Hinter den Kulissen

Programmieren für Ingenieure
Sommer 2015

Andreas Zeller, Universität des Saarlandes

Klausur

Nachklausur

Hinter den Kulissen

Programmieren für Ingenieure
Sommer 2015

Andreas Zeller, Universität des Saarlandes

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…;
}
```

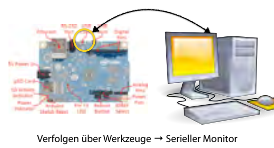
- Bei uns also:

```
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

Themen heute

- Variablen
- Zuweisungen
- Maschinenmodelle

Werte speichern

- Während Berechnungen wollen wir Ergebnisse *speichern*
- Wir wollen Ergebnis einer Variablen *zuweisen.*

Zuweisung

- Die Anweisung

```
name = wert
```

bewirkt, dass die Variable *name* den neuen Wert *wert* hat.

- Im weiteren Programmablauf liefert jeder spätere Zugriff auf die Variable den Wert *wert* (bis zur nächsten Zuweisung)

Zuweisung

```
int x = 0;

x = 1;
Serial.println(x);   - gibt 1 aus

x = 2;
Serial.println(x);   - gibt 2 aus

if (x > 2) {
   x = -1;
}
if (x > 1) {
   x = 100;
}
Serial.println(x);   - gibt 100 aus
```

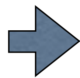
Fibonacci

```
int a   = 0;
int b   = 1;
int sum = 0;

void loop() {
    sum = a + b;
    Serial.println(sum);
    delay(1000);
    b = a;
    a = sum;
}
```

– gibt 1 2 3 5 8 13 … aus

Was passiert hier?

```
int a   = 0;
int b   = 1;
int sum = 0;

void loop() {
    sum = a + b;
    Serial.println(sum);
    delay(1000);
    b = a;
    a = sum;
}
```



Wir haben das Programm, und wir haben den Rechner. Wie führt der Rechner das Programm aus?

Ihr Rechner

Ihr Rechner

Intel® Quark SoC X1000 Application Processor

https://communities.intel.com/docs/DOC-21822

Ihr Rechner

DDR3 x8 | 256 MB | DDR3 x8

Speicher
enthält *Daten*

Intel® Quark SoC X1000 Application Processor

Prozessor
führt *Anweisungen* aus

Ihr Rechner

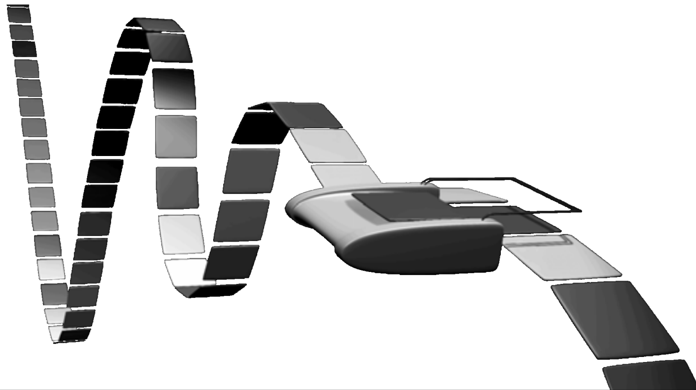
0	I	I	0	I	0	0	0	I	...

Speicher

Prozessor

Wir können noch weiter abstrahieren

Turingmaschine



Eine Turingmaschine ist ein wichtiges Rechnermodell der Theoretischen Informatik. Sie modelliert die Arbeitsweise eines Computers auf besonders einfache und mathematisch gut zu analysierende Weise. Eine Berechnung besteht dabei aus schrittweisen Manipulationen von Symbolen bzw. Zeichen, die nach bestimmten Regeln auf ein Speicherband geschrieben und auch von dort gelesen werden. Ketten dieser Symbole können verschieden interpretiert werden, unter anderem als Zahlen. Damit

Ein Turing-Programm

aktueller Zustand	gelesenes Symbol		schreibe Symbol	neuer Zustand	Kopf-richtung
s1	1	→	0	s2	R
s1	0	→	0	s6	0
s2	1	→	1	s2	R
s2	0	→	0	s3	R
s3	1	→	1	s3	R
s3	0	→	1	s4	L
s4	1	→	1	s4	L
s4	0	→	0	s5	L
s5	1	→	1	s5	L
s5	0	→	1	s1	R

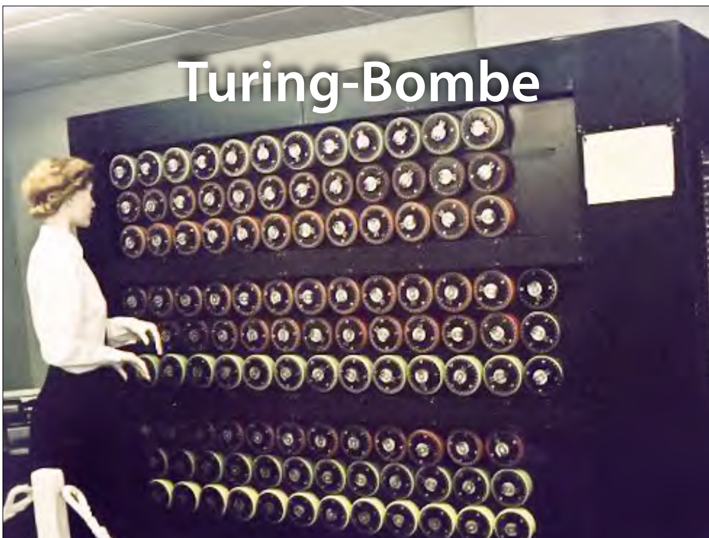
Verdoppelt die Anzahl der Einsen auf dem Band (Quelle: Wikipedia)

Alan Turing

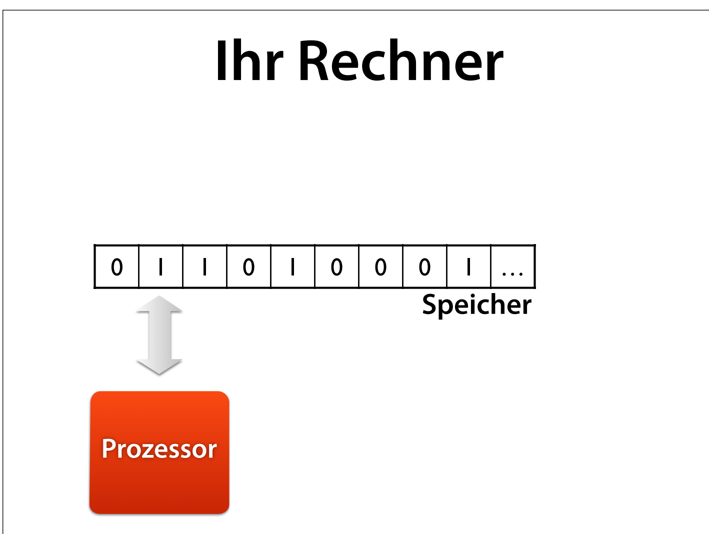
1912–1954

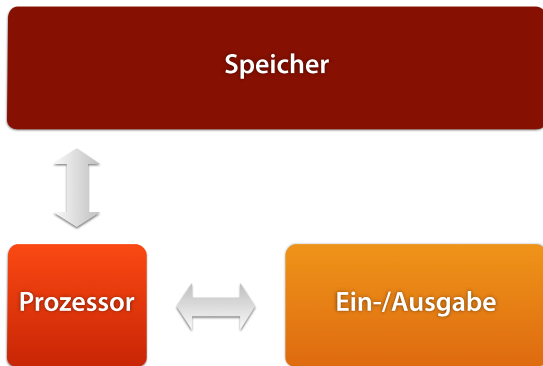
Alan Mathison Turing (* 23. Juni 1912 in London; † 7. Juni 1954 in Wilmslow, Cheshire) war ein britischer Logiker, Mathematiker, Kryptoanalytiker und Informatiker. Er gilt heute als einer der einflussreichsten Theoretiker der frühen Computerentwicklung und Informatik. Turing schuf einen großen Teil der theoretischen Grundlagen für die moderne Informations- und Computertechnologie. Das von ihm entwickelte Berechenbarkeitsmodell der Turingmaschine bildet eines der Fundamente der theoretischen Informatik. Während des Zweiten Weltkrieges war er maßgeblich an der Entzifferung der mit der Enigma verschlüsselten deutschen Funksprüche beteiligt. Nach ihm benannt sind der Turing Award, die bedeutendste Auszeichnung in der Informatik, sowie der Turing-Test zum Nachweis künstlicher Intelligenz. (Wikipedia)

