Hochschule Bochum
Bochum University
of Applied Sciences

Grundlagen Rechnertechnik

Prof. Dr. Peter Gerwinski

8. Januar 2013

Grundlagen Rechnertechnik

5.6 Byte-Reihenfolge – Endianness

5.6.1 Konzept

Eine Zahl geht über mehrere Speicherzellen.
Beispiel: 16-Bit-Zahl in 2 8-Bit-Speicherzellen

Welche Bits liegen wo?

5.6 Byte-Reihenfolge – Endianness

5.6.1 Konzept

Eine Zahl geht über mehrere Speicherzellen.
Beispiel: 16-Bit-Zahl in 2 8-Bit-Speicherzellen

Welche Bits liegen wo?

$$1027 = 1024 + 2 + 1 = 0000\,0100\,0000\,0011_2 = 0403_{16}$$

5.6 Byte-Reihenfolge – Endianness

5.6.1 Konzept

Eine Zahl geht über mehrere Speicherzellen.
Beispiel: 16-Bit-Zahl in 2 8-Bit-Speicherzellen

Welche Bits liegen wo?

$$1027 = 1024 + 2 + 1 = 0000\,0100\,0000\,0011_2 = 0403_{16}$$

Speicherzellen:

04	03

Big-Endian „großes Ende zuerst“

5.6 Byte-Reihenfolge – Endianness

5.6.1 Konzept

Eine Zahl geht über mehrere Speicherzellen.
Beispiel: 16-Bit-Zahl in 2 8-Bit-Speicherzellen

Welche Bits liegen wo?

$$1027 = 1024 + 2 + 1 = 0000\,0100\,0000\,0011_2 = 0403_{16}$$

Speicherzellen:

04	03

Big-Endian „großes Ende zuerst“
für Menschen leichter lesbar

5.6 Byte-Reihenfolge – Endianness

5.6.1 Konzept

Eine Zahl geht über mehrere Speicherzellen.
Beispiel: 16-Bit-Zahl in 2 8-Bit-Speicherzellen

Welche Bits liegen wo?

$$1027 = 1024 + 2 + 1 = 0000\,0100\,0000\,0011_2 = 0403_{16}$$

Speicherzellen:

04	03

Big-Endian „großes Ende zuerst“
für Menschen leichter lesbar

03	04

Little-Endian „kleines Ende zuerst“

5.6 Byte-Reihenfolge – Endianness

5.6.1 Konzept

Eine Zahl geht über mehrere Speicherzellen.
Beispiel: 16-Bit-Zahl in 2 8-Bit-Speicherzellen

Welche Bits liegen wo?

$$1027 = 1024 + 2 + 1 = 0000\,0100\,0000\,0011_2 = 0403_{16}$$

Speicherzellen:

04	03

Big-Endian „großes Ende zuerst“
für Menschen leichter lesbar

03	04

Little-Endian „kleines Ende zuerst“
bei Additionen effizienter

5.6 Byte-Reihenfolge – Endianness

5.6.1 Konzept

Eine Zahl geht über mehrere Speicherzellen.
Beispiel: 16-Bit-Zahl in 2 8-Bit-Speicherzellen

Welche Bits liegen wo?

$$1027 = 1024 + 2 + 1 = 0000\,0100\,0000\,0011_2 = 0403_{16}$$

Speicherzellen:

04	03

Big-Endian „großes Ende zuerst“
für Menschen leichter lesbar

03	04

Little-Endian „kleines Ende zuerst“
bei Additionen effizienter

$\longrightarrow$ Geschmackssache

5.6 Byte-Reihenfolge – Endianness

5.6.1 Konzept

Eine Zahl geht über mehrere Speicherzellen.
Beispiel: 16-Bit-Zahl in 2 8-Bit-Speicherzellen

Welche Bits liegen wo?

$$1027 = 1024 + 2 + 1 = 0000\,0100\,0000\,0011_2 = 0403_{16}$$

Speicherzellen:

04	03

Big-Endian „großes Ende zuerst“
für Menschen leichter lesbar

03	04

Little-Endian „kleines Ende zuerst“
bei Additionen effizienter

$\longrightarrow$ Geschmackssache
. . . außer bei Datenaustausch!

5.6 Byte-Reihenfolge – Endianness

5.6.1 Konzept

Eine Zahl geht über mehrere Speicherzellen.
Beispiel: 16-Bit-Zahl in 2 8-Bit-Speicherzellen

Welche Bits liegen wo?

$\longrightarrow$ Geschmackssache
. . . außer bei Datenaustausch!

