Programming embedded devices with C

Johannes Zeitler

Sommerakademie in Leysin, August 2016

Abstract

In vielen Bereichen unseres alltäglichen Lebens begegnen uns eingebettete Systeme, sei es das Fitnessarmband, die Waschmaschine, der Fernseher, das Smartphone oder das Multimediasystem im Auto. Sie alle beinhalten mehr oder weniger komplexe und leistungsfähige Microcontroller, die für die Steuerung des jeweiligen Gerätes verantwortlich sind. Die Anforderungen an diese eingebetteten Systeme sind, je nach Einsatzzweck, sehr unterschiedlich. So müssen zum Beispiel die in Armbanduhren verbauten Systeme einen möglichst geringen Energieverbrauch und kleine Abmessungen aufweisen, Controller in der Fahrzeugsteuerung müssen hohen Echtzeitanforderungen genügen und der massenhaft in einem Kinderspielzeug eingebaute Chip muss vor allem billig sein. Um all diese unterschiedlichsten Anforderungen erfüllen zu können, muss sich auch die Programmierung flexibel auf die jeweiligen Systeme anpassen lassen. Im Folgenden wird dargestellt werden, warum die Programmiersprache „C“ besonders gut für eingebettete Systeme geeignet ist.

Inhalt

1 Eigenschaften der Sprache C

C ist eine sogenannte imperative Sprache, gliedert ein Programm also in eine Menge von Funktionen und Variablen. Die verglichen mit anderen Programmiersprachen relativ kleine Menge an Schlüsselwörtern bietet die Möglichkeit, sehr einfache Compiler zu schreiben. So erfolgt bei der Entwicklung eines neuen Systems die erste Programmierung meist in C. Die Sprache bietet die Möglichkeit, Speicherzellen direkt anzusprechen, womit auch ein unmittelbarer Zugriff auf in den Speicher eingebundene Hardware-Register möglich ist. Zur Laufzeit des Programms erfolgt fast keine Fehlerprüfung, nicht einmal grobe Fehler wie die Division durch Null werden auf allen Systemen abgefangen und gemeldet. C bietet die Möglichkeit einer Modularisierung auf Dateiebene, um beispielsweise Funktionen aus externen Bibliotheken einzubinden. Eine Objektorientierung wie bei Java oder C# ist nicht vorhanden.

2 Eignung der Sprache C für eingebettete Systeme

Eine Anforderung an Programmiersprachen für eingebettete Systeme ist die Laufzeiteffizienz des Codes. Im Fall eines Microcontrollers läuft ein übersetztes C-Programm direkt auf dem Prozessor, es sind keine weiteren Abstraktionsschichten vorhanden. Zur Laufzeit erfolgt weder eine Prüfung auf Programmierfehler noch auf Datenzugriffe, was bei gründlicher und sorgfältiger Programmierung einen erheblichen Geschwindigkeitsgewinn bedeutet.

Des Weiteren bietet die Sprache eine große Platzeffizienz. Code und Daten lassen sich sehr kompakt speichern. In Anbetracht des knappen und teuren Speichers auf Microcontrollern ist dies von großem Vorteil.

Ein weiterer Aspekt ist die Direktheit, mit der in C Zugriffe auf Speicher und Register möglich sind. Jede Speicherzelle und jedes Hardware-Register kann über C-Code direkt angesprochen werden, was effiziente hardwarenahe Programmierung überhaupt erst ermöglicht.

Nicht zuletzt punktet C auch vor allem durch seine Portabilität, da beinahe für jede existierende Plattform ein Compiler vorhanden ist. Außerdem existieren für C verschiedene Cross- und Target-Compiler, die es ermöglichen, den auf einer Plattform übersetzten Code auf einer anderen auszuführen. Anders wäre zum Beispiel die Programmierung von Microcontrollern gar nicht möglich, da oft die Hardware dieser eingebetteten Systeme nicht ausreichend für eine Programmoptimierung und –kompilierung ist.

3 Grundlagen der Sprache C

Im Folgenden werden kurz die syntaktischen Grundlagen eines C-Programms dargestellt, um danach vertieft in die Programmierung eingebetteter Systeme einsteigen zu können.

