1 C für µC-Anwendungen

Die im Folgenden beschriebenen Sprachelemente von C sind zwar nicht auf µC-Programme beschränkt, werden dort aber sehr viel häufiger als in PC-Programmen verwendet. Der Grund ist, dass sich einige Probleme auf dem PC gar nicht und andere sehr selten ergeben.

1.1 Datentypen mit definierter Länge

Die bisher bekannten Typen in C haben mit Ausnahme des *char* keine allgemeingültige Länge. Für Anwendungen in einem µC muß aber in manchen Fällen die Länge exakt bekannt sein, beispielsweise um bei einem Zugriff auf ein Register dessen Länge nicht zu überschreiten. Zudem ist gerade in µC-Systemen Datenspeicher knapp (oder alternativ teuer), so dass es sinnvoll ist, die jeweils kürzestmögliche Integerdarstellung für ein Datum zu wählen. Außerdem sind in µC-Systemen noch immer Rechenkerne mit einer natürlichen Wortlänge von 8 Bit weitverbreitet. Alle Werte mit größerer Stellenzahl werden dann erheblich langsamer bearbeitet.
In der Standarderweiterung *C99* gibt es u.a. deswegen Typen, deren Länge genau definiert ist. Diese Typen haben einen systematischen Aufbau:

BNF: <typ> := [u]int<len>_t
<len> := 8, 16, 32, 64

Das optionale *u* steht für unsigned (vorzeichenloser Typ), die Länge *len* kann die Werte 8, 16, 32 und 64 annehmen. Damit wäre eine Definition eines vorzeichenlosen Integer mit 16 Bit Länge: uint16_t data;

Der C-Compiler muß dann den Standard C99 unterstützen und zudem sollte der Header *stdint.h* eingebunden werden.

1.2 Schiebeoperationen (<<, >>)

Mit den Schiebeoperationen kann ein Integer um n Bitstellen nach links (<<) oder rechts (>>) verschoben werden. Stellen, die herausgeschoben werden, werden einfach weggeworfen und sind sofort verloren. Auf der anderen Seite müssen dagegen Stellen nachgezogen werden. Bei einer Verschiebung nach links werden immer Nullen nachgezogen. Jede Verschiebung um eine Stelle entspricht damit eine Multiplikation des Wertes mit 2 (Achtung bei Integer mit Vorzeichen auf Wertebereich!).
Bei einer Verschiebung nach rechts wird normalerweise wie folgt vorgegangen: Ist der Integer ohne Vorzeichen (unsigned), dann werden Nullen nachgezogen. Ist der Wert mit Vorzeichen (signed), dann wird der Wert des MSB nachgezogen. Damit ist sichergestellt, dass negative Werte negativ bleiben. Allerdings ist diese Eigenschaft nicht im Standard festgeschrieben.
Eine Rechtsverschiebung um eine Stelle ist bei positiven Zahlen gleichbedeutend mit einer Division durch 2.

Beispiele: 0x33>>2 ergibt 0x0c, 0x33<<3 ergibt 0x98.

Schiebeoperationen dienen zum einen als Ersatz für Multiplikationen mit 2 bzw. Divisionen durch 2, weil µC spezielle Schiebebefehle haben, die sehr schnell ausgeführt werden. Außerdem werden Schiebeoperationen auch gerne benutzt, um eine einzelne 1 an eine bestimmte Position zu bringen. In diesem Anwendungsfall werden fast immer Konstanten mit Hilfe der Verschiebung gebildet. Konstanten können vom Compiler zum Zeitpunkt der Kompilation berechnet werden, so dass die Verwendung einer Schiebeoperation weder zusätzliche Rechenzeit zur Laufzeit noch Programmspeicher erfordert.

Beispiel:
Der Praktikums-µC hat für jeden der 8 GPIO-Ports ein „Setz-Register“ (SET0-SET7), bei dem jedem Bit eines solchen Registers ein Bit zugeordnet ist (UM, Kapitel 19.5.3.7). Dem GPIO5[3] ist das dritte Bit im Register SET5 zugeordnet, dem GPIO3[29] das 29. Bit im Register SET3. Schreibt man bei diesen acht Registern eine 0 an eine Bitstelle, dann passiert mit dem zugeordneten GPIO *gar nichts*. Schreibt man aber eine 1 an eine Bitstelle, dann wird der zugeordnete GPIO *gesetzt*. (Damit das einen Effekt am Pin hat, muss natürlich der Pin elektrisch als Ausgang definiert sein und der GPIO muss auch als Ausgang konfiguriert sein.)
Möchte man nun in der Anwendung den GPIO3[11] setzen, ohne dass sonst irgendein GPIO beeinflusst wird, dann kann man in das Register SET3 den Wert 0x20000000 schreiben:

```
SET3=0x20000000;
```

Man kann aber auch dem Compiler die Berechnung der Konstante überlassen:

```
SET3=(1<<29);
```

Es ist offensichtlich, dass die zweite Variante für den Programmierer wesentlich einfacher zu schreiben ist.