- Dateiformate
- Datenübertragung

5.6 Byte-Reihenfolge – Endianness

5.6.2 Dateiformate

Audio-Formate: Reihenfolge der Bytes in 16- und 32-Bit-Zahlen

- RIFF-WAVE-Dateien (.wav): Little-Endian
- Au-Dateien (.au): Big-Endian

5.6 Byte-Reihenfolge – Endianness

5.6.2 Dateiformate

Audio-Formate: Reihenfolge der Bytes in 16- und 32-Bit-Zahlen

- RIFF-WAVE-Dateien (.wav): Little-Endian
- Au-Dateien (.au): Big-Endian
- ältere AIFF-Dateien (.aiff): Big-Endian
- neuere AIFF-Dateien (.aiff): Little-Endian

5.6 Byte-Reihenfolge – Endianness

5.6.2 Dateiformate

Audio-Formate: Reihenfolge der Bytes in 16- und 32-Bit-Zahlen

- RIFF-WAVE-Dateien (.wav): Little-Endian
- Au-Dateien (.au): Big-Endian
- ältere AIFF-Dateien (.aiff): Big-Endian
- neuere AIFF-Dateien (.aiff): Little-Endian

Grafik-Formate: Reihenfolge der Bits in den Bytes

- PBM-Dateien: Big-Endian
- XBM-Dateien: Little-Endian

5.6 Byte-Reihenfolge – Endianness

5.6.2 Dateiformate

Audio-Formate: Reihenfolge der Bytes in 16- und 32-Bit-Zahlen

- RIFF-WAVE-Dateien (.wav): Little-Endian
- Au-Dateien (.au): Big-Endian
- ältere AIFF-Dateien (.aiff): Big-Endian
- neuere AIFF-Dateien (.aiff): Little-Endian

Grafik-Formate: Reihenfolge der Bits in den Bytes

- PBM-Dateien: Big-Endian, MSB first
- XBM-Dateien: Little-Endian, LSB first

MSB/LSB = most/least significant bit

5.6 Byte-Reihenfolge – Endianness

5.6.3 Datenübertragung

- RS-232 (serielle Schnittstelle): MSB first
- I^2C: LSB first
- USB: beides

5.6 Byte-Reihenfolge – Endianness

5.6.3 Datenübertragung

- RS-232 (serielle Schnittstelle): MSB first
- I^2C: LSB first
- USB: beides

- Ethernet: LSB first
- TCP/IP (Internet): Big-Endian

5.7 Speicherausrichtung – Alignment

5.7 Speicherausrichtung – Alignment

```
#include <stdint.h>

uint8_t a;
uint16_t b;
uint8_t c;
```

5.7 Speicherausrichtung – Alignment

```
#include <stdint.h>

uint8_t a;
uint16_t b;
uint8_t c;
```

Speicheradresse durch 2 teilbar – „16-Bit-Alignment“

- 2-Byte-Operation: effizienter

5.7 Speicherausrichtung – Alignment

```
#include <stdint.h>

uint8_t a;
uint16_t b;
uint8_t c;
```

Speicheradresse durch 2 teilbar – „16-Bit-Alignment“

- 2-Byte-Operation: effizienter
- . . . oder sogar nur dann erlaubt

5.7 Speicherausrichtung – Alignment

```
#include <stdint.h>

uint8_t a;
uint16_t b;
uint8_t c;
```

Speicheradresse durch 2 teilbar – „16-Bit-Alignment“

- 2-Byte-Operation: effizienter
- . . . oder sogar nur dann erlaubt

$\longrightarrow$ Compiler optimiert Speicherausrichtung

5.7 Speicherausrichtung – Alignment

```
#include <stdint.h>

uint8_t a;
uint16_t b;
uint8_t c;
```

Speicheradresse durch 2 teilbar – „16-Bit-Alignment“

- 2-Byte-Operation: effizienter
- . . . oder sogar nur dann erlaubt

⟶ Compiler optimiert Speicherausrichtung

```
uint8_t a;
uint8_t dummy;
uint16_t b;
uint8_t c;
```

5.7 Speicherausrichtung – Alignment

```
#include <stdint.h>

uint8_t a;
uint16_t b;
uint8_t c;
```

Speicheradresse durch 2 teilbar – „16-Bit-Alignment“

- 2-Byte-Operation: effizienter
- . . . oder sogar nur dann erlaubt

⟶ Compiler optimiert Speicherausrichtung

```
uint8_t a;
uint8_t dummy;
uint16_t b;
uint8_t c;
```

```
uint8_t a;
uint8_t c;
uint16_t b;
```

5.7 Speicherausrichtung – Alignment

```
#include <stdint.h>

uint8_t a;
uint16_t b;
uint8_t c;
```

Speicheradresse durch 2 teilbar – „16-Bit-Alignment“

- 2-Byte-Operation: effizienter
- . . . oder sogar nur dann erlaubt

$\longrightarrow$ Compiler optimiert Speicherausrichtung

```
uint8_t a;
uint8_t dummy;
uint16_t b;
uint8_t c;
```

```
uint8_t a;
uint8_t c;
uint16_t b;
```

Fazit:

- Adressen von Variablen sind systemabhängig

5.7 Speicherausrichtung – Alignment

```
#include <stdint.h>

uint8_t a;
uint16_t b;
uint8_t c;
```

Speicheradresse durch 2 teilbar – „16-Bit-Alignment“

- 2-Byte-Operation: effizienter
- . . . oder sogar nur dann erlaubt

$\longrightarrow$ Compiler optimiert Speicherausrichtung

```
uint8_t a;
uint8_t dummy;
uint16_t b;
uint8_t c;
```

```
uint8_t a;
uint8_t c;
uint16_t b;
```

Fazit:

- Adressen von Variablen sind systemabhängig
- Bei Definition von Datenformaten Alignment beachten $\longrightarrow$ effizienter