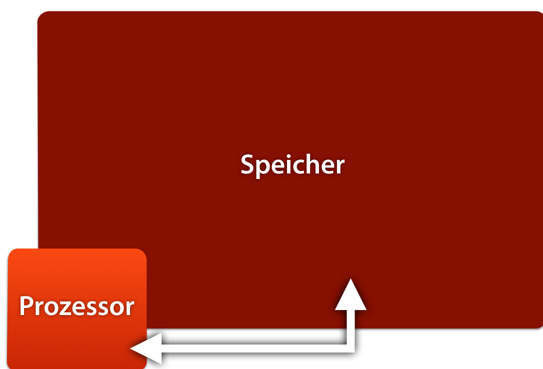
Die ENIGMA (griechisch αἴνιγμα ainigma „Rätsel“) ist eine Rotor-Schlüsselmaschine, die im Zweiten Weltkrieg zur Verschlüsselung des Nachrichtenverkehrs des deutschen Militärs verwendet wurde. Trotz mannigfaltiger Verbesserungen der Verschlüsselungsqualität der Maschine vor und während des Krieges, gelang es den Alliierten mit hohem Aufwand zur Entzifferung, die deutschen Funksprüche nahezu kontinuierlich zu brechen. (Wikipedia)

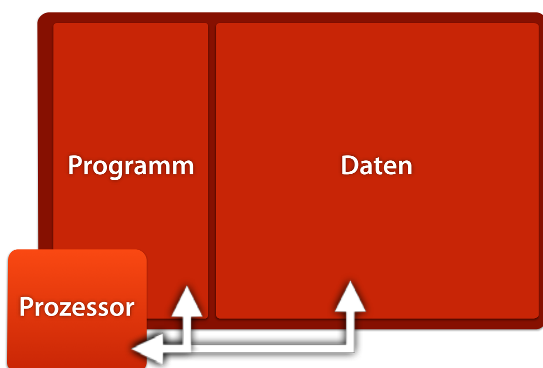


Mit Hilfe der Turing-Bombe (hier ein Nachbau in Bletchley Park, bedient von einer „Wren“) schrumpft der vorher noch so gigantisch erscheinende Schlüsselraum von etwa $2 \cdot 10^{23}$ Möglichkeiten auf vergleichsweise winzige 120·17.576 = 2.109.120 (gut zwei Millionen) Möglichkeiten (etwa 21 bit), eine Zahl, die auch bereits zu Zeiten des Zweiten Weltkriegs mithilfe der damaligen elektromechanischen Technik exhaustiv (erschöpfend) abgearbeitet werden konnte. (Wikipedia)

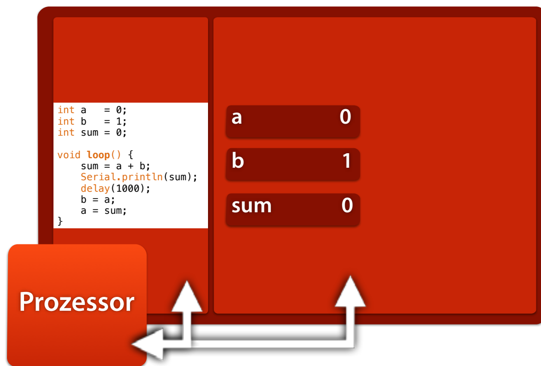



Ihr Rechner
Speicher
Prozessor
Ein-/Ausgabe

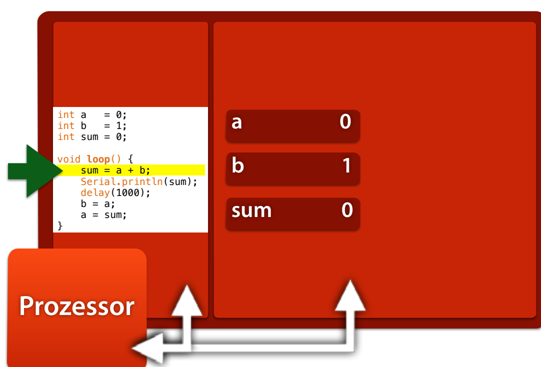

Der Speicher
Speicher
Prozessor


Der Speicher
Programm
Daten
Prozessor

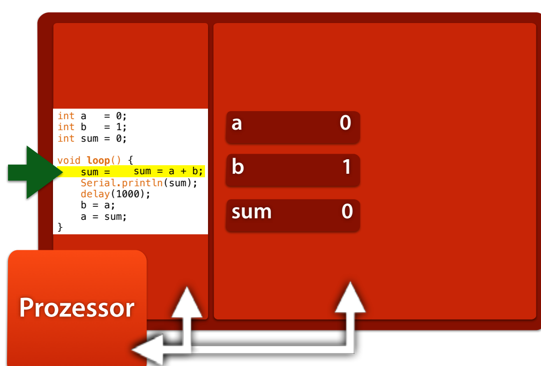
Der Speicher



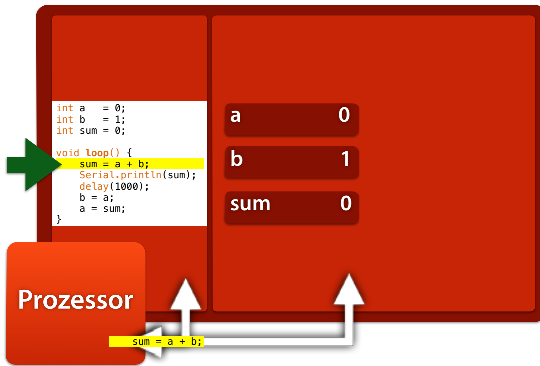
Der Programmzähler



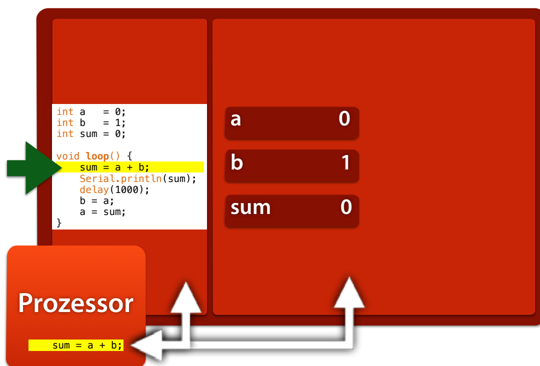
Befehle lesen



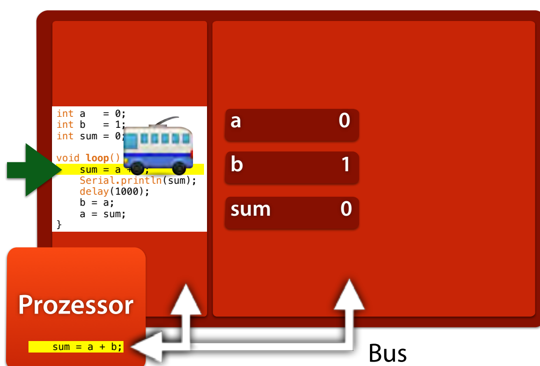
Befehle lesen



Befehle lesen

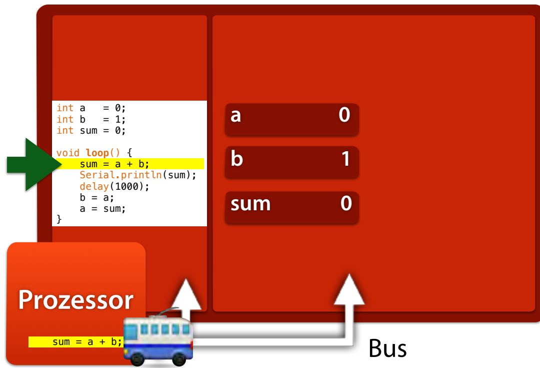


Der Bus

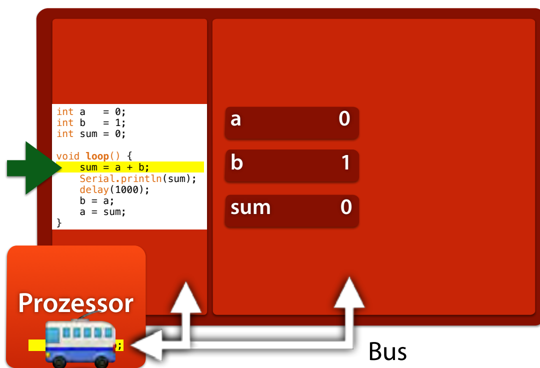


Der Bus transportiert Anweisungen in den Prozessor…

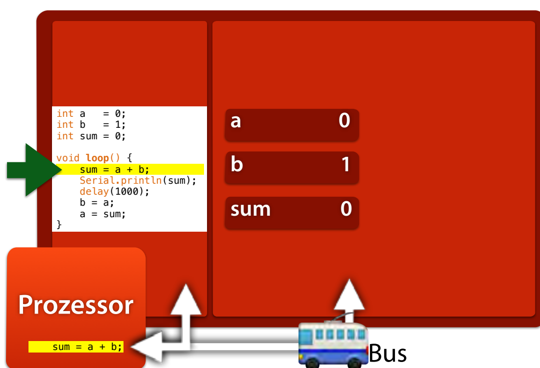
Der Bus



Der Bus

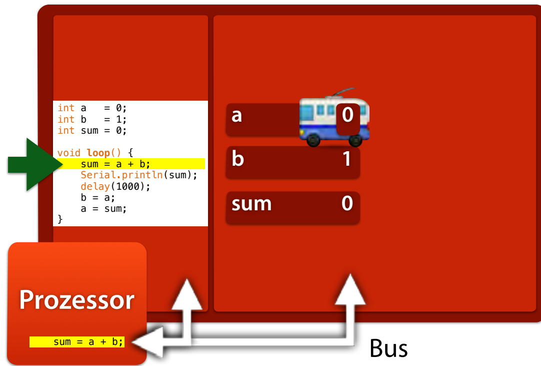


Der Bus

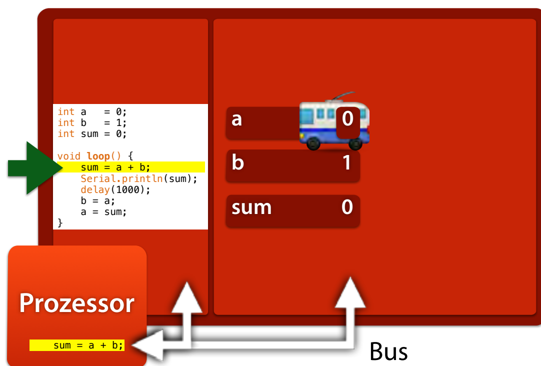


…holt und schreibt aber auch Daten in den Speicher.

