Zahlentypen und mathematische Funktionen in C

Jörn Loviscach

Versionsstand: 19. November 2010, 09:25

Die nummerierten Felder sind absichtlich leer, zum Ausfüllen in der Vorlesung.
Videos dazu: http://www.youtube.com/joernloviscach

1 Ganzzahlige Typen: Zahlenumfang, Zweierkomplement

Daten werden im Speicher als Sammlung von Bits abgelegt. Welche Bedeutung diese Bits haben, beschreibt der „Typ“ des jeweiligen Objekts. Wir kennen bereits die Typen `int` und `bool`. Der Typ `int` steht für das typische Ganzzahlformat des jeweiligen Prozessors, also 16 Bit beim MSP430. Der Typ `bool` muss eigentlich nur 0 und 1 speichern können, belegt aber auf praktisch allen Rechnern mindestens ein Byte.

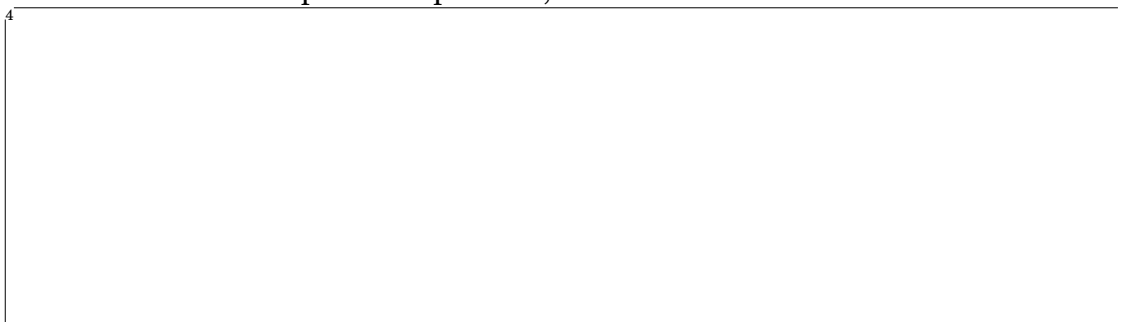
Wenn man alle 16-Bit-Zahlen ohne weitere Tricks von binär nach dezimal umrechnet, erhält man als Ergebnisse: [1]

Zahlen außerhalb dieses Bereichs lassen sich also nicht mit den 16 Bit darstellen. Die negativen Zahlen würden komplett fehlen.

Offensichtlich kann `int` aber auch mit negativen Zahlen umgehen. Der Trick dafür ist in praktisch allen Rechnern die Darstellung negativer Zahlen per Zweierkomplement. Die Bitmuster, deren höchstwertiges Bit eine 1 ist, erhalten dann eine andere Bedeutung: Sie werden die negativen Zahlen. Zum Beispiel ist `0xFFFD` dann nicht mehr [2], sondern [3]. Der Vorteil dieser Darstellung ist, dass die Rechenwerke bei Addition, Subtraktion

und Multiplikation keinen Unterschied zwischen positiven und negativen Zahlen machen müssen. Beispiel: Was passiert, wenn man -3 und 5 addiert?

[4]

