten, Sprung zum Hauptprogramm „main()“, usw. Diesen Code sollte man möglichst nicht händisch verändern oder löschen!

Komfortables Editieren, Compilieren, Linken, Simulieren, Debuggen ist innerhalb einer **IDE (Integrated Development Environment)** möglich. Diese wird von den meisten Herstellern von Mikrocontrollern gratis zur Verfügung gestellt.