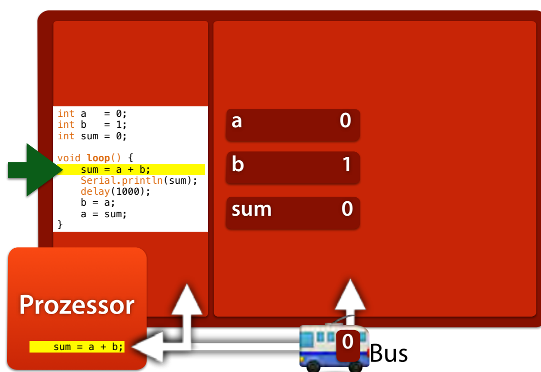
Der Bus



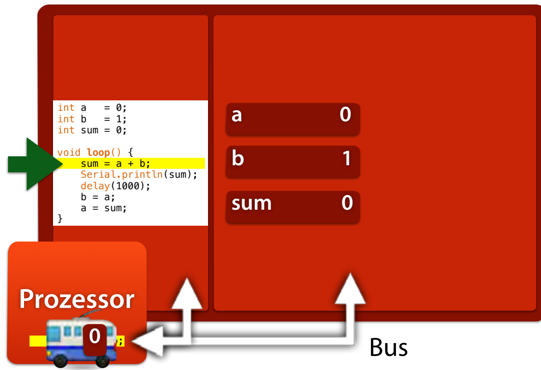
Der Bus



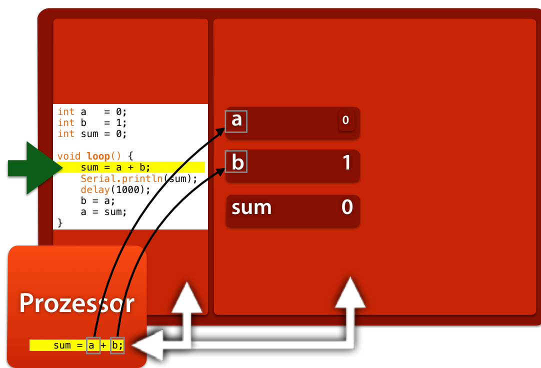
Der Bus



Der Bus

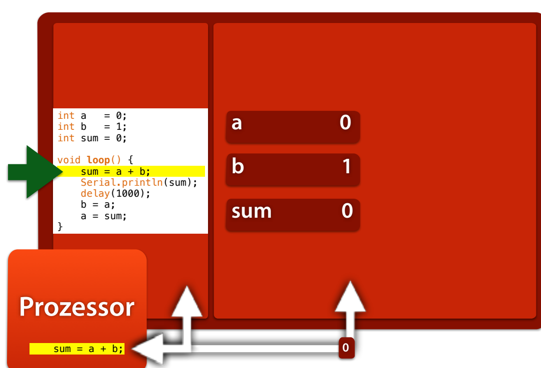


Daten adressieren

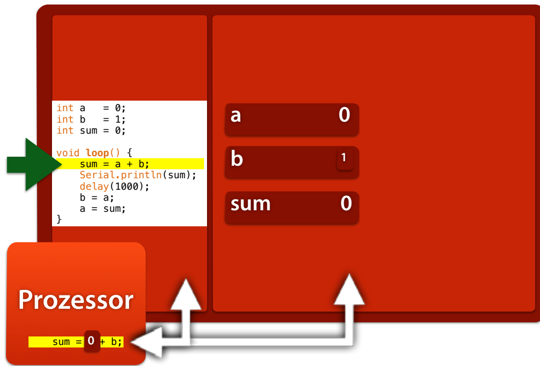


Dazu muss der Bus aber wissen, wo die Daten zu finden sind

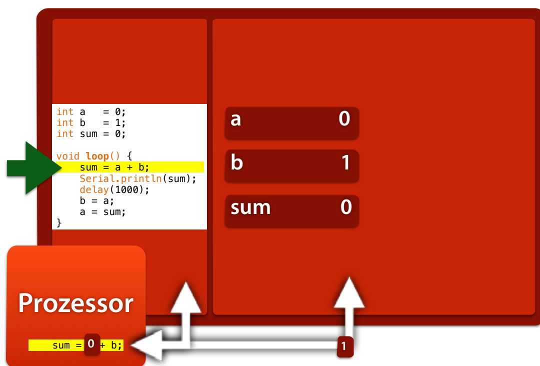
Daten lesen



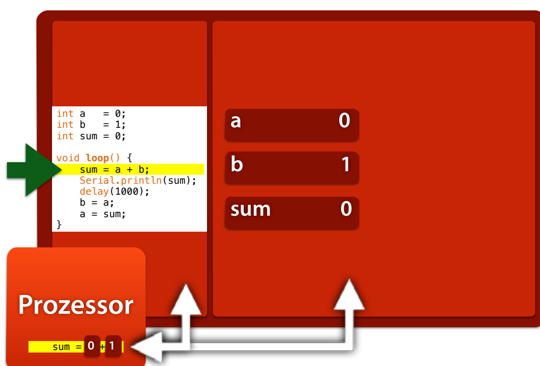
Daten lesen



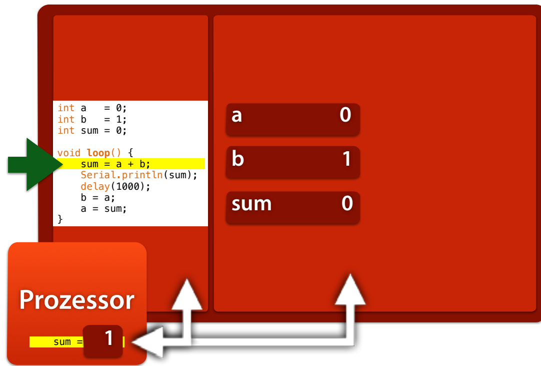
Daten lesen



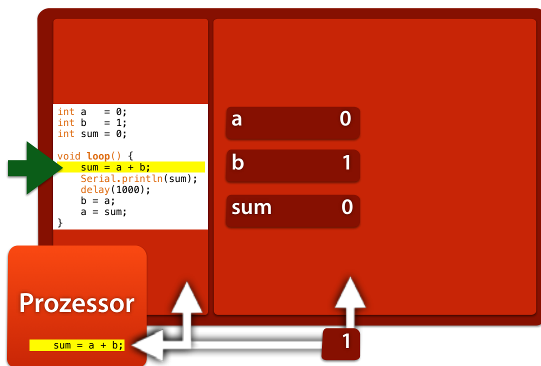
Daten lesen



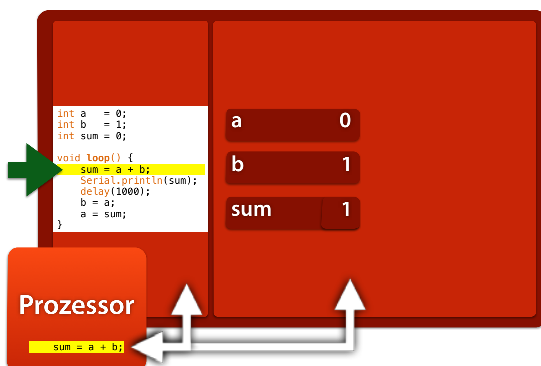
Daten berechnen



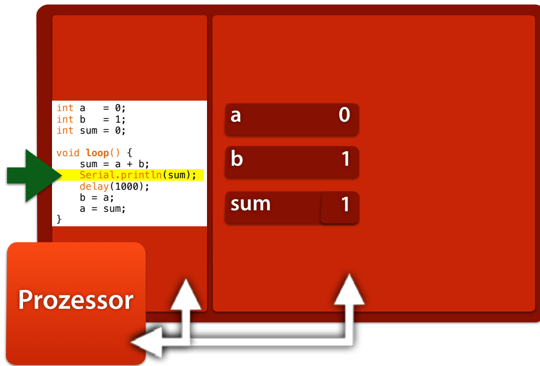
Daten schreiben



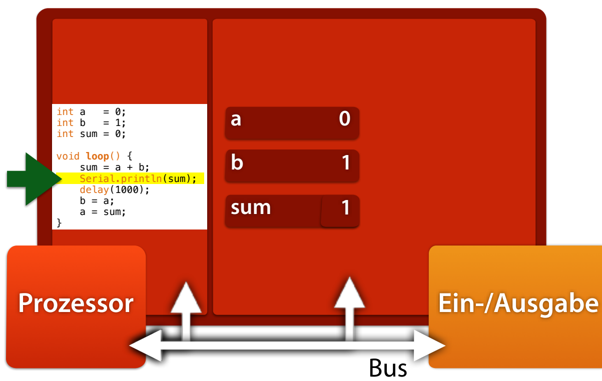
Daten schreiben



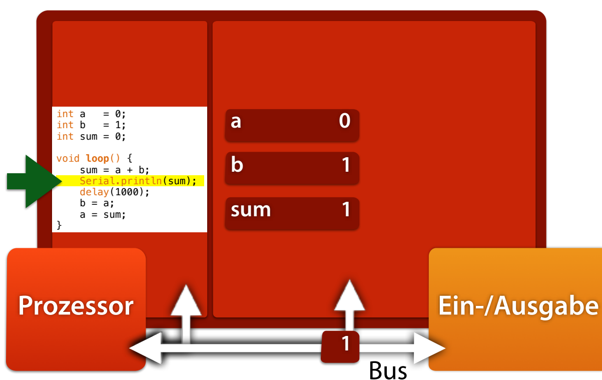