3.1 Struktur eines C-Programms

Jedes in C geschriebene Programm hat einen ähnlichen Aufbau. Zunächst werden externe Bibliotheken eingebunden, anschließend werden Globale Variablen und Subfunktionen definiert. Die main-Methode ist der Einstiegspunkt eines C-Programms, sie wird direkt beim Programmstart aufgerufen.

```
#include <display.h>                // Bibliotheken einbinden

int zahl = 42;                      // Globale Variablen

void addAndShow(int a, int b) {     // Sub-Funktionen
      int sum = a + b;
      displayShow(sum);
}

int main(void) {                    // Main-Funktion
      addAndShow(zahl, 17);
      while(1) {}
}
```

3.2 Datentypen

In C existieren ausschließlich Datentypen zur Darstellung von Zahlen. Der C-Standard definiert dabei lediglich mindest-Bitlängen für die einzelnen Datentypen. Zur Programmierung eingebetteter System empfiehlt sich die Verwendung der Bibliothek „stdint.h“, welche Zahlentypen definierter Länge bereitstellt. Die folgenden Tabellen geben einen kurzen Überblick über die häufigsten Datentypen.

C-Standard				
char	short	int	long	long long
>= 8 bit	>= 16 bit	>= 16 bit	>= 32 bit	>= 64 bit

stdint.h			
uint8_t	0 … 255	int8_t	-128 … +127
uint16_t	0 … 65.535	int16_t	-32.768 … + 32.767
uint32_t	0 … 4.294.967.295	int32_t	-2.147.483.648 … +2.147.483.647
uint64_t	0 … 1,8*10^19	int64_t	-9.2*10^18 … +9.2*10^18

Neben diesen Datentypen existiert noch der leere Typ „void“, der vor allem bei der Deklaration von Funktionen Verwendung findet. Die Datentypen für die Darstellung von Dezimalzahlen „float“, „double“, „long double“ sowie für komplexe Zahlen „_Complex“ und „_Imaginary“ finden aufgrund des unverhältnismäßig hohen Speicherverbrauchs so gut wie keine Anwendung in der Programmierung eingebetteter Systeme, weshalb an dieser Stelle nicht weiter auf sie eingegangen wird.

3.3 Literalformen

Zur Eingabe von Zahlen können unterschiedliche Zahlensysteme verwendet werden, wie das folgende Programmierbeispiel demonstriert.

```
int a = 42;        // dezimal
int b = 0x2A;      // 0x…: hexadezimal
int c = 052;       // 0…:  oktal
```

3.4 Bitweise Operatoren

Für das Verständnis der nachfolgenden Programmierbeispiele ist die Kenntnis über die bitweisen Operatoren in C essentiell. Die einzelnen Operatoren werden nun kurz vorgestellt und in je einem Beispiel veranschaulicht.

Operator	Beschreibung
&	bitweises "und"
|	bitweises "oder"
^	bitweises "Exklusiv-Oder"
~	bitweise Inversion
<<	bitweises Linksschieben
>>	bitweises Rechtsschieben

Bit	7	6	5	4	3	2	1	0
x = 115	0	1	1	1	0	0	1	1
~x	1	0	0	0	1	1	0	0
7	0	0	0	0	0	1	1	1
x & 7	0	0	0	0	0	0	1	1
x | 7	0	1	1	1	0	1	1	1
x ^ 7	0	1	1	1	0	1	0	0
x << 2	1	1	0	0	1	1	0	0

4 Zeiger

Eine Besonderheit der Programmiersprache C, die sie sehr geeignet zur hardwarenahen Programmierung macht, ist die Implementierung von Zeigern. Diese erlauben den direkten Zugriff auf bestimmte Speicherzellen, womit unter anderem das direkte Auslesen von Hardware-Registern oder die Auslegung von Funktionen als „call-by-reference“ möglich ist.

4.1 Syntax

Eine Zeigervariable enthält als Wert die Adresse einer Speicherzelle. Durch den Adressoperator &x wird die Adresse der Variable x abgerufen, der Verweisoperator *y liefert den Wert, der im Speicher an Adresse y gespeichert ist.

Im folgenden Programmierbeispiel wird die grundlegende Syntax beim Umgang mit Zeigern erläutert.

```
int x = 2;          //Deklaration der Integer-Variable x, Zuweisung des Wertes 2
int *zeiger;        //Deklaration einer Zeigervariable
zeiger = &x;        //Die Variable zeiger erhält als Wert die Adresse von x
int y;              //Deklaration der Variable y
y = *zeiger;        //Die Variable y erhält den Wert, der an der Adresse
                    //gespeichert ist, auf die zeiger verweist
```

4.2 Call by reference

Eine häufige Anwendung von Zeigern ist die Implementierung von call-by-reference-Methoden. Sie bekommen als Übergabeparameter eine oder mehrere Speicheradressen von Variablen und sind dadurch in der Lage, diese direkt zu verändern. Im nachfolgenden Code-Beispiel wird eine Funktion zum Tauschen des Inhaltes zweier Variablen vorgestellt.

```
void tausche (int *a, int *b) // tausche erhält als Parameter zwei Zeiger
{
      int tmp;                // Puffer anlegen
      tmp = *a;               // Wert von Speicher a in den Puffer schreiben
      *a = *b;                // Wert von Speicher b in Speicher a schreiben
      *b = tmp;               // Puffer in Speicher b schreiben
}

int main()
{
      int z1 = 1;
      int z2 = 2;
      tausche(&z1, &z2);            // Adressen von z1 und z2 übergeben
}
```

