S. 26 Mitte

FALSCH:
Wir nehmen willkürlich an, dass der Programmstart sich auf die Speicherposition 0x8000 bezieht.

RICHTIG:
Wir nehmen willkürlich an, dass der Programmstart sich auf die Speicherposition 0x800 bezieht.

S. 28 unten

FALSCH:
Die nächsten Vorgänge werden in einer Schleife so lange wiederholt bis *y* gleich null ist:

RICHTIG:
Die nächsten Vorgänge werden in einer Schleife so lange wiederholt bis *n* gleich null ist:

S. 37 oben
Tabelle 1.6

FALSCH:

Größter relativer Fehler	Genauigkeit	Kleinste positive Zahl	Größte positive Zahl
2^{-24} ca. $6 \cdot 10^{-8}$	ca. 7 Dezimalstellen	2^{-126} ca. $1{,}2 \cdot 10^{-38}$	$(1 - 2^{-23})\ 2^{127}$ ca. $3{,}4 \cdot 10^{38}$

RICHTIG:

Größter relativer Fehler	Genauigkeit	Kleinste positive Zahl[1]	Größte positive Zahl
2^{-24} ca. $6 \cdot 10^{-8}$	ca. 7 Dezimalstellen	2^{-126} ca. $1{,}2 \cdot 10^{-38}$	$(1 - 2^{-23})\ 2^{127}$ ca. $1{,}7 \cdot 10^{38}$

S. 37 unten
Tabelle 1.7

FALSCH:

Größter relativer Fehler	Genauigkeit	Kleinste positive Zahl	Größte positive Zahl
2^{-53} ca. $6 \cdot 10^{-16}$	ca. 16 Dezimalstellen	2^{-1022} ca. $2{,}2 \cdot 10^{-38}$	$(1 - 2^{-52})\ 2^{1023}$ ca. $1{,}8 \cdot 10^{308}$

RICHTIG:

Größter relativer Fehler	Genauigkeit	Kleinste positive Zahl[1]	Größte positive Zahl
2^{-53} ca. $1{,}1 \cdot 10^{-16}$	ca. 16 Dezimalstellen	2^{-1022} ca. $2{,}2 \cdot 10^{-308}$	$(1 - 2^{-52})\ 2^{1023}$ ca. 10^{308}

1 In der Norm IEEE 754 gibt es als Sonderfall noch sogenannte „denormalisierte Zahlen“ zur Darstellung von extrem kleinen Zahlenwerten, auf die wir nicht weiter eingehen werden. Sie sind auch in den Tabellen 1.6 und 1.7 nicht berücksichtigt.

S. 75 oben

FALSCH:
Ist `vdiff` kleiner gleich 60, …

RICHTIG:
Ist `vDiff` kleiner gleich 60, …

S. 130 oben
letzte Tabellenzeile „Weitere Beispiele“

```
7 ^ 1
```

Besser:

```
n = 7 ^ 1
```

S. 130
Mitte unten
Ersten beiden Tabellenzeilen

FALSCH:

n << 4	Die Bits von `n` werden um vier Bit-Positionen nach **links** verschoben. Null-Bits werden dabei nachgeschoben. Entspricht einer Multiplikation mit dem Faktor 24
n >> 4	Die Bits von `n` werden um vier Bit-Positionen nach **rechts** verschoben. Null-Bits werden dabei nachgeschoben. Entspricht einer ganzzahligen Division durch 24

RICHTIG:

n << 4	Die Bits von `n` werden um vier Bit-Positionen nach **links** verschoben. Null-Bits werden dabei nachgeschoben. Entspricht einer Multiplikation mit dem Faktor 2^4
n >> 4	Die Bits von `n` werden um vier Bit-Positionen nach **rechts** verschoben. Null-Bits werden dabei nachgeschoben. Entspricht einer ganzzahligen Division durch 2^4

25.6.2012

S. 175 Mitte

FALSCH:
Insgesamt ergibt sich dann folgendes Bild des Arbeitsspeichers des Rechners:

500		502		504	505				509	50A						510	511
?	?	?	?	48	61	6C	6C	6F	00	?	?	?	?	?	?	?	?
n		**m**		**H**	**a**	**l**	**l**	**o**	**\0**				**p**			**z**	

RICHTIG:
Insgesamt ergibt sich dann folgendes Bild des Arbeitsspeichers des Rechners:

500		502		504	505				509	50A						510	511
?	?	?	?	48	61	6C	6C	6F	00	?	?	?	?	?	?	?	?
n		**m**		**H**	**a**	**l**	**l**	**o**	**\0**					**p**		**z**	

S. 182 oben

FALSCH:

500		502		504		506				50A						510	511
00	02	00	03	00	05	48	61	6C	6C	6F	00	?	?	?	?	05	02
d[0]		d[1]		d[2]		**B**	**e**	l	l	o	\0			p		z	

RICHTIG:

500		502		504		506				50A						510	511
00	02	00	03	00	05	42	65	6C	6C	6F	00	?	?	?	?	05	02
d[0]		d[1]		d[2]		**B**	**e**	l	l	o	\0			p		z	

S. 261 unten	FALSCH: Die Objektdateien „sortiere.o“, „auswahl.o“, „bubble.o“, „quick.o“ werden erzeugt durch: RICHTIG: Die Objektdateien „sortiere.o“, „selection.o“ und „quick.o“ werden erzeugt durch:

Verbesserungen

Die im Studienbuch vorgestellte Implementierung des Quick-Sort-Algorithmus (siehe Studienbuch Seite 255) besitzt bei Datensätzen, die viele identische Zahlenwerte enthalten, ein relativ ungünstiges Laufzeitverhalten. Dies lässt sich durch eine kleine (unscheinbare) Quelltextänderung in der Funktion `partition()` beheben; die geänderten Stellen sind grau hinterlegt.

```
static int partition(int a[], int startIndex, int endIndex)
{
  int i, j, tmp, trenn;    // trenn: Trennelement

  trenn=a[endIndex];  // Trennelement ist stets das letzte Element
  i = startIndex - 1;
  j = endIndex;

  while(1) {
    do {          /* linke Hälfte */
      i++;
    } while(a[i] < trenn);

    do {          // rechte Hälfte
      j--;
    } while( j > startIndex && a[j] > trenn);

    if(i >= j)
      break;   // Suche von rechts trifft die Suche von links
    tmp = a[i];
    a[i] = a[j];
    a[j] = tmp;
  }
  tmp = a[i];          // Trennelement in die richtige Position bringen
  a[i] = a[endIndex];  // i: neue Position des Trennelements
  a[endIndex] = tmp;

  return i;
}
```