Nächste Anweisung

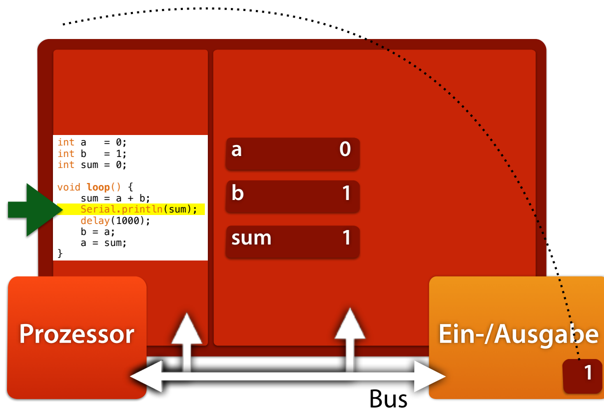


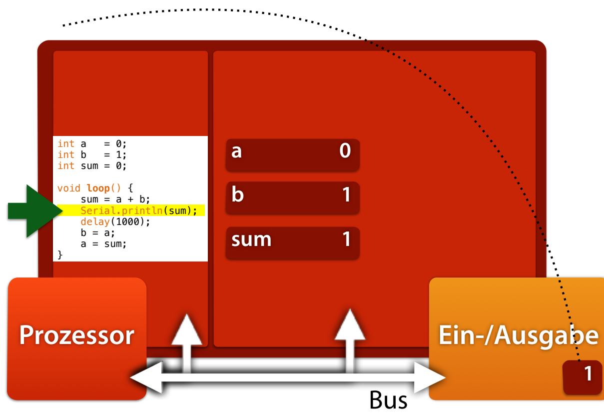
Ein-/Ausgabe

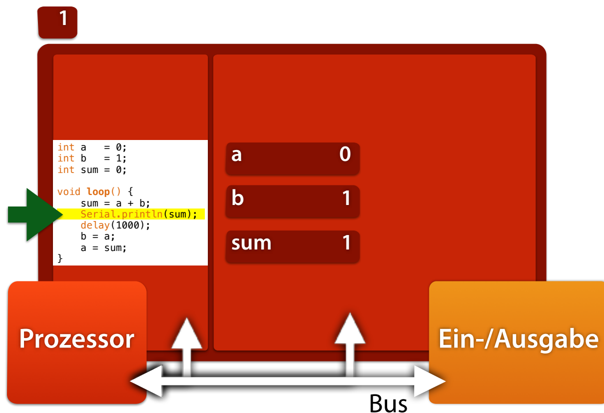


Ein-/Ausgabe

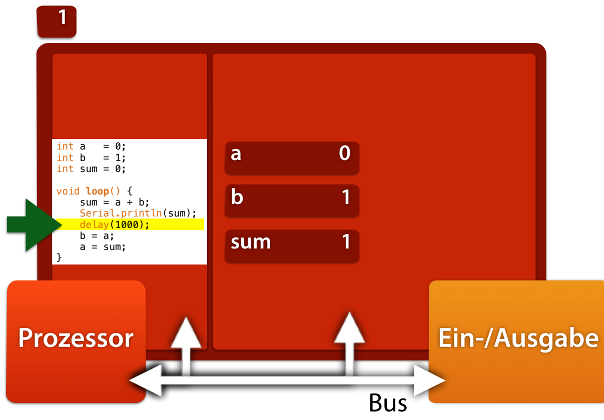



Ein-/Ausgabe
int a = 0;
int b = 1;
int sum = 0;
void loop() {
sum = a + b;
Serial.println(sum);
delay(1000);
b = a;
a = sum;
}
a 0
b 1
sum 1
Prozessor
Ein-/Ausgabe
1
Bus

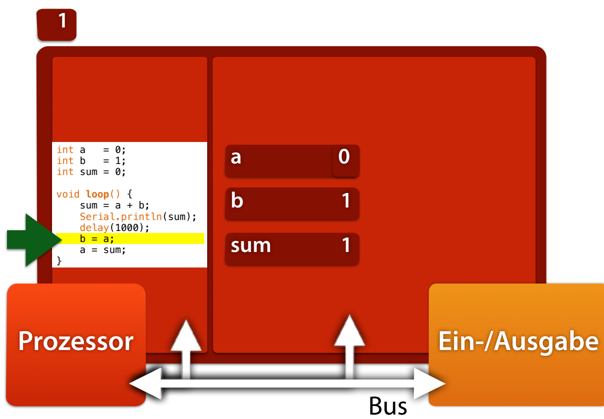

Ein-/Ausgabe
int a = 0;
int b = 1;
int sum = 0;
void loop() {
sum = a + b;
Serial.println(sum);
delay(1000);
b = a;
a = sum;
}
a 0
b 1
sum 1
Prozessor
Ein-/Ausgabe
1
Bus


Ein-/Ausgabe
1
int a = 0;
int b = 1;
int sum = 0;
void loop() {
sum = a + b;
Serial.println(sum);
delay(1000);
b = a;
a = sum;
}
a 0
b 1
sum 1
Prozessor
Ein-/Ausgabe
Bus

