C-Appendix

Benjamin Steffen, Nico Fröhlich

October 7, 2019

Contents

1 Die Programmiersprache C

C wurde 1972-1973 von Dennis Ritchie an den Bell Labs entwickelt. C ist eine systemnahe Sprache, da sie sehr nahe an der Hardware liegt. Sie gehört zu den am weitesten verbreiteten Sprachen und findet Anwendung in Betriebssystemen, Nutzeranwendungen, Integrierten Systemen uvm..
Dieses Dokument bescheibt eine Abwandlung von C wie sie der SDCC nutzt. Dies ist allerdings nicht der offizielle Standard, dieser wird in
The C Programming Language (Dennis Ritchie, Brian Kernigham) beschrieben.

1.1 Datentypen

C kennt folgende primitive Datentypen:

Datentyp	**Speichergröße**	**Wertebereich**
bool	1 Byte	0 oder 1
char	1 Byte	-128 bis 127
short	2 Bytes	-32,768 bis 32,767
int	2 Bytes	-32,768 bis 32,767
long	4 Bytes	-2,147,483,648 bis 2,147,483,647
long long	8 Bytes	-9223372036854775808 bis 9223372036854775807
float	4 Bytes	1.2E-38 bis 3.4E+38 (6 Dezimalstellen)
double	8 Bytes	2.3E-308 bis 1.7E+308 (15 Dezimalstellen)

Bis auf float und double haben all diese Datentypen auch eine *unsigned* Variante. Das bedeutet, dass dann keine negative Werte, aber doppelt so viele positive Werte möglich sind. Desweiteren gibt es noch den Datentyp *void*, welcher das Fehlen eines Datentyps darstellt. Normalerweise gibt es auch keinen booleschen Datentyp, dann stellt 0 den Wert *false* dar, während alles andere *true* ist

1.2 Literals

Literal	**Nutzung**	**Beispiel**
0b	Binärschreibweise	0b0001 für 1
0x	Hexadezimalschreibweise	0x000F für 16
0*	Oktalschreibweise	012 für 10

1.3 Pointer

Ein Merkmal und mächtiges Werkzeug von C sind *Pointer*. Pointer sind Variablen die keine alleinstehenden Werte beinhalten, sondern auf die Speicheradresse einer anderen Variable zeigen. Die Adresse einer variable kann mithilfe des *&-Operators ermittelt werden.* Der **-Operator* hat zwei Aufgaben. Zum einen zeigt er an, dass eine Variable ein Pointer ist, zum anderen nutzt man ihn um den Wert eines Pointers auszulesen (Dereferenzieren). Hat man also beispielsweise einen Integerwert in einem **int wert = 22** gespeichert, so kann

man den Ort, an dem sich dieser Wert befindet in einem Pointer **int *ptr = &wert** festhalten. Hierbei muss beachtet werden, dass weiterhin der Datentyp angegeben wird. Grund dafür ist die unterschiedliche Größe der Datentypen. Intern wird zwischen ihnen nicht unterschieden, deshalb muss dem Compiler gesagt werden, wie viele Bytes gelesen oder geschrieben werden sollen. Hier sind Pointer zwischen einem und vier Bytes groß.

1.4 Strings

In C werden Zeichenketten (Strings) wörtlich genommen, dort gibt es nämlich keine Stringklasse wie in Java. Hier wird lediglich ein Array an Zeichen verwendet bzw. ein *char**, wobei die Zeichenkette bei dem ASCII code 0 bzw *'\0'* endet. Angegeben wird diese wie in Java im code mit "
Beispiel: *char* string = "Hello, World!";*.

1.5 Compiler und Linker

Anders als Java ist C nicht interpretiert, sondern wird direkt in Maschinencode umgewandelt. Dieser Vorgang ist generell in zwei Schritte unterteilt. Zuerst wird der C Code Kompiliert, also in Maschinencode übersetzt. Dabei entstehen sogenannte *Object Files*. Der Linker bringt diese Dateien dann in einer Ausführbaren Datei zusammen und stellt Referenzen zwischen Funktionen und Bibliotheken her.

1.6 Header

Neben den Sourcedateien (.c) gibt es auch *Headerdateien* (.h). Diese Dateien werden genutzt um Funktions- und Strukturprototypen zu deklarieren und diese durch eine Bibliothek oder eine andere Sourcedatei zugänglich zu machen. Headerdateien werden über *#include <header.h>* eingebunden.
Beim Kompilationsprozess wird die hier angegebene Datei einfach an die Stelle des include Befehls kopiert.

