Teile und Herrsche

Programmieren für Ingenieure
Sommer 2014

Andreas Zeller, Universität des Saarlandes

1

Vom Programm zum Prozessor

Prüfen und Übersetzen

Hochladen über USB

Funktionsaufrufe

- Die meisten Funktionen haben *Parameter*, die ihre Funktionsweise bestimmen

```
digitalWrite(pin_number, value)
```

- Beim Aufruf muss für jeden Parameter ein Wert *(Argument)* angegeben werden

```
digitalWrite(13, HIGH);
```

Funktionsname — Wert für *pin_number* — Wert für *value*

Variablen

- *Variablen* dienen dazu, *Werte* zu speichern.
- Mit der Anweisung

```
int led = 13;
```

wird `led` als eine *Variable* eingeführt, die mit dem Wert 13 belegt ist.

- Nach der Anweisung steht `led` stellvertretend für den Variablenwert

Symbolisches Blinken

```
// Pin 13 has an LED connected on most
// Arduino boards.  Give it a name:
int led = 13;

void setup() {
  pinMode(led, ~~HIGH~~, OUTPUT
}

void loop() {
    digitalWrite(led, HIGH);
    delay(1000);
    digitalWrite(led, LOW);
    delay(1000);
}
```

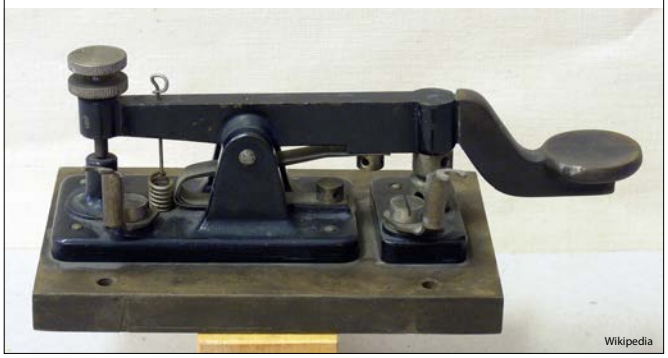
2

Themen heute

- Eigene Funktionen
- Parameter
- Fallunterscheidungen
- Fehlersuche

3

Morse-Code



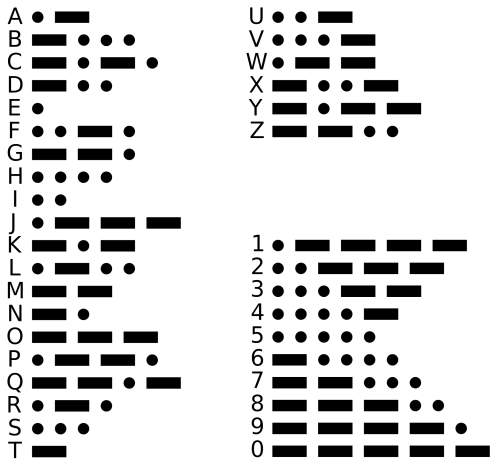
4

Morse-Code

Besteht aus drei Symbolen:

- Punkt (*Dit*)
- Strich (*Dah*)
- Schweigen

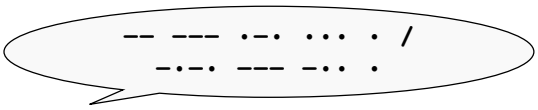
5



6

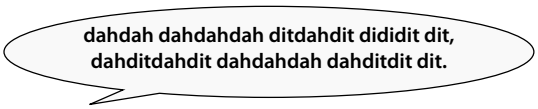
Der Code ist so angelegt, dass häufige Buchstaben kurze Codes haben

Morse-Code



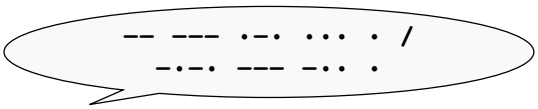
7

Morse-Code



8

Morse-Code



MORSE CODE

9

Morse-Code

- Ein *Dah* ist dreimal so lang wie ein *Dit*.
- Die Pause zwischen zwei gesendeten Symbolen ist ein *Dit* lang.
- Zwischen Buchstaben in einem Wort wird eine Pause von der Länge eines *Dah* (oder drei *Dits*) eingeschoben.
- Die Länge der Pause zwischen Wörtern entspricht sieben *Dits*.