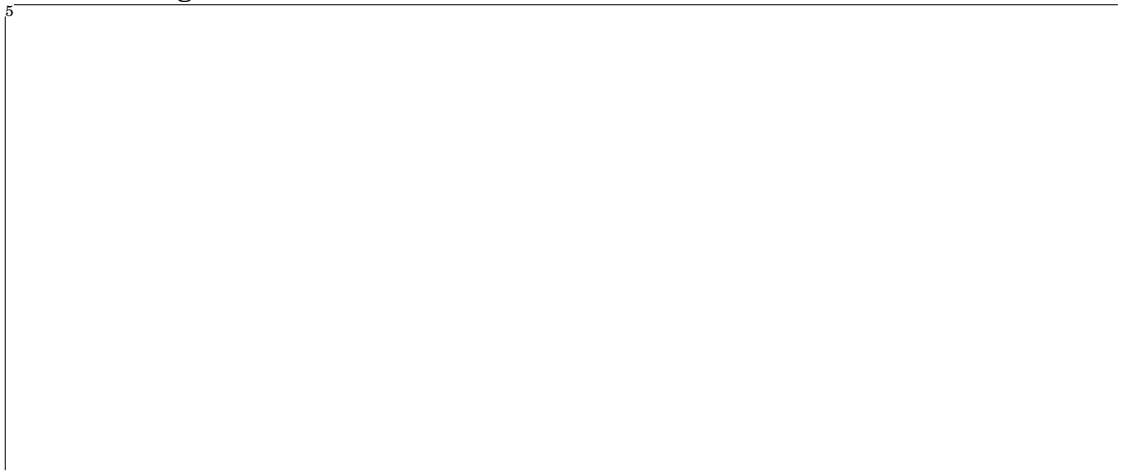


Man beachte: Der Überlauf [overflow] wird ignoriert. Das kann auch gefährlich sein; dazu nachher mehr.

Ein Beispiel, wieso das funktioniert: $(-7)\cdot(-10) = ?$. Die -7 wird gespeichert als $2^{16}-7$. Die -10 wird gespeichert als $2^{16}-10$. Wenn man die beiden gespeicherten Zahlen multipliziert, ergibt sich $(2^{16}-7)\cdot(2^{16}-10) = 2^{32}-17\cdot 2^{16}+70$. Die beiden ersten Summanden fallen durch den Überlauf weg. Dies ist Rechnen modulo 2^{16}.

Die 16-bittigen Binärzahlen mit Vorzeichen in Zweierkomplementdarstellung sind damit die folgenden:

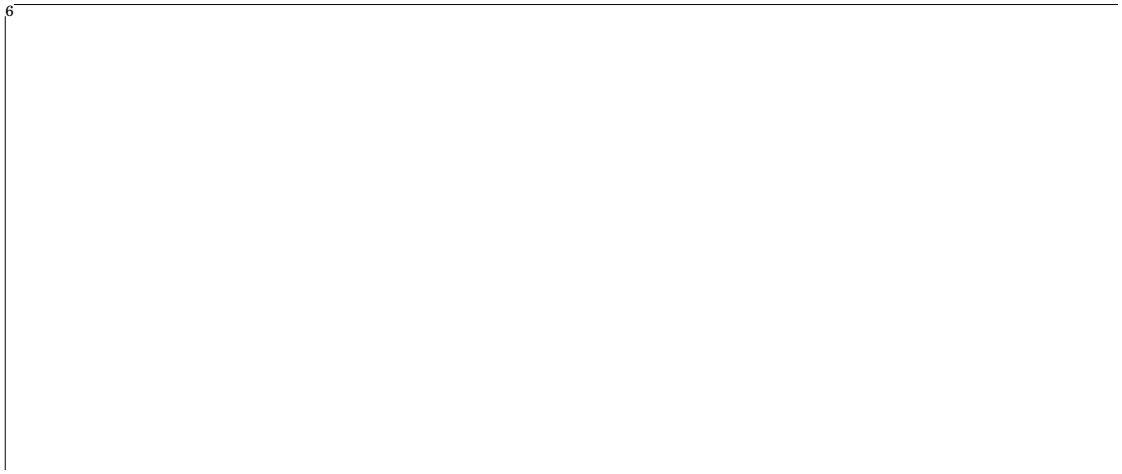
[5]



So funktioniert der Typ `int` auf praktisch allen 16-Bit-Rechnern, einschließlich dem MSP430.