1.7 Syntax

Im folgenden wird kurz aufgezeigt, wie das Deklarieren und Verwenden von Variablen und Kontrollstrukturen aussieht. Beispiel:

```
#include <stdio.h> //Einbinden eines Headers

int main(int argc, char** argv) { //Definition des Programmeinstiegpunktes
    int zahl = 12345;             //Deklaration verschiedenster Variablen
    float kommazahl = 123.45;
    char charakter = 'a';
    char *text = "Beispiel Text\0";  // \0 Markiert das Ende eines Characterstrings,
                                     // ist aber je nach implementation nicht immer nötig
    for(int i=0; i<10; i++) {
        printf("Hello World! %d", i); //10 malige ausgabe eines Textes,
    }                                  //mit integer Wert in einem Formatstring


    while(zahl>0) //Blöcke {} sind nicht notwendig,
        zahl--;   //wenn unter dem Schleifenkopf nur eine Anweisung steht

    return 0; //Die Funktion ended mit "return 0;" um erfolgreiche Abarbeitung zu melden
}
```

1.8 Operator

Die logischen sowie die bitweisen Operatoren sind in C genau wie in Java.

Operator	Beschreibung	Beispiel
&&	Logisches und	(a && b)
!	Logisches nicht	!a
||	Logisches oder	(a || b)
+, +=	Addition (und setzen)	a + b, a += b
-, -=	Subtraktion (und setzen)	a - b, a -= b
$*, * =$	Multiplikation (und setzen)	$a * b, a* = b$
/, /=	Division (und setzen)	a / b, a /= b
%, %=	Modulo (und setzen)	a % b, a %= b
&	Bitweise UND	0b011 & 0b010
|	Bitweise inclusive OR	$0b011 \vert 0b010$
^	Bitweise exclusive OR	0b011 ^0b010
~	Bitweise Negation	~0
<<	Bitweise links verschieben	1 <<3
>>	Bitweise links verschieben	4 >>2

1.9 C8051F340.h

Siehe: https://github.com/MrTroble/ADC-C8051F340/blob/master/C8051F340.h

Der C8051F340 Header ist ein von Cygnal/SiLabs bereit gestellter Header der bereits alle ASM Register mit ihren jeweiligen Adressen enthält. Dabei können die jeweiligen Register wie normale variablen mit bit Operatoren angesteuert werden. Beispiel:

```
#include <C8051F340.h>

void main() {
    ADC0CN |= 0x80; // Enable ADC0 subsystem
    AMX0N = 0x1F; // Set negative mux to GND -> use single end mode
    // use AMX0P default config -> P1.0
    while (1) { // Infinite loop
        ADC0CN |= 0x10; // or 16 to trigger AD0BUSY
        while (!(ADC0CN & 0x20 /* or 32*/)) {continue;} // wait for the AD0INT to become 1
        // go ahead after thread block
        P3 = ADC0L; // Put low bits in P3
        P0_0 = ADC0H & 1; // Put first bit of high bits into P0.0
        P0_1 = ADC0H & 2; // Put second bit of high bits into P0.1
    }
}
```

2 Advanced

2.1 Structs

Structs ermöglichen es, verschiedene Datentypen in einem Datenobjekt zusammenzufassen. Dies ist bei Hierarchiestrukturen nützlich und trägt auch generell zu einer guten Code Organisation bei. Das zugreifen auf Elemente ist mittels des *Punktoperators* möglich. Structs werden außerhalb von Funktionen definiert.

```
struct Person {
    char name[64];
    int alter;
};

int main() {
    struct Person Max;
    strcpy(max.name, "Max");
    max.alter = 17;
}
```

2.2 Unions

Unions ermöglichen es, verschiedene Datentype im gleichen Speicherbereich abzulegen. Allerdings kann nur ein Wert auf einmal belegt werden. Dies ermöglicht Speichereffizientes programmieren.

```
union overlay {
    long longWert;
    float floatWert;
}
```

Hier wird ein *long* und ein *float* in den gleichen Speicher gelegt. Zugriff erflogt wie bei Structs.

2.3 Bitfields

Mit Bitfields ist es möglich, die genaue Anzahl an Bits, die ein Wert belegen soll, zu bestimmen. Hat man beispielsweise mehrere *boolean* Werte, wäre es Verschwendung dafür 1, 2, oder sogar 4 Bytes pro Wert zu belegen. Im folgenden Beispiel wird gezeigt wie man dieses Problem löst.

```
struct {
    unsigned char bool0: 1;
    unsigned char bool1: 1;
    unsigned char bool2: 1;
    unsigned char bool3: 1;
    unsigned char bool4: 1;
    unsigned char bool5: 1;
    unsigned char bool6: 1;
    unsigned char bool7: 1;
};
```

Diese Struktur beinhaltet nun 8 Werte, die alle ein bit groß sind und daher genauso viel Speicher benötigen wie ein regulärer *char*.

2.4 typedef

Das *typedef* keyword wird verwendet, um Typen neue Namen zu geben. So kann man beispielsweise das Erstellen einer struct vereinfachen. (vergleiche oben)

```
typedef struct person_t {
    (...)
} Person;

int main() {
    Person p;
}
```

2.5 SDCC - Small Device C Compiler

Siehe: http://sdcc.sourceforge.net/doc/sdccman.pdf