Wikipedia

10

● ● ● ● ● ● ● ● ● ● ●

```
int dit_delay = 500;           // length of a dit in ms

void loop() {
    // send a dit
    digitalWrite(led, HIGH);
    delay(dit_delay);

    digitalWrite(led, LOW);
    delay(dit_delay);
}
```

11

● ▬ ● ▬ ● ▬ ● ▬ ● ▬

```
int dit_delay = 500;           // length of a dit in ms
int dah_delay = dit_delay * 3; // length of a dah in ms

void loop() {
    // send a dit
    digitalWrite(led, HIGH);
    delay(dit_delay);

    digitalWrite(led, LOW);
    delay(dit_delay);

    // send a dah
    digitalWrite(led, HIGH);
    delay(dah_delay);

    digitalWrite(led, LOW);
    delay(dit_delay);
}
```

Berechnung!

12

Arithmetische Operatoren

13

In aufsteigender Bindung:

1. Addition (+), Subtraktion (–)
   Assoziativität: von links nach rechts
2. Multiplikation (*), Division (/), Modulus (%)
   Assoziativität: von links nach rechts
3. Vorzeichen (+, –)
   Assoziativität: von rechts nach links

```
int y = -3 + 7 % 3
int y = (-3) + (7 % 3)
```

Bevor wir aber etwas berechnen, müssen wir erst einmal dafür sorgen, dass wir es ausgeben…

Eigene Funktionen

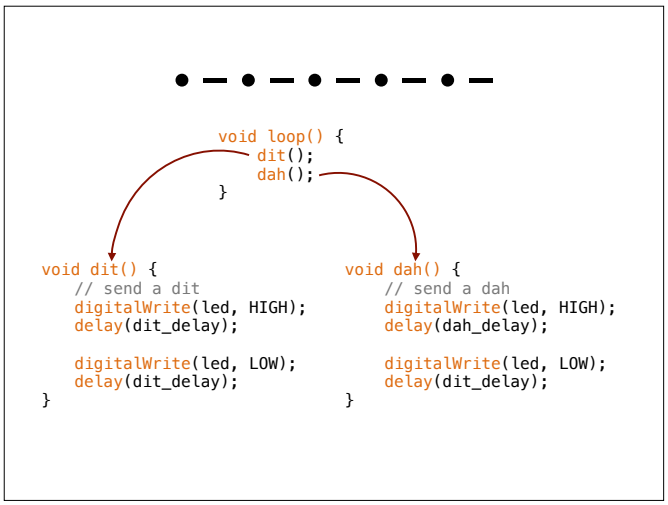
14

- Wir wollen die Anweisungen für *Dahs* und *Dits* in *eigene Funktionen* zusammenfassen
- Eine eigene Funktion wird wie setup() und loop() als *Folge von Anweisungen* definiert:

```
void name() {
   Anweisung 1;
   Anweisung 2;
   …
}
```

15


```
int dit_delay = 500;           // length of a dit in ms
int dah_delay = dit_delay * 3; // length of a dah in ms

void loop() {
    // send a dit
    digitalWrite(led, HIGH);
    delay(dit_delay);

    digitalWrite(led, LOW);
    delay(dit_delay);

    // send a dah
    digitalWrite(led, HIGH);
    delay(dah_delay);

    digitalWrite(led, LOW);
    delay(dit_delay);
}
```



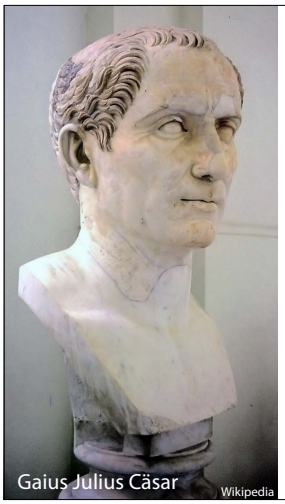
16

loop() **ruft** dit() und dah() **auf**

Demo

17

- dit() und dah()