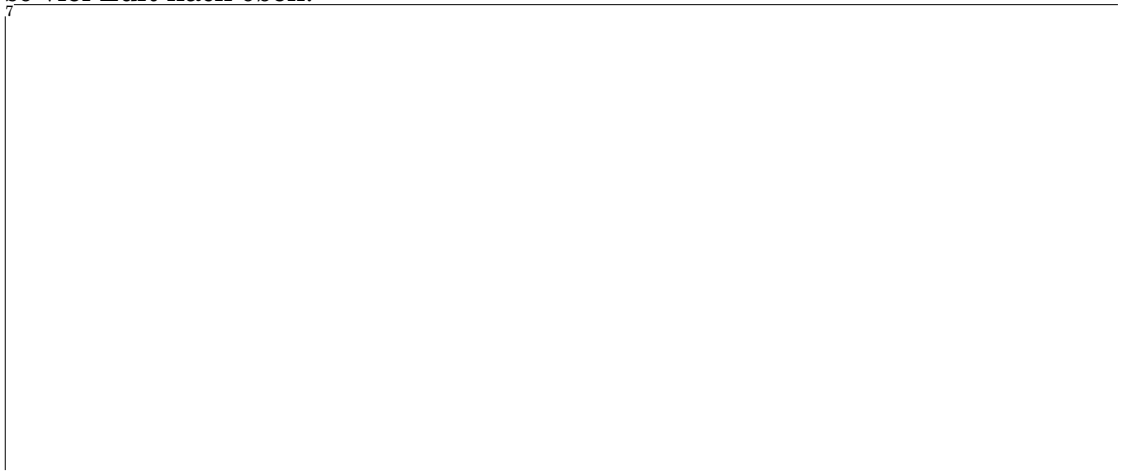
2 Standard-Ganzzahltypen von C

Die grundlegenden ganzzahligen Typen von C, ihre Bit-Länge und ihre Bereiche auf dem MSP430 sind:

[6]


Alle gängigen Systeme verwenden das Zweierkomplement, um negative Zahlen dieser Typen darzustellen.

Alle diese Typen gibt es aber auch ohne negative Zahlen und deshalb mit doppelt so viel Luft nach oben:

[7]


Anders als zum Beispiel Java und C# schreibt C die Größen in Bits nicht genau vor. Man kann die Größen auf dem jeweiligen Compiler mit Hilfe der Konstanten in `<limits.h>` in eigenen Programmen abfragen. Besser noch benutzt man Typen wie `int_least8_t` oder `uint_32_t` aus `<stdint.h>`.

`char` kommt von „character“ (Aussprache meist „tschar“, auch wenn die Herkunft „khar“ nahelegt) und soll ein Zeichen [character] speichern. Der C99-Standard verlangt dafür acht oder mehr Bits. Es sind praktisch überall genau acht Bits. Gefährlich ist das nackte `char`, denn das darf laut C99-Standard ein Vorzeichen haben oder auch nicht. Sicherheitshalber sollte man immer ausdrücklich `signed char` oder `unsigned char` schreiben.

Statt `short` kann man in Langfassung `short int` schreiben, und so weiter.

3 Fehler mit ganzzahligen Typen

Praktisch auf allen Rechnern laufen die ganzzahligen Typen von C über, ohne dass etwas Besonderes passiert, insbesondere kein erzwungener Programmabbruch.

Es werden einfach die oben überstehenden Bits ignoriert. Dies ist ja sowieso der wesentliche Trick der Zweierkomplement-Darstellung. (In der Sprache C# lassen sich Überläufe dagegen abfangen.)

Dieser stille Überlauf führt aber zu übel versteckten Fehlern: Was macht diese Schleife auf dem MSP430?

```
for(int i = 0; i < 1000000; i++)
{
  //...
}
```

Das Teilen durch null führt mit ganzen Zahlen dagegen auf dem meisten Systemen zum Programmabbruch, allerdings nicht auf dem MSP430. Wenn man direkt `0/0` im Programmcode schreibt, meckert schon der Compiler.