1.3 Bit-Operationen

Sehr häufig müssen in einer µC-Anwendung einzelne Bits innerhalb eines Integer bearbeitet werden. Das liegt daran, dass in Registern eine einzelne Bitstelle unabhängig von den anderen Bitstellen jeweils eine eigene Funktion haben kann. Man muss dann darauf achten, dass man immer nur den Wert einer einzelnen Bitstelle verändert oder abfragt. Dazu gibt es in C vier Operatoren, die auf einzelnen Bits arbeiten: und, oder, exklusiv-oder, nicht. Tabelle 1 zeigt die Wahrheitstabellen der vier Operatoren

a	b	a und b	a oder b	a exor b	nicht a
0	0	0	0	0	1
0	1	0	1	1	0
1	0	0	1	1	
1	1	1	1	0	

Tabelle 1: Wahrheitstabellen für die Bitverknüpfungen

Wichtig:
In C gibt es auch das logische Und (&&), das logische Oder (||) und die logische Negation (!). Diese Operatoren sind hier nicht verwendbar, da sie nicht auf Bitebene arbeiten!

1.3.1 Bitweise Negation (~)

Die bitweise Negation kehrt alle Stellen eines Integer *einzeln* um (0 -> 1, 1 -> 0). Mit Hilfe der Schiebeoperationen lässt sich sehr einfach eine Konstante erzeugen, die an einer Stelle n eine 1 hat und an allen anderen Stellen eine 0: *(1<<n)* .
Oft benötigt man aber eine Konstante, die an *einer* Stelle n eine 0 hat und an allen anderen Stellen eine 1. Dazu kann man die bitweise Negation einsetzen: *~(1<<n)* .
Würde man stattdessen *(0<<n)* schreiben, dann hätte man eine Konstante erzeugt, die an *allen* Bitstellen eine Null aufweist.

1.3.2 Und-Verknüpfung (&)

Die Und-Verknüpfung hat in µC-Programmen zwei häufige Anwendungen:

a) Gezieltes Nullsetzen einer oder mehrerer Bitstellen
b) Abfragen des Wertes einer Bitstelle, unabhängig vom Wert der übrigen Stellen.

Der Praktikums-µC hat für jeden der 8 GPIO-Ports ein Richtungs-Register (DIR0-DIR7), bei dem jedem Bit eines solchen Registers ein Bit zugeordnet ist (UM, Kapitel 19.5.3.3). Dem GPIO5[3] ist z.B. das dritte Bit im Register DIR5 zugeordnet.
Schreibt man bei diesen acht Registern eine 0 an eine Bitstelle, dann ist der zugeordnete GPIO als Eingang konfiguriert, sonst als Ausgang.
Möchte man den GPIO5[3] als Eingang konfigurieren, ohne die Richtungseigenschaft der anderen GPIO5[x] zu ändern, dann muss man gezielt die dritte Bitstelle im Register GPIO5 nullsetzen:

```
GPIO5 = GPIO5 & ~(1<<3);
```

Der Praktikums-µC hat für jeden der 8 GPIO-Ports ein Pin-Register (PIN0-PIN7), bei dem jedem Bit eines solchen Registers ein Bit zugeordnet ist (UM, Kapitel 19.5.3.5). Dem GPIO5[3] ist z.B. das dritte Bit im Register PIN5 zugeordnet.
Liest man ein solches Register, dann bekommt man an jeder Stelle den Wert (0 oder 1), dem die jetzt gerade am zugehörigen Pin anliegende Spannung (L oder H) entspricht.
Möchte man gezielt den GPIO5[3] abfragen, ohne dass die anderen GPIO5[x] auf die Abfrage einen Einfluss haben, dann kann man in C schreiben:

```
if (PIN5 & (1<<3))
{
   // GPIO5[3] ist gerade H
   ...
}
```

1.3.3 Oder-Verknüpfung (|)

Die Hauptanwendung der Oder-Verknüpfung in µC-Programmen ist das gezielte Setzen einer oder mehrerer Stellen in einem Datenwort. Alle anderen Stellen bleiben dabei unverändert.