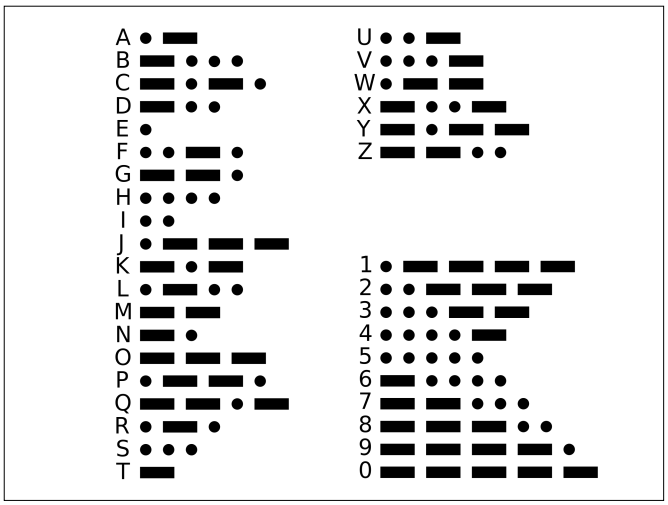


Teile und Herrsche

- Idee: Ein Problem in (kleinere) Teilprobleme zerlegen
- Prinzip politischen Handelns
- Grundprinzip der Informatik

18

Das Problem, längere Texte zu übermitteln, haben wir in die **Teilprobleme** dit() und dah() aufgeteilt, die ihre Arbeit wieder auf andere Funktionen aufteilen.



19

deditditdit dedahdahdah

```
int dit_delay = 500;           // length of a dit in ms
int dah_delay = dit_delay * 3; // length of a dah in ms


void dit() {
    // send a dit
    digitalWrite(led, HIGH);
    delay(dit_delay);

    digitalWrite(led, LOW);
    delay(dit_delay);
}
```

```
void dah() {
    // send a dah
    digitalWrite(led, HIGH);
    delay(dah_delay);

    digitalWrite(led, LOW);
    delay(dit_delay);
}
```

20

S senden

```
void morse_S() {
    dit();
    dit();
    dit();
}
```

oder (kürzer)

```
void morse_S() {
    dit(); dit(); dit();
}
```

21

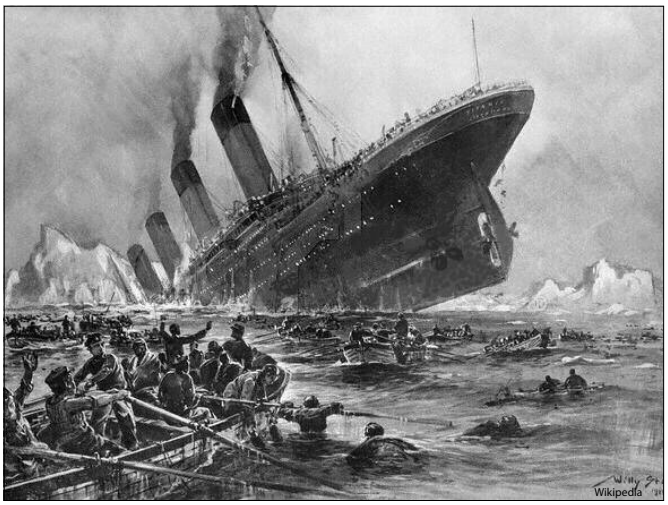
22

Save Our Souls

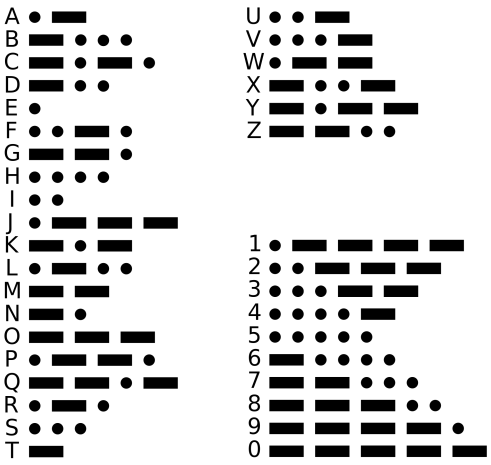
```
void morse_S() {
    dit(); dit(); dit();
}

void morse_O() {
    dah(); dah(); dah();
}
void morse_SOS() {
    morse_S(); morse_O(); morse_S();
    delay(dit_delay * 6);
}
```

•••−−−•••/•••−−−•••/•••−−−•••

23



24



25

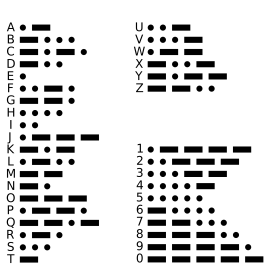
A	morse_A()	U	morse_U()
B	morse_B()	V	morse_V()
C	morse_C()	W	morse_W()
D	morse_D()	X	morse_X()
E	morse_E()	Y	morse_Y()
F	morse_F()	Z	morse_Z()
G	morse_G()		
H	morse_H()		
I	morse_I()		
J	morse_J()		
K	morse_K()	1	morse_1()
L	morse_L()	2	morse_2()
M	morse_M()	3	morse_3()
N	morse_N()	4	morse_4()
O	morse_O()	5	morse_5()
P	morse_P()	6	morse_6()
Q	morse_Q()	7	morse_7()
R	morse_R()	8	morse_8()
S	morse_S()	9	morse_9()
T	morse_T()	0	morse_0()