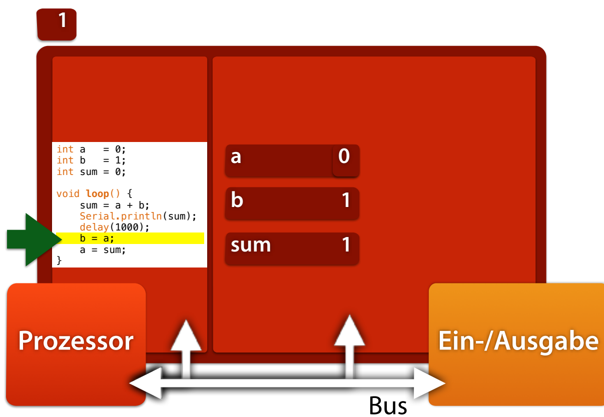
Pause

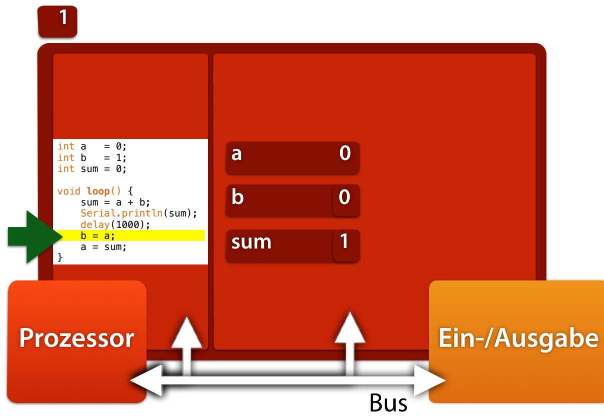


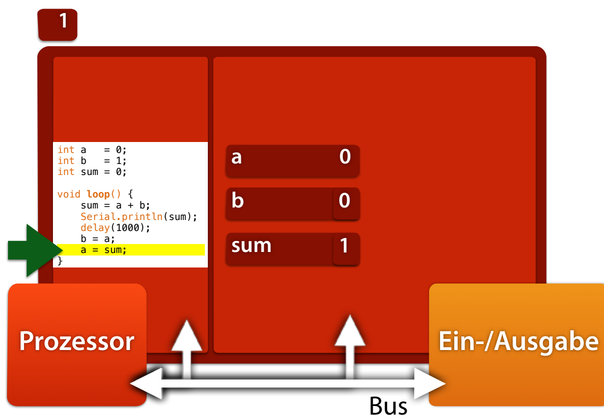
Zuweisung

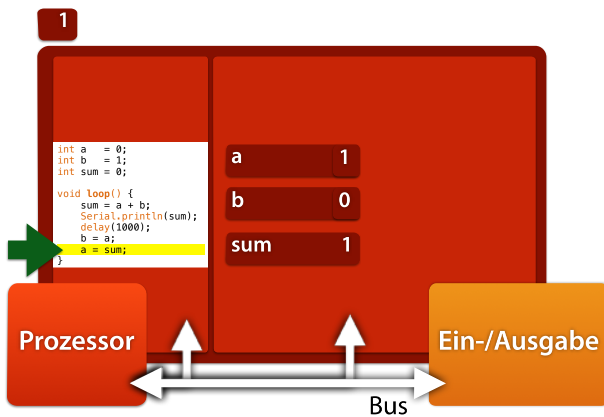


Zuweisung

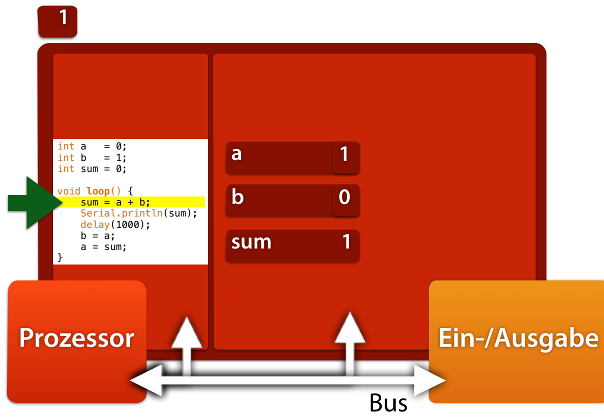



Zuweisung
1
int a   = 0;
int b   = 1;
int sum = 0;
void loop() {
sum = a + b;
Serial.println(sum);
delay(1000);
b = a;
a = sum;
}
a 0
b 0
sum 1
Prozessor
Ein-/Ausgabe
Bus

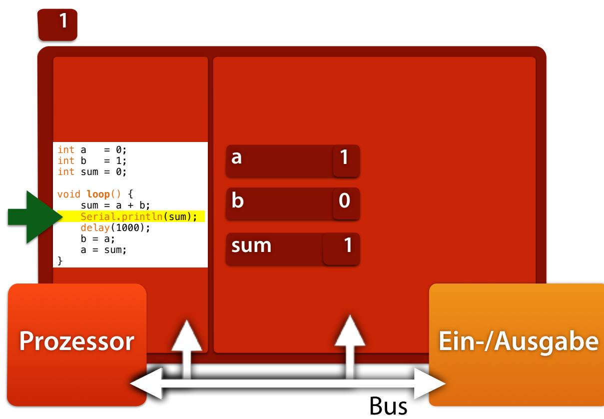

Zuweisung
1
int a   = 0;
int b   = 1;
int sum = 0;
void loop() {
sum = a + b;
Serial.println(sum);
delay(1000);
b = a;
a = sum;
}
a 0
b 0
sum 1
Prozessor
Ein-/Ausgabe
Bus


Zuweisung
1
int a   = 0;
int b   = 1;
int sum = 0;
void loop() {
sum = a + b;
Serial.println(sum);
delay(1000);
b = a;
a = sum;
}
a 1
b 0
sum 1
Prozessor
Ein-/Ausgabe
Bus

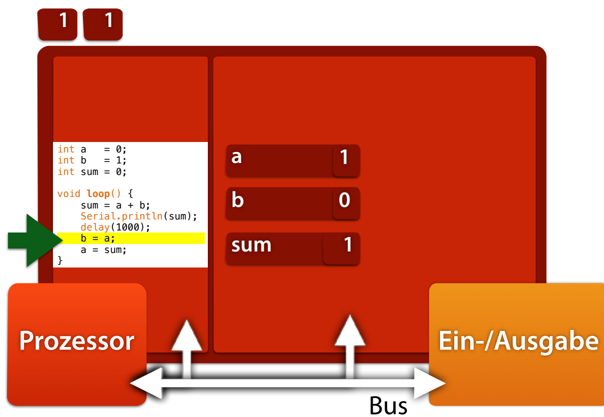
Zuweisung

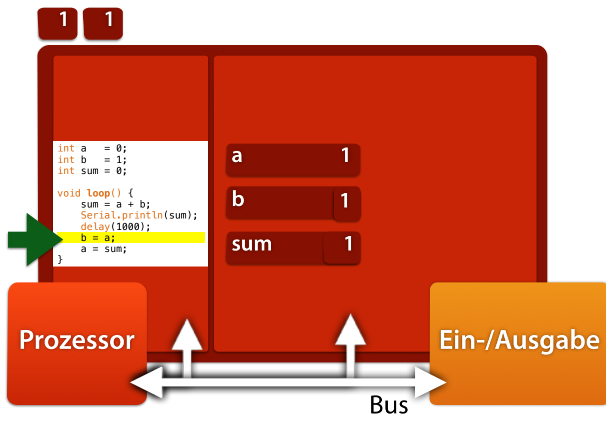


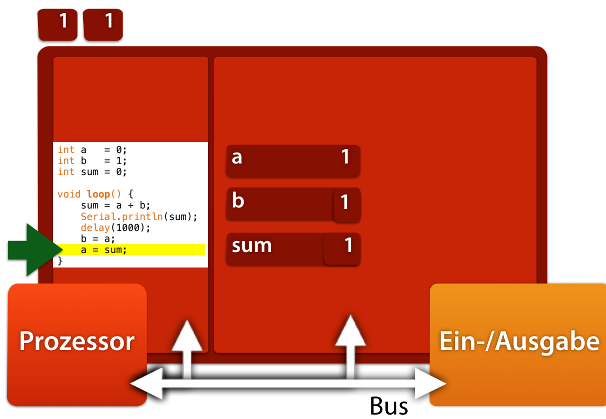
Zuweisung

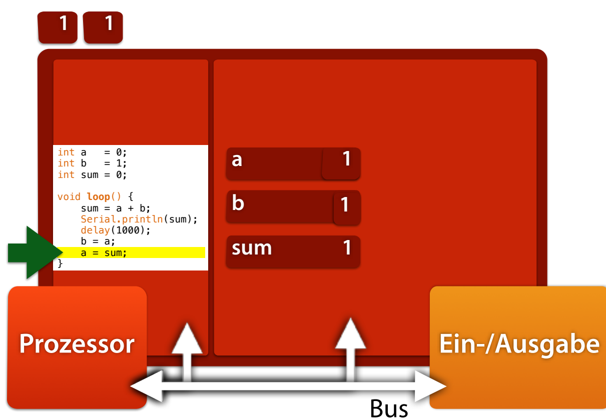


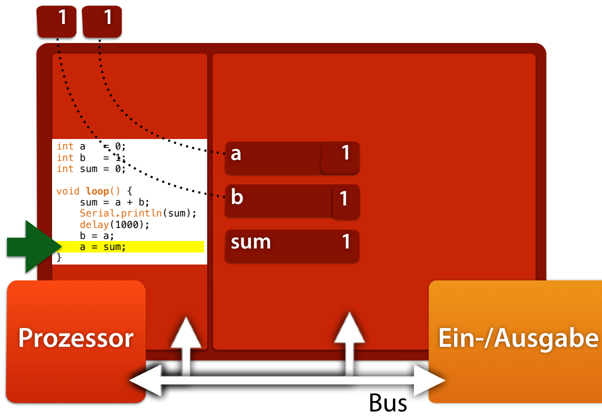
Zuweisung



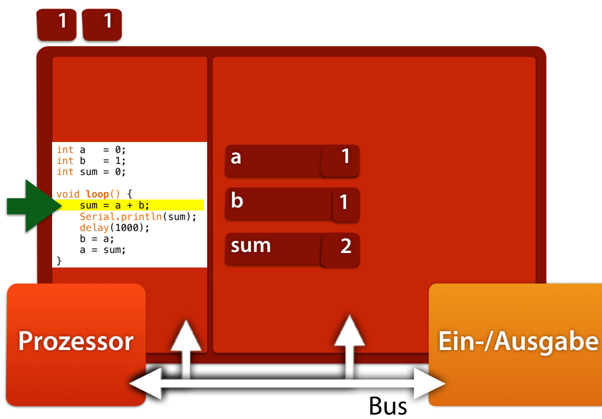

Zuweisung
1
1
int a = 0;
int b = 1;
int sum = 0;
void loop() {
sum = a + b;
Serial.println(sum);
delay(1000);
b = a;
a = sum;
}
a 1
b 1
sum 1
Prozessor
Ein-/Ausgabe
Bus


Zuweisung
1
1
int a = 0;
int b = 1;
int sum = 0;
void loop() {
sum = a + b;
Serial.println(sum);
delay(1000);
b = a;
a = sum;
}
a 1
b 1
sum 1
Prozessor
Ein-/Ausgabe
Bus

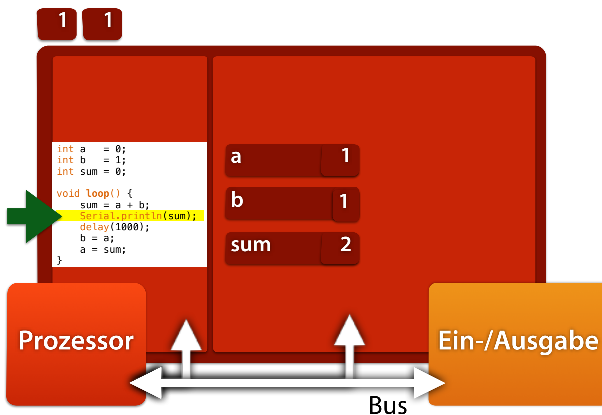

Zuweisung
1
1
int a = 0;
int b = 1;
int sum = 0;
void loop() {
sum = a + b;
Serial.println(sum);
delay(1000);
b = a;
a = sum;
}
a 1
b 1
sum 1
Prozessor
Ein-/Ausgabe
Bus