4.3 Zeiger auf Funktionen

Zeigervariablen können nicht nur auf die Speicherzelle einfacher Variablen verweisen, sie können auch die Adresse des Startpunktes von Funktionen enthalten. Im folgenden Programmierbeispiel wird dieses Konzept genutzt, um eine Funktion zu implementieren, mit der beliebige andere Methoden mehrfach ausgeführt werden können.

Die Methode „mehrfachAusfuehren“ bekommt die Adresse einer Funktion sowie die Anzahl an Wiederholungsdurchläufen übergeben. In jedem Durchlauf der Zählschleife springt das Programm an die übergebene Funktions-Adresse, um nach Beendigung der Aufgabe wieder in die Schleife zurückzukehren.

```
void mehrfachAusfuehren( void (*aufgabe)(void) , uint16_t anzahl)
{
	for(uint16_t i = 0; i < anzahl; i++) {
		aufgabe();
	}
}

void blinken(void)
{
	ledAn();
	warten(500);
	ledAus();
	warten(500);
}

int main(void)
{
	mehrfachAusfuehren(blinken, 20);
	while(1) {}
}
```

5 Hardwarenahe Programmierung am Beispiel eines ATmega32

Für das folgende Beispiel wird ein ATmega32-Microcontroller verwendet, an dem an Pin 7 des Port D eine LED in active-low-Schaltung angeschlossen ist. Im ersten Schritt wird Pin7, PortD als Ausgang konfiguriert, indem das siebte Bit im Data-Direction-Register D (DDRD) auf 1 gesetzt wird. Das anschließende An- und Ausschalten der LED erfolgt durch das Setzen einer 1 bzw. 0 im siebten Bit des PORTD-Registers. Alle drei Schritte des Hardware-Zugriffs werden zunächst mit Hilfe der Bibliothek „avr/io.h“ ausgeführt, die beispielsweise die Wertzuweisung in DDRD- und PORTD-Register bereitstellt. Unter jedem dieser Schritte befindet sich auskommentiert eine äquivalente Zuweisung, die die Register direkt über die in der Dokumentation des ATmega32 zu findenden Speicheradressen mit Hilfe von Zeiger-Syntax anspricht.

```
#include <avr/io.h>     // Bibliothek einbinden

int main(void)
{     // Hardware initialisieren (LED an Port D Pin 7, active low)

      DDRD |= (1<<7);   // Port D Pin 7 ist Output
      // (* (volatile uint8_t*) ( 0x31 ) ) |= (1<<7);


      PORTD |= (1<<7);  // Pin 7 high -> LED ist aus
      // (* (volatile uint8_t*) ( 0x32 ) ) |= (1<<7);


      PORTD &= ~(1<<7); // Port D Pin7 low -> LED an
      // (* (volatile uint8_t*) ( 0x32 ) ) &= ~(1<<7);


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

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
$31	DDRD	DDD7	DDD6	DDD5	DDD4	DDD3	DDD2	DDD1	DDD0
$32	PORTD	PORTD7	PORTD6	PORTD5	PORTD4	PORTD3	PORTD2	PORTD1	PORTD0

6 Zusammenfassung

Die Programmiersprache C zeigt sich aufgrund einer Vielzahl an Eigenschaften als besonders geeignet für die Verwendung in eingebetteten Systemen. Da ist zum einen die geringe Komplexität des grundlegenden C-Codes zu nennen, der effiziente und kleine Programme ermöglicht, welche auch für Plattformen mit knappen Ressourcen bezüglich Speicher und Rechenleistung geeignet sind. Die Beschränkung auf Integer-Datentypen unterschiedlicher Bitlängen ermöglicht eine optimale Speicherausnutzung. Durch die Verwendung von Zeigern kann der Speicher bereits zum Zeitpunkt der Programmierung effizient strukturiert werden und es werden direkte Hardware-Zugriffe ermöglicht.

Zusammenfassend lässt sich feststellen, dass die vermeintlichen Beschränkungen von C in Bezug auf fehlende Objektorientierung oder die geringe Anzahl standardmäßig integrierter Funktionen als vorteilhaft für die Verwendung in eingebetteten Systemen verstanden werden müssen. Vielmehr ermöglicht C eine effiziente und hardwarenahe Programmierung ohne unnötigen Ballast.