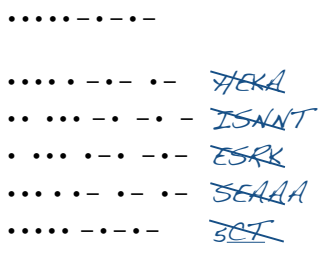
26

SINK

```
void morse_S() {
    dit(); dit(); dit();
}

void morse_I() {
    dit(); dit();
}

void morse_SINK() {
    morse_S(); morse_I(); morse_N(); morse_K();
     •••       ••        −•      −•−
}
```

27





Der Code kann verschieden dekodiert werden

CT ohne Pause ist ein Steuerzeichen; es steht für “Commencing Transmission”.

28

Si tacuisses

```
void morse_S() {
    dit(); dit(); dit();
    pause_letter();
}

void morse_I() {
    dit(); dit();
    pause_letter();
}

void morse_SINK() {
    morse_S(); morse_I(); morse_N(); morse_K();
    pause_word();
}
```

••• •• –• –•–

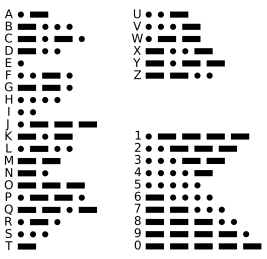
29

Si tacuisses

```
int dit_delay = 500;           // length of a dit in ms
int dah_delay = dit_delay * 3; // length of a dah in ms


// dit() and dat() already include dit_delay
int letter_delay = dah_delay - dit_delay;

// letters already include letter delay
int word_delay   = dit_delay * 7 - letter_delay;


void pause_letter() {
   delay(letter_delay);
}
```

```
void pause_word() {
   delay(word_delay);
}
```

30



••• •• –• –•– SINK

CQD CQD SOS Titanic Position 41.44 N 50.24 W. Require immediate assistance. Come at once. We struck an iceberg. Sinking

Quelle Audio: https://www.youtube.com/watch?v=snkwsU98QlQ

Demo

31

- dit(), dah(), pause_word(), pause_letter()

32

Heute hat man natürlich Sprechfunk – der bringt aber seine eigenen Probleme.
So was kann mit Morse Code nicht passieren!

Eigene Parameter

33

- Ziel: Eine Funktion send_number(*n*), die eine gegebene Zahl *n* per Morse ausgibt
- *n* wird zum *Parameter* der Funktion

Eigene Parameter

- Parameter werden (mitsamt Typen) bei der Definition in Klammern angegeben

```
void name(int p1, int p2, ...) {
   Anweisungen…;
}
```

- Bei uns also:

```
void morse_number(int n) {
   Anweisungen…;
}
```


Fallunterscheidung

- Je nach Wert von *n* müssen unterschiedliche Anweisungen ausgeführt werden:
  - Ist *n* = 1, dann • – – – – senden
  - Ist *n* = 2, dann • • – – – senden
  - usw.

Fallunterscheidung

- Die *if-Kontrollstruktur* dient zum Programmieren von *Fallunterscheidungen:*

```
if (Bedingung) {
   Anweisungen…;
}
```

- Die Anweisungen werden *nur* ausgeführt, wenn die Bedingung erfüllt ist.

Vergleichsoperatoren

In aufsteigender Bindung:

1. Logisches Oder ∨ (| |)
2. Logisches Und ∧ (&&)
3. Größenvergleiche (<, >, <=, >=)
4. Gleichheit = (==), Ungleichheit ≠ (!=)
5. Logisches Nicht ¬ (!)

==, nicht = !

```
if (x >= y && !(x == y))
```

Fallunterscheidung

- Je nach Wert von *n* müssen unterschiedliche Anweisungen ausgeführt werden:
  - Ist *n* = 1, dann • – – – – senden
  - Ist *n* = 2, dann • • – – – senden
  - usw.