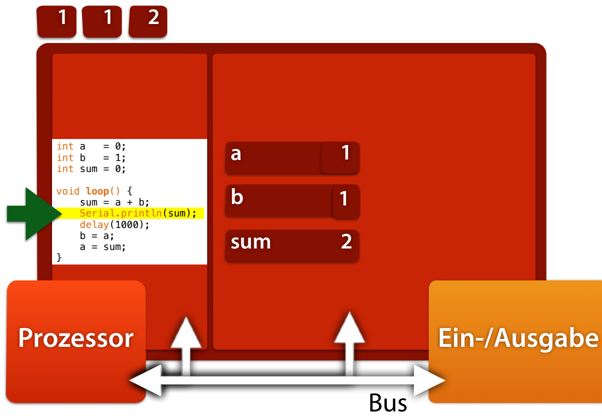
Beobachtung: a ist immer die letzte, b die vorletzte Zahl der Reihe



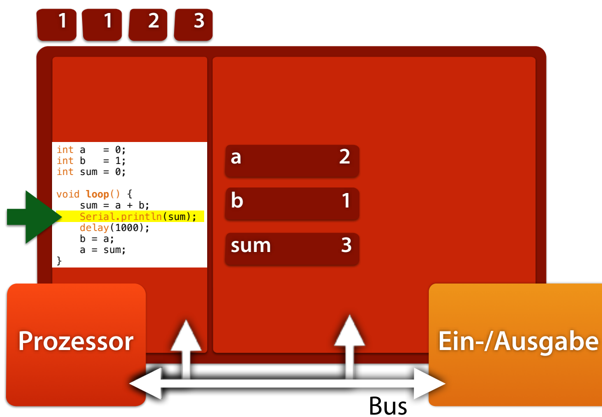
Beobachtung: a ist immer die letzte, b die vorletzte Zahl der Reihe



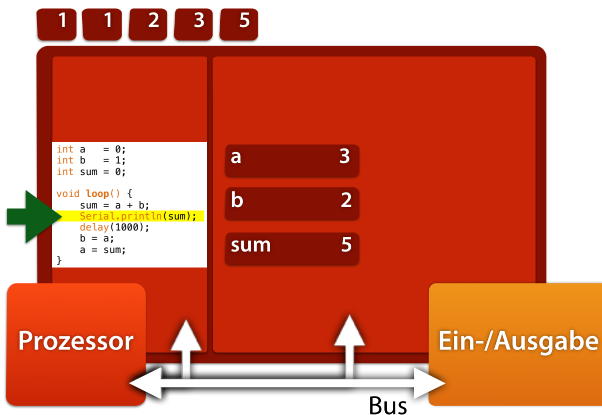
Beobachtung: a ist immer die letzte, b die vorletzte Zahl der Reihe



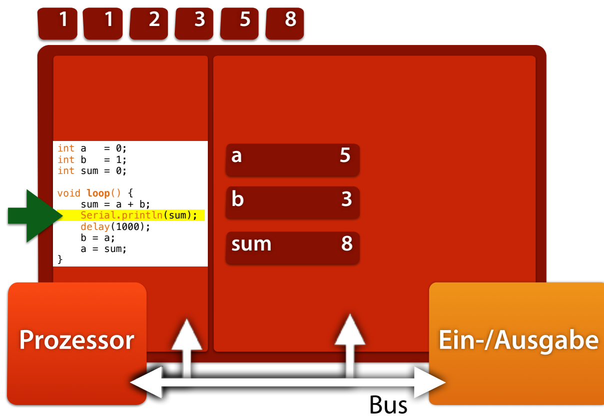
Beobachtung: a ist immer die letzte, b die vorletzte Zahl der Reihe



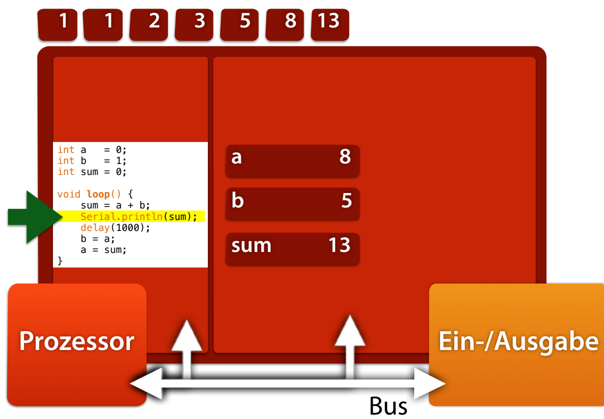
Beobachtung: a ist immer die letzte, b die vorletzte Zahl der Reihe



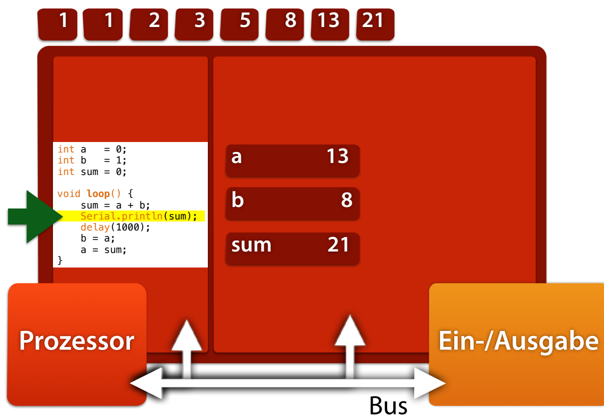
Beobachtung: a ist immer die letzte, b die vorletzte Zahl der Reihe



Beobachtung: a ist immer die letzte, b die vorletzte Zahl der Reihe

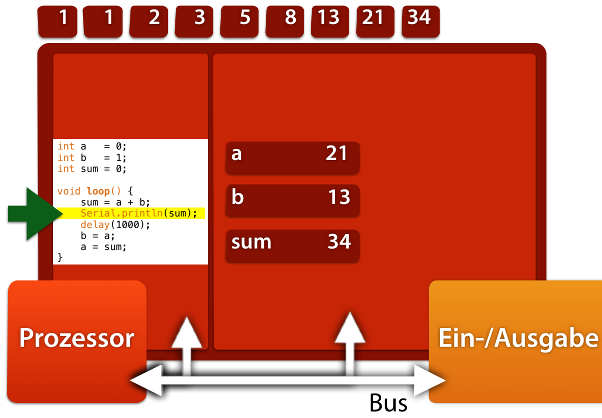


Beobachtung: a ist immer die letzte, b die vorletzte Zahl der Reihe



Beobachtung: a ist immer die letzte, b die vorletzte Zahl der Reihe

Nächste Iteration



Beobachtung: a ist immer die letzte, b die vorletzte Zahl der Reihe

---

Fibonacci-Folge

1 1 2 3 5 8 13 21 34

Beobachtung: a ist immer die letzte, b die vorletzte Zahl der Reihe

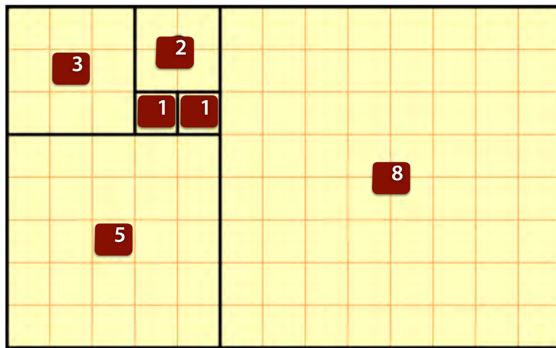
---

Fibonacci-Folge

1 1 2 3 5 8 13 21 34

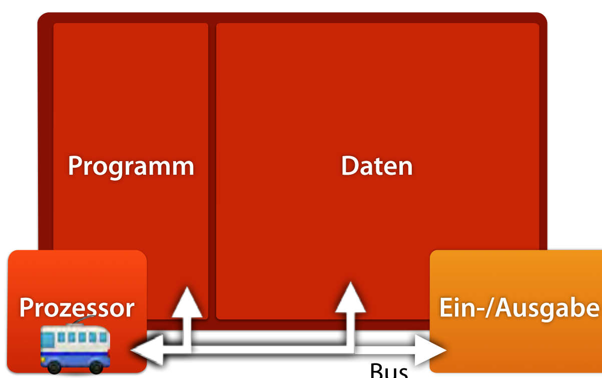
Fibonacci illustrierte diese Folge durch die einfache mathematische Modellierung des Wachstums einer Population von Kaninchen nach folgenden Regeln:

1. Jedes Paar Kaninchen wirft pro Monat ein weiteres Paar Kaninchen.
2. Ein neugeborenes Paar bekommt erst im zweiten Lebensmonat Nachwuchs (die Austragungszeit reicht von einem Monat in den nächsten).
3. Die Tiere befinden sich in einem abgeschlossenen Raum („in quodam loco, qui erat