Beim Zuweisen an eine weniger breite Variable werden auf praktisch alles Systemen die höherwertigen Bits einfach abgeschnitten – wie beim Überlauf. In Rechenoperationen[c1] wie + und ⋆ werden die Zahlen rechts und links auf gleiches Format gebracht („integer conversion[c2]“):

[c1] jl: Absatz überarbeitet

[c2] jl: promotion

```
unsigned char a = 7;
int b = 100;
int c = a * b / b; // ergibt 7
int d = 10000;
int e = a * d / d; // ergibt nicht 7
```

Nackt angegebene Zahlen wie `100` gelten als `int`, es sei denn, sie sind zu groß. Dann werden sie zu `long` oder `long long`. Gemäß den Mustern `100U` (100, unsigned int) und `-42L` (−42, long) kann man bei konstanten Zahlen einen bestimmten, großen Typ erzwingen.

Rechenoperationen mit kleineren Typen werden auf den meisten Systemem mit der Breite von `int` ausgeführt („integer promotion“). Beispiel:

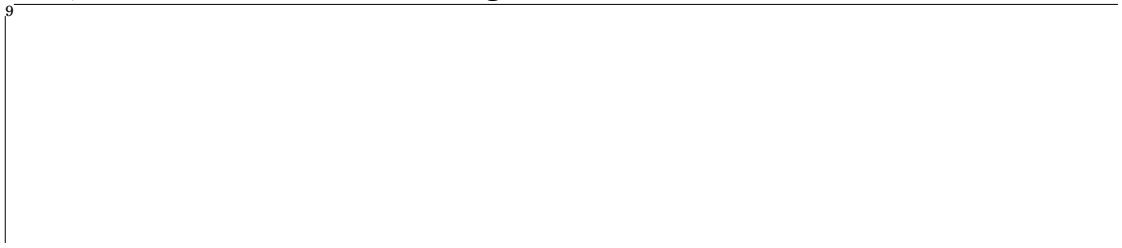
```
signed char a = 7;
signed char b = 100;
int c = a * b; // ergibt 700
int d = 700;
int e = 100;
long f = d * e; // ergibt nicht 70000
```

4 Festkomma und Gleitkomma

Um gebrochene Zahlen darzustellen, kann man sich an einer festen Position ein Dezimalkomma denken. Addition und Subtraktion ändern sich dann nicht:

[8]


Bei Multiplikation und Division erhält man mit Komma die gleichen Ziffern wie ohne, muss aber das Komma im Ergebnis verschieben:

[9]


Dasselbe kann man auch binär machen, zum Beispiel 16 Bits in 8 Bits vor und 8 nach dem Komma zerlegen:

[10]


So etwas ist eine Festkomma-Darstellung [fixed-point representation]. Sie lässt sich recht einfach mit den Ganzzahl-Rechenoperationen von C nachahmen. Eine besondere Unterstützung gibt es dafür aber in der Sprache nicht.

Was C und seine Nachfahren direkt unterstützen, sind dagegen Zahlen in Gleitkomma-Darstellung [floating-point representation]. Die Idee dahinter ist die naturwissenschaftliche Schreibweise von Zahlen mit Zehnerpotenzen. Man schiebt das Komma hinter die am weitesten links stehende Ziffer, die nicht null ist:

[11]


Mit Hilfe des Exponenten kann man Zahlen erzeugen, die extrem groß sind, aber auch Zahlen erzeugen, die sehr dicht an null liegen. Der Exponent lässt das Komma durch die Ziffernfolge gleiten, daher der Name. Die relative (d. h. prozentuale) Genauigkeit bleibt dabei immer gleich. Sie hängt von der Zahl der gültigen Nachkommastellen an.