Fallunterscheidung


```
// send n in morse code
void morse_digit(int n) {
```

Fallunterscheidung


```
// send n in morse code
void morse_digit(int n) {
    if (n == 0) {
        dah(); dah(); dah(); dah(); dah();
    }
```

Fallunterscheidung


```
// send n in morse code
void morse_digit(int n) {
    if (n == 0) {
        dah(); dah(); dah(); dah(); dah();
    }
    if (n == 1) {
        dit(); dah(); dah(); dah(); dah();
    }
```



```
// send n in morse code
void morse_digit(int n) {
    if (n == 0) {
        dah(); dah(); dah(); dah(); dah();
    }
    if (n == 1) {
        dit(); dah(); dah(); dah(); dah();
    }
    if (n == 2) {
        dit(); dit(); dah(); dah(); dah();
    }
    // usw. für 3–8
    if (n == 9) {
        dah(); dah(); dah(); dah(); dit();
    }
    pause_letter();
}
```

Aufruf

- Einmal definiert, wird morse_digit() wie jede andere Funktion aufgerufen:

```
void morse_digit(int n) {
    // wie oben
}

void loop() {
    morse_digit(5);
    morse_digit(0);
    morse_digit(2);
    morse_digit(4);
}
```

Position 41.44 N 50.24 W

Demo

Von Ziffern zu Zahlen

- Wie geben wir mehrstellige Zahlen aus?
- Ziel: Funktion morse_number(*n*), die *n* durch Aufrufe an morse_digit() ausgibt

```
morse_number(5024) →
    morse_digit(5)
    morse_digit(0)
    morse_digit(2)
    morse_digit(4)
```

Von Ziffern zu Zahlen

46

- Beobachtung: Will ich 5024 ausgeben, kann ich 502 ausgeben, gefolgt von 4.

```
morse_number(5024) →
    morse_number(502)
    morse_digit(4)
```

- Um 502 auszugeben, kann ich 50 ausgeben, gefolgt von 2.

```
morse_number(502) →
    morse_number(50)
    morse_digit(2)
```

Von Ziffern zu Zahlen

47

Allgemeines Prinzip:

1. Hat n mehr als eine Ziffer (d.h. $n \geq 10$) , gebe ich zuerst n / 10 aus
2. Anschließend gebe ich die letzte Ziffer aus (d.h. n mod 10)

48

Demo

Von Ziffern zu Zahlen

So sieht morse_number() aus:

```
void morse_number(int n) {
    if (n >= 10) {
        morse_number(n / 10);
    }
    morse_digit(n % 10);
}
```


Von Ziffern zu Zahlen

```
void morse_number(int n) {
    if (n >= 10) {
        morse_number(n / 10);
    }
    morse_digit(n % 10);
}
```

```
morse_number(5024)
    → morse_number(502)
        → morse_number(50)
            → morse_number(5)
                → morse_digit(5)    • • • • •
            → morse_digit(0)        – – – – –
        → morse_digit(2)            • • – – –
    → morse_digit(4)                • • • • –
```


Demo

Rekursion

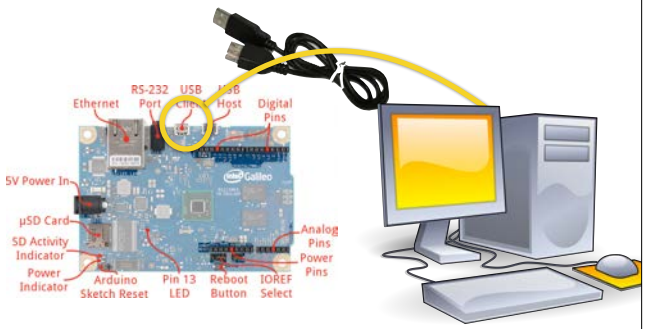
- Ruft eine Funktion sich selbst erneut auf, nennt man dies *Rekursion*
- Jede Berechnung lässt sich durch ausschließlich *Funktionen*, *Bedingungen* und *Rekursion* ausdrücken
- Alles, was man (irgendwie) berechnen kann, können Sie jetzt programmieren

(im Prinzip jedenfalls)