Kachelmuster aus Quadraten, deren Seitenlänge der Fibonacci-Folge entsprechen
(Quelle: Wikipedia)

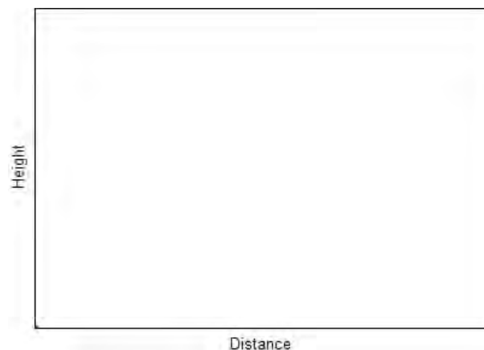
Viele Pflanzen weisen in der Anordnung ihrer Blätter und anderer Teile Spiralen auf, deren Anzahlen durch Fibonacci-Zahlen gegeben sind, wie beispielsweise bei den Früchten in Fruchtständen. Das ist dann der Fall, wenn der Winkel zwischen architektonisch benachbarten Blättern oder Früchten bezüglich der Pflanzenachse der Goldene Winkel ist. Hintergrund ist der Umstand, dass die rationalen Zahlen, die den zugrunde liegenden Goldenen Schnitt am besten approximieren, Brüche von aufeinanderfolgenden



John von Neumann

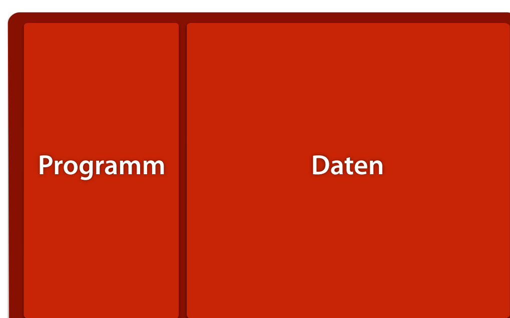
János Neumann Margittai (* 28. Dezember 1903 in Budapest (Österreich-Ungarn) als János Lajos Neumann; † 8. Februar 1957 in Washington, D.C.) war ein Mathematiker österreichisch-ungarischer Herkunft. Er leistete bedeutende Beiträge zur mathematischen Logik, Funktionalanalysis, Quantenmechanik und Spieltheorie und gilt als einer der Väter der Informatik. Auch an der Weiterentwicklung des amerikanischen Nuklearbomben-Programms bis hin zur

Ballistische Tabellen

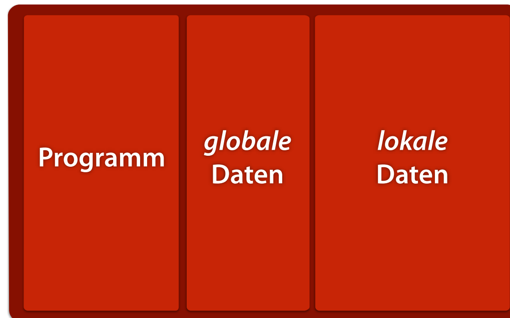


Und was wurde mit diesen ersten Rechnern gemacht? Sie wurden für ballistische Berechnungen genutzt.

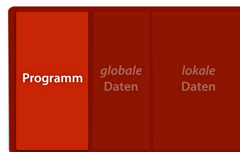
Der Speicher



Der Speicher

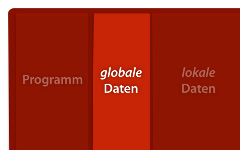


Das Programm



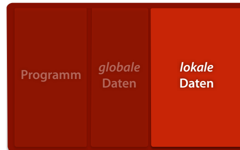
- Liegt wie Daten im Speicher
- Kann nicht auf sich selbst zugreifen oder verändern
- Das *Betriebssystem* lädt und verwaltet Programme im Speicher

Globale Daten



- Enthält Variablen mit *globaler* Sichtbarkeit (= *außerhalb* von Funktionen definiert)
- Von *allen* Funktionen zugänglich

Lokale Daten



- Enthält Variablen mit *lokaler* Sichtbarkeit (= *innerhalb* von Funktionen definiert)
- Lokale Variablen und Parameter existieren nur *während der Ausführung* der jeweiligen Funktion
- Ein *Funktionsrahmen* speichert diese

Globale Variablen

```
int a   = 0;
int b   = 1;      ——— globale Variablen
int sum = 0;

void loop() {
    sum = a + b;
    Serial.println(sum);
    delay(1000);
    b = a;
    a = sum;
}
```

– gibt 1 2 3 5 8 13 … aus

Lokale Variablen

```
int a   = 0;      ——— globale Variablen
int b   = 1;

void loop() {
    int sum = a + b;  ——— lokale Variable
    Serial.println(sum);
    delay(1000);
    b = a;
    a = sum;
}
```

– gibt 1 2 3 5 8 13 … aus

Lokale Variablen

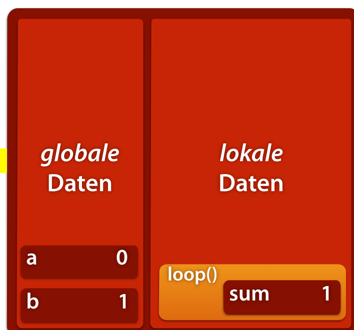
```
int a   = 0;
int b   = 1;

void loop() {
    int sum = a + b;
    Serial.println(sum);
    delay(1000);
    b = a;
    a = sum;
}
```

Zu Beginn haben wir keine lokalen Daten – die werden erst während der Ausführung erzeugt

---

Lokale Variablen

```
int a   = 0;
int b   = 1;

void loop() {
    int sum = a + b;
    Serial.println(sum);
    delay(1000);
    b = a;
    a = sum;
}
```

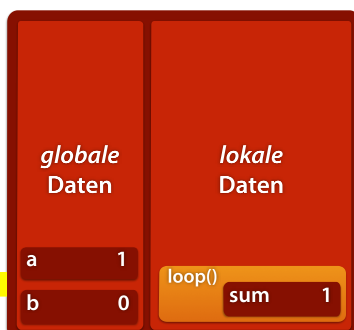


Zu Beginn haben wir keine lokalen Daten – die werden erst während der Ausführung erzeugt

---

Lokale Variablen

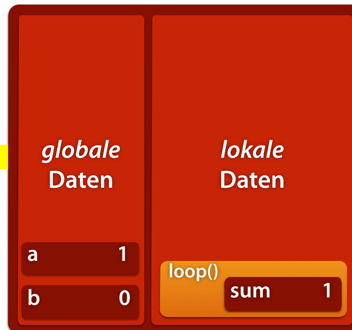
```
int a   = 0;
int b   = 1;

void loop() {
    int sum = a + b;
    Serial.println(sum);
    delay(1000);
    b = a;
    a = sum;
}
```



Zu Beginn haben wir keine lokalen Daten – die werden erst während der Ausführung erzeugt

Lokale Variablen

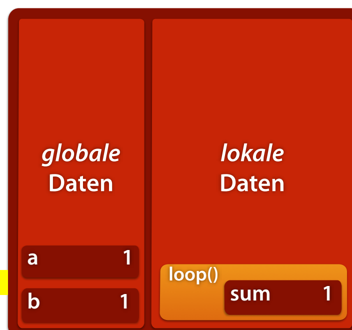
```
int a   = 0;
int b   = 1;

void loop() {
    int sum = a + b;
    Serial.println(sum);
    delay(1000);
    b = a;
    a = sum;
}
```



Zu Beginn haben wir keine lokalen Daten – die werden erst während der Ausführung erzeugt

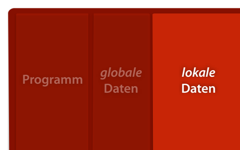
Lokale Variablen

```
int a   = 0;
int b   = 1;

void loop() {
    int sum = a + b;
    Serial.println(sum);
    delay(1000);
    b = a;
    a = sum;
```



Zu Beginn haben wir keine lokalen Daten – die werden erst während der Ausführung erzeugt

Funktionsstapel



- Die Funktionsrahmen sind als *Stapel* organisiert
- Der oberste Rahmen ist jeweils *aktiv*
- Bei Rückkehr aus der obersten Funktion wird die darunterlegende (aufrufende) Funktion hinter der Aufrufstelle fortgesetzt

```
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();
}

void loop() {
    morse_SINK();
}
```

Morsecode

Funktionsstapel

```
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();
}

void loop() {
    morse_SINK();
}
```

Funktionsstapel

```
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();
}

void loop() {
    morse_SINK();
}
```

Funktionsstapel

```
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();
}

void loop() {
    morse_SINK();
}
```

Funktionsstapel

```
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();
}

void loop() {
    morse_SINK();
}
```

Funktionsstapel

```
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();
}

void loop() {
    morse_SINK();
}
```

Funktionsstapel

```
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();
}

void loop() {
    morse_SINK();
}
```

Funktionsstapel

```
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();
}

void loop() {
    morse_SINK();
}
```

Funktionsstapel

```
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();
}

void loop() {
    morse_SINK();
}
```

Funktionsstapel

```
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();
}

void loop() {
    morse_SINK();
}
```

Funktionsstapel

```
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();
}

void loop() {
    morse_SINK();
}
```