Dieselbe Idee lässt sich auch mit Binärzahlen durchführen, dann aber mit 2^x statt 10^x, um das Komma der binären Zahl zu verschieben:

[12]

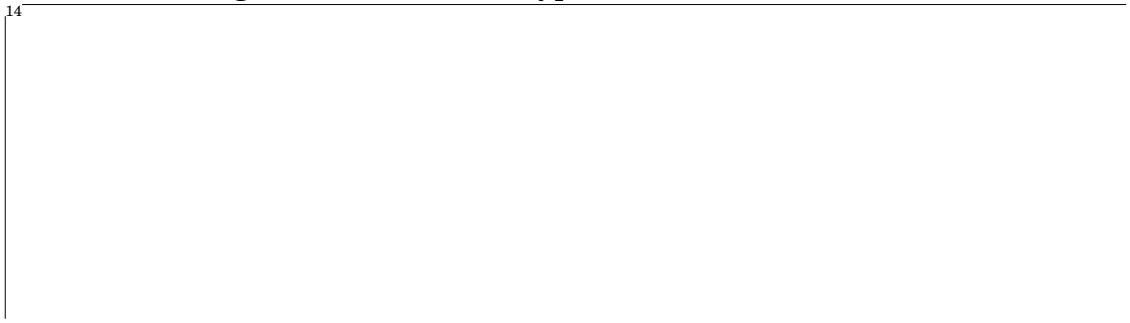

Die erste Stelle vor dem Komma ist dann eine 1. (Technische Feinheit: Das klappt allerdings nicht, wenn die Zahl ist zu dicht bei null ist und man den Exponenten nicht noch negativer machen kann. Dann erhält man sogenannte „denormalized numbers“).

C99 sieht drei Gleitkommatypen vor:

[13]

Der C99-Standard schreibt die Details zwar nicht fest, aber praktisch alle C-Compiler und alle Nachfolgesprachen bieten für `float` und `double` die folgenden Gleitkommatypen aus dem Standard IEEE-754 an:

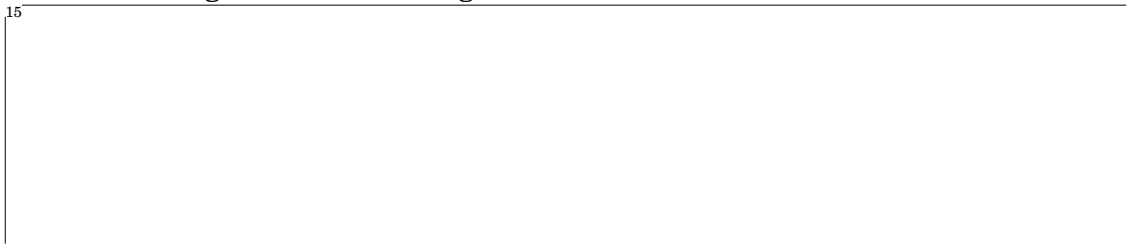
[14]



Diese Angaben lassen sich auch der Datei `<float.h>` entnehmen.

Solche Zahlen gibt man in der folgenden Form ein:

[15]



Wie viele andere Mikrocontroller und anders als die gängigen PC-Prozessoren hat der MSP430 keine Gleitkomma-Recheneinheit. Deshalb muss hier jede Gleitkomma-Operationen mit Ganzzahl-Operationen zusammengestückelt werden, so ähnlich wie man schriftlich addiert und multipliziert. Damit werden aus einer Zeile C Dutzende Zeilen Maschinensprache (Demo).

5 Eingebaute mathematische Funktionen

Die meisten der mathematischen Funktionen in C befassen sich mit Gleitkommawerten. Eine wichtige Funktion gibt es aber für ganze Zahlen: den Absolutbetrag. Für den Typ `int` heißt er `abs`. Die Deklaration dazu findet sich in `<stdlib.h>`.

Die vom Taschenrechner bekannten Funktionen – und viele mehr – für `double`-Zahlen sind in `<math.h>` deklariert. Demo: Überraschende Fehler, wenn man vergisst, diese Datei zu inkludieren. In `<math.h>` finden sich zum Beispiel `sqrt` und `log`. Achtung: Weil diese auf dem MSP430 nicht in Hardware unterstützt sind, sondern mit Ganzzahloperationen ausbuchstabiert werden, reicht oft der Programmspeicherplatz des kleinen MSP430G2231 nicht dafür. Um dennoch damit mit diesen Funktionen zu experimentieren, kann man auf die Softwaresimulation eines größeren MSP430-Typs umschalten.