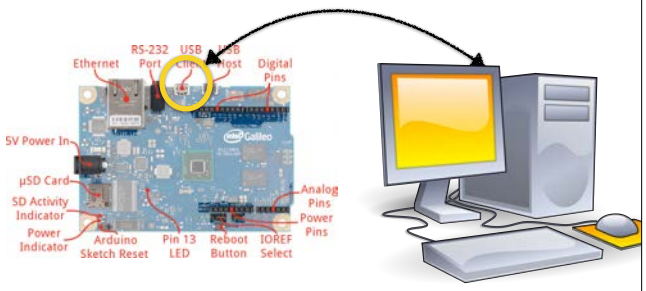
Fehlersuche

- Während einer komplexen Berechnung ist es hilfreich, zu verfolgen, was geschieht
- Hierfür dient die *serielle Ausgabe* der Arduino-Plattform

USB-Anschluss



Datenübertragung



Verfolgen über Werkzeuge → Serieller Monitor

Serial.begin()

- Serial.begin(*baud*) richtet die *serielle Schnittstelle* ein, um mit Geschwindigkeit *baud* (bits/s) Daten zu übertragen
- Beispiel:

```
void setup() {
   // Transfer at 9600 bits/s
   Serial.begin(9600);
}
```


Serial.print()

- Die Funktion Serial.print(*x*) gibt *x* auf der *seriellen Schnittstelle* aus
- Serial.println(x): Genauso, aber mit Zeilenende
- Beispiel:

```
void morse_number(int n) {
   Serial.println(n);
   Anweisungen…;
}
```

Text ausgeben

- Mit Serial.print() und Serial.println() kann man auch *Text* ausgeben
- Text wird in "…" eingeschlossen
- Beispiel:

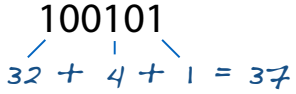
```
void morse_number(int n) {
   Serial.print("n = ");
   Serial.println(n);
   Anweisungen…;
}
```

Demo

- Logging von morse_number() und morse_digit()

Binärzahlen

- Rechner stellen Daten intern als *Bits* dar – nur 0 und 1
- Zahlen werden im *Binärsystem* gespeichert
- Die Zahl 37 etwa wird gespeichert als



Von Ziffern zu Zahlen

61

So sieht morse_number() aus:

```
void morse_number(int n) {
    if (n >= 10) {
        morse_number(n / 10);
    }
    morse_digit(n % 10);
}
```

Kann ich auch eine andere Zahlenbasis als 10 nehmen?

Zahlen binär morsen

62

Um Zahlen im Binärsystem zu morsen:

```
void morse_binary(int n) {
    if (n >= 2) {
        morse_number(n / 2);
    }
    morse_digit(n % 2);
}
```

Demo

63

- Logging von morse_number() und morse_digit()
- Binäre Ausgabe verfolgen

Basis 10 und 2

```
void morse_decimal(int n) {
    if (n >= 10) {
        morse_decimal(n / 10);
    }
    morse_digit(n % 10);
}

void morse_binary(int n) {
    if (n >= 2) {
        morse_binary(n / 2);
    }
    morse_digit(n % 2);
}
```

64

Beliebige Zahlenbasis

Zahlen in Basis *base* ausgeben:

```
void morse_number(int n, int base) {
    if (n >= base) {
        morse_number(n / base, base);
    }
    morse_digit(n % base);
}
```

65

Demo

66

- Zahlenbasis 2, 8, 10
- Verfolgen über serielle Schnittstelle

Ausblick

- Zuweisungen
- Eigene Schleifen
- Verkehrssteuerung
- Eingabeelemente

67

Eigene Funktionen

```
void loop() {
    dit();
    dah();
}

void dit() {
  // send a dit
  digitalWrite(led, HIGH);
  delay(dit_delay);

  digitalWrite(led, LOW);
  delay(dit_delay);
}
```

```
void dah() {
  // send a dah
  digitalWrite(led, HIGH);
  delay(dah_delay);

  digitalWrite(led, LOW);
  delay(dit_delay);
}
```

Eigene Parameter

- Parameter werden (mitsamt Typen) bei der Definition in Klammern angegeben

```
void name(int p1, int p2, ...) {
    Anweisungen…;
}
```

- Bei uns also:

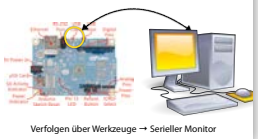
```
void morse_number(int n) {
    Anweisungen…;
}
```

Rekursion

```
void morse_number(int n) {
    if (n >= 10) {
        morse_number(n / 10);
    }
    morse_digit(n % 10);
}
```

```
morse_number(5024)
    → morse_number(502)
        → morse_number(50)
            → morse_number(5)
                → morse_digit(5)    • • • • •
            → morse_digit(0)        – – – – –
        → morse_digit(2)            • • – – –
    → morse_digit(4)                • • • • –
```