Möchte man den GPIO5[3] als Ausgang konfigurieren, ohne die Richtungseigenschaft der anderen GPIO5[x] zu ändern, dann muss man gezielt die dritte Bitstelle im Register GPIO5 setzen:

```
GPIO5 = GPIO5 |(1<<3);
```

1.3.4 Exor-Verknüpfung (^)

Der Name lautet ausgeschrieben Exklusiv-Oder, in der µC-Technik ist als Kürzel *xor* gebräuchlich. Exklusiv-Oder steht für „ausschließendes Oder“. Dies ist die Verknüpfung, die auch in der Umgangssprache gemeint ist. „Gehen wir zu dir oder mir“ schließt die Möglichkeit, beides gleichzeitig zu tun aus. In Tabelle 1 ist dieser Sachverhalt dargestellt.

Die Hauptanwendung der Exor-Verknüpfung in µC-Programmen ist die Invertierung einer Stelle (0->1, 1->0), ohne die anderen Stellen zu verändern.

Möchte man den Ausgabewert des GPIO5[3] invertieren, ohne die anderen GPIO5[x] zu ändern, dann kann man schreiben:

```
PIN5 = PIN5 ^ (1<<3);
```

In der Praxis kommt diese Aufgabenstellung deutlich seltener als die vorhergehenden (Und, Oder) vor.

1.4 Variableneigenschaften

In C können Variablen mit speziellen Eigenschaften versehen werden, um ein bestimmtes Verhalten zur Laufzeit zu erzwingen. Davon werden hier zwei Eigenschaften behandelt.

1.4.1 volatile

Die Kennzeichnung einer Variable als *volatile* bedeutet, dass diese Variable außerhalb des normalen (sequentiellen) Programmflusses verändert werden kann. Aus diesem Grund muß das Programm zur Laufzeit den Inhalt der Variable bei jeder Abfrage neu aus dem Speicher holen. Das ist (später) speziell bei der Programmierung mit Interrupts erforderlich. Ein gern benutzter Nebeneffekt ist dabei, dass Ausdrücke, in denen diese Variable vorkommt, nicht optimiert werden. Heutige Compiler sind bereits so ausgereift, dass sie „leere" Schleifen wegoptimieren. Anstelle des Codes *for (i=0; i<10000; i++);* setzt der Compiler gleich *i=10000;* ein. Das ist zwar vom Ergebnis her richtig, nur wollte der Programmierer hier vermutlich nur Zeit verstreichen lassen, indem er den µC 10000 mal die Variable i hochzählen ließ. In der optimierten Version ist dieser Effekt eliminiert. Hier wäre also die folgende Deklaration angebracht: *volatile int i;*

1.4.2 static

Wenn diese Kennzeichnung bei einer **lokalen** Variable benutzt wird, dann wird die Lebensdauer der Variable auf „solange das Programm lebt" gesetzt. Andernfalls ist die Lebensdauer auf „solange der Block lebt" begrenzt.
Was die Lebensdauer betrifft, verhält sich eine solche Variable also wie eine globale Variable. Sie ist aber nur innerhalb des Blocks sichtbar, in dem sie definiert ist – hier verhält sie sich also weiterhin wie eine lokale Variable.
Eine solche Variable kann wie jede andere Variable initialisiert werden. Diese Initialisierung wird aber einmalig beim Programmstart vorgenommen, obwohl es im C-Quelltext so aussieht, als würde die Variable bei jedem Funktionsaufruf initialisiert:

```
uint32_t messung(void)
{
  static uint32_t n=0;  // Mitzählen der Messzyklen
  …

  n++;
  return n;
}
```

Die Funktion *messung()* soll selber mitzählen, wie oft sie bereits aufgerufen wurde und das dem Aufrufer mitteilen. Dazu wird die statische Variable n deklariert, die nur in dieser Funktion sichtbar ist und einmalig mit 0 initialisiert wird.
Am Ende jedes Durchlaufs der Funktion wird die Variable erhöht und der Wert dem Aufrufer zurückgegeben. Ohne die Eigenschaft *static* würde der Wert aber bei jedem Funktionsaufruf erneut auf Null gesetzt und ein Mitzählen wäre nicht möglich.

(Die Kennzeichnung **globaler** Variablen als *static* hat einen anderen Effekt.)