Die Potenzfunktion `pow(double x, double y)` berechnet x^y, auch in Fällen wie $4{,}2^{0{,}5}$ und $(-4{,}2)^{13}$. Das verlangt einen komplizierten Rechenweg. Wenn man nur ein Quadrat braucht, schreibt man das mit einem Produkt hin, gegebenenfalls als `inline`-Funktion.

Sinus und Freunde arbeiten hier ausschließlich im Bogenmaß. Allerdings ist die Zahl π nicht Teil des C99-Standards.

Der Absolutbetrag für `double`-Zahlen heißt `fabs`, zur Unterscheidung vom Absolutbetrag für ganze Zahlen.

Anders als C++ und seine Nachfolger kann C jeden Funktionsnamen nur einmal vergeben. Deshalb existieren die Funktionen in `<math.h>` alle dreimal mit verschiedenen Namen. So ist `sin` der Sinus für `double`-Werte, `sinf` der Sinus für `float`-Werte und `sinl` der Sinus für `long double`-Werte. In Sprachen wie Java und C# heißen die alle gleich. C99 kann das nur mit einem Trick, den Makros in `<tgmath.h>`.

Bei Zuweisungen von Ganzzahlen an Gleitkommavariable passiert eine automatische Umwandlung. Im umgekehrten Fall, bei Zuweisungen von Gleitkommavariablen an Ganzzahlen, wird ebenfalls automatisch umgewandelt – zumindest in C und C++; das ist aber gefährlich, weil die Nachkommastellen abgeschnitten werden und weil die Gleitkommazahlen zu große für den Wertebereich der Ganzzahlen sein können.

Sobald bei den Grundrechenarten mindestens eine Gleitkommazahl beteiligt ist, wird die Rechnung in Gleitkomma ausgeführt:[16]

Um zu erzwingen, dass auch Ganzzahl/Ganzzahl in Gleitkomma gerechnet wird, kann man Zähler oder Nenner oder bei mit einem vorangestellten `(float)` oder `(double)` zu Gleitkommazahlen machen – in den angegebenen Typ „casten“.

6 Gleitkomma-Typen: Rundungsfehler, verbotene Rechenoperationen

Gleitkommaoperationen sind praktisch immer mit Rundungsfehlern behaftet. Schon die Zahl 0,1 wird im binären Gleitkommaformat nicht exakt dargestellt. Was sollte *eigentlich* aus dieser Schleife herauskommen? (Demo)

```
float sum = 0.0f;
for(int i = 0; i < 10000; i++)
{
  sum += 0.1f;
}
```

Die Genauigkeit von `float` mit 32 Bit reicht für die meisten Anwendungen. Eine der wenigen Ausnahmen sind GPS-Navigationssysteme: Wenn man 10 Meter Distanz auf 40.000 km Erdumfang unterscheiden will, werden die Fehler drastisch:

```
float a = 40000000.0f;
float b = 40000010.0f;
float c = b - a;
```

Fehler einer anderen Art: In komplizierten Rechnungen lässt es sich nicht ausschließen, dass man Rechenoperationen aufruft, die eigentlich nicht erlaubt sind, zum Beispiel die Wurzel aus einer negativen Zahl zieht oder durch eine Zahl nahe null teilt. Netterweise gehen die üblichen Prozessoren damit freundlich um. Der IEEE-754-Standard sieht dafür drei Spezialfälle von Zahlen vor, alle als spezielle Bitmuster in denselben Bits gespeichert wie die üblichen Zahlen:

[17]

Demo mit MSP430: Mit diesen Spezialzahlen wird korrekt weitergerechnet.