Ablauf verfolgen

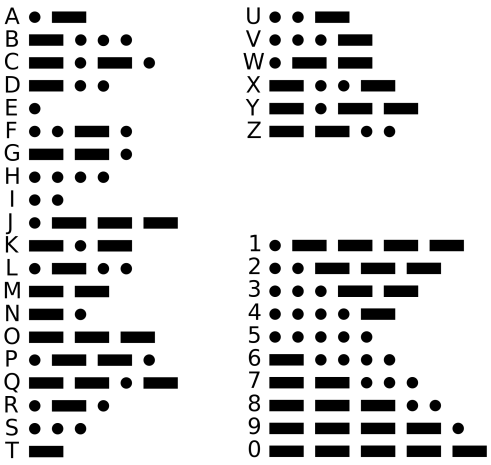


Verfolgen über Werkzeuge → Serieller Monitor

68

Handouts

69



Eigene Funktionen

- Wir wollen die Anweisungen für *Dahs* und *Dits* in *eigene Funktionen* zusammenfassen
- Eine eigene Funktion wird wie setup() und loop() als *Folge von Anweisungen* definiert:

```
void name() {
   Anweisung 1;
   Anweisung 2;
   …
}
```


ditditdit dahdahdah

```
int dit_delay = 500;           // length of a dit in ms
int dah_delay = dit_delay * 3; // length of a dah in ms

// dit() and dat() already include dit_delay
int letter_delay = dah_delay - dit_delay;

// letters already include letter delay
int word_delay   = dit_delay * 7 - letter_delay;

void dit() {
    // send a dit
    digitalWrite(led, HIGH);
    delay(dit_delay);

    digitalWrite(led, LOW);
    delay(dit_delay);
}

void dah() {
    // send a dah
    digitalWrite(led, HIGH);
    delay(dah_delay);

    digitalWrite(led, LOW);
    delay(dit_delay);
}

void pause_letter() {
   delay(letter_delay);
}

void pause_word() {
   delay(word_delay);
}
```

SINK

```
void morse_S() {
    dit(); dit(); dit();
    pause_letter();
}

void morse_I() {
    dit(); dit();
    pause_letter();
}

void morse_SINK() {
    morse_S(); morse_I(); morse_N(); morse_K();
    pause_word();
}
```

••• •• –• –•–

73

Eigene Parameter

- Parameter werden (mitsamt Typen) bei der Definition in Klammern angegeben

```
void name(int p1, int p2, ...) {
    Anweisungen…;
}
```

- Bei uns also:

```
void morse_number(int n) {
    Anweisungen…;
}
```

74

```
// send n in morse code
void morse_digit(int n) {
    if (n == 0) {
        dah(); dah(); dah(); dah(); dah();
    }
    if (n == 1) {
        dit(); dah(); dah(); dah(); dah();
    }
    if (n == 2) {
        dit(); dit(); dah(); dah(); dah();
    }
    // usw. für 3–8
    if (n == 9) {
        dah(); dah(); dah(); dah(); dit();
    }
    pause_letter();
}
```

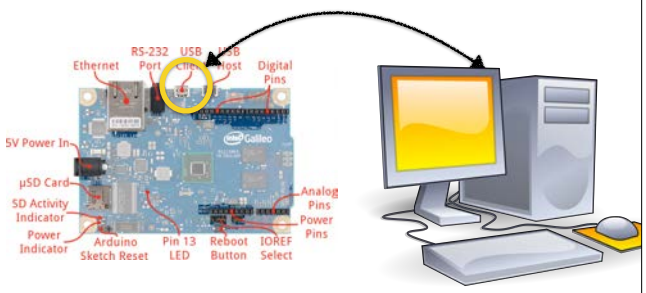
75

Rekursion

```
void morse_number(int n) {
    if (n >= 10) {
        morse_number(n / 10);
    }
    morse_digit(n % 10);
}
```

```
morse_number(5024)
    → morse_number(502)
        → morse_number(50)
            → morse_number(5)
                → morse_digit(5)    • • • • •
            → morse_digit(0)        – – – – –
        → morse_digit(2)            • • – – –
    → morse_digit(4)                • • • • –
```


Ablauf verfolgen



Verfolgen über Werkzeuge → Serieller Monitor