Funktionsstapel

```
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();
}

void loop() {
    morse_SINK();
}
```

Funktionsstapel

```
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();
}

void loop() {
    morse_SINK();
}
```

Funktionsstapel

```
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();
}

void loop() {
    morse_SINK();
}
```

Funktionsstapel

```
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();
}

void loop() {
    morse_SINK();
}
```

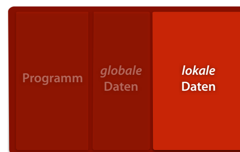
Funktionsstapel

- Aufruf: ablegen *(push)*
- Rückkehr: wegnehmen *(pop)*

Funktionsstapel – analog zu Tablettstapel Mensa

Quelle: http://www.blanco-professional.com/de/catering/produkte/blanco_spender/tablettspender.cfm

Argumente

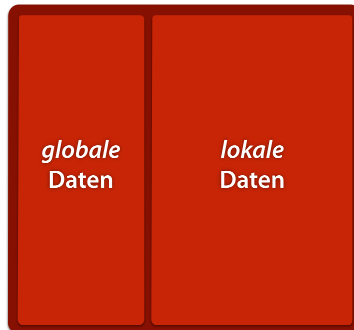


- Beim Aufruf einer Funktion *f* werden *Argumente wie lokale* Variablen abgelegt – also im *Funktionsrahmen* von *f*

```
// 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();
  }
  // usw. für 2–8
  if (n == 9) {
    dah(); dah(); dah(); dah(); dit();
  }
  pause_letter();
}
```

Argumente

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

void loop() {
   morse_digit(1);
   morse_digit(2);
   morse_digit(3);
}
```



Zu Beginn haben wir keine lokalen Daten – die werden erst während der Ausführung erzeugt

---

Argumente

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

void loop() {
   morse_digit(1);
   morse_digit(2);
   morse_digit(3);
}
```

Zu Beginn haben wir keine lokalen Daten – die werden erst während der Ausführung erzeugt

---

Argumente

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

void loop() {
   morse_digit(1);
   morse_digit(2);
   morse_digit(3);
}
```

Zu Beginn haben wir keine lokalen Daten – die werden erst während der Ausführung erzeugt

Argumente

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

void loop() {
   morse_digit(1);
   morse_digit(2);
   morse_digit(3);
}
```

Zu Beginn haben wir keine lokalen Daten – die werden erst während der Ausführung erzeugt

---

Argumente

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

void loop() {
   morse_digit(1);
   morse_digit(2);
   morse_digit(3);
}
```

• — — — —

Zu Beginn haben wir keine lokalen Daten – die werden erst während der Ausführung erzeugt

---

Argumente

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

void loop() {
   morse_digit(1);
   morse_digit(2);
   morse_digit(3);
}
```

Zu Beginn haben wir keine lokalen Daten – die werden erst während der Ausführung erzeugt

Argumente

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

void loop() {
   morse_digit(1);
   morse_digit(2);
   morse_digit(3);
}
```

Zu Beginn haben wir keine lokalen Daten – die werden erst während der Ausführung erzeugt

---

Argumente

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

void loop() {
   morse_digit(1);
   morse_digit(2);
   morse_digit(3);
}
```

• • – – –

Zu Beginn haben wir keine lokalen Daten – die werden erst während der Ausführung erzeugt

---

Argumente

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

void loop() {
   morse_digit(1);
   morse_digit(2);
   morse_digit(3);
}
```

Zu Beginn haben wir keine lokalen Daten – die werden erst während der Ausführung erzeugt

Argumente

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

void loop() {
   morse_digit(1);
   morse_digit(2);
   morse_digit(3);
}
```

Zu Beginn haben wir keine lokalen Daten – die werden erst während der Ausführung erzeugt

Argumente

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

void loop() {
   morse_digit(1);
   morse_digit(2);
   morse_digit(3);
}
```

• • • – –

Zu Beginn haben wir keine lokalen Daten – die werden erst während der Ausführung erzeugt

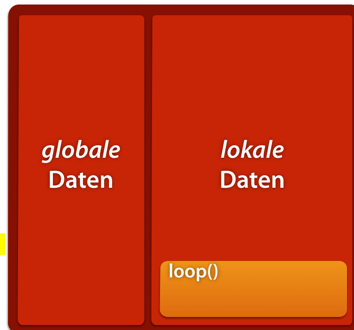
Argumente

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

void loop() {
   morse_digit(1);
   morse_digit(2);
   morse_digit(3);
}
```

Zu Beginn haben wir keine lokalen Daten – die werden erst während der Ausführung erzeugt

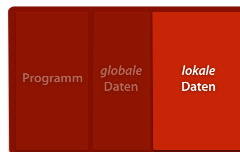
Argumente

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

void loop() {
   morse_digit(1);
   morse_digit(2);
   morse_digit(3);
}
```



Zu Beginn haben wir keine lokalen Daten – die werden erst während der Ausführung erzeugt

Aktive Funktion



- Das Programm kann nur auf den *jeweils aktiven* (= den obersten) Funktionsrahmen zugreifen
- Ein Aufrufer kann sich darauf verlassen, dass seine lokalen Variablen *unverändert bleiben*

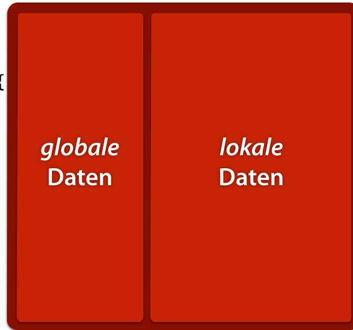
Von Ziffern zu Zahlen

morse_number() gibt eine Zahl *rekursiv* aus:

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

void loop() {
  morse_number(123);
}
```

Rekursion

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

void loop() {
  morse_number(123);
}
```



Zu Beginn haben wir keine lokalen Daten – die werden erst während der Ausführung erzeugt

Rekursion

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

void loop() {
  morse_number(123);
}
```

Zu Beginn haben wir keine lokalen Daten – die werden erst während der Ausführung erzeugt

Rekursion

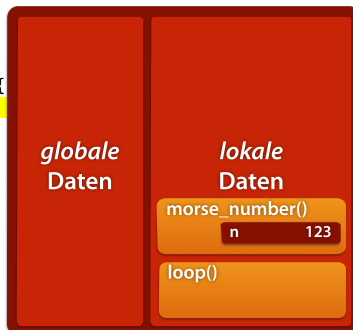
```
void morse_number(int n) {
  if (n >= 10) {
    morse_number(n / 10);
  }
  morse_digit(n % 10);
}

void loop() {
  morse_number(123);
}
```

Rekursion

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

void loop() {
  morse_number(123);
}
```

Rekursion

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

void loop() {
  morse_number(123);
}
```

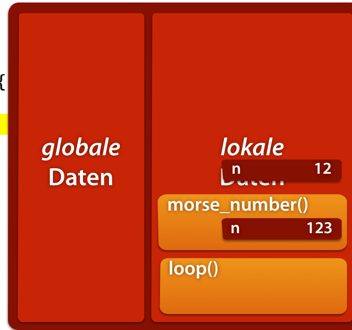


Rekursion

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

void loop() {
  morse_number(123);
}
```

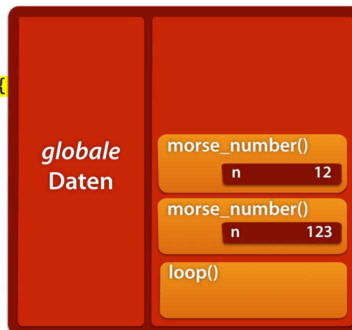
Rekursion

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

void loop() {
  morse_number(123);
}
```



Rekursion

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

void loop() {
  morse_number(123);
}
```



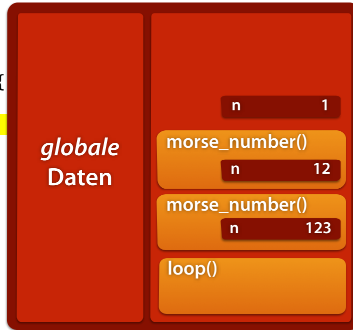
Rekursion

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

void loop() {
  morse_number(123);
}
```

Hier wird auf das **oberste** n der aktiven Funktion zugegriffen – die “unteren”, inaktiven sind nicht zugänglich

Rekursion

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

void loop() {
  morse_number(123);
}
```



Rekursion

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

void loop() {
  morse_number(123);
}
```

Rekursion

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

void loop() {
  morse_number(123);
}
```

Rekursion

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

void loop() {
  morse_number(123);
}
```

Rekursion

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

void loop() {
  morse_number(123);
}
```

• – – – –

Rekursion

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

void loop() {
  morse_number(123);
}
```

• – – – –

Rekursion

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

void loop() {
  morse_number(123);
}
```

• — — — — • • — — —

Rekursion

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

void loop() {
  morse_number(123);
}
```

• — — — — • • — — —

Rekursion

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

void loop() {
  morse_number(123);
}
```

• — — — — • • — — —

Rekursion

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

void loop() {
  morse_number(123);
}
```

• — — — —   • • — — —

Rekursion

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

void loop() {
  morse_number(123);
}
```

• — — — —   • • — — —   • • • — —

Rekursion

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

void loop() {
  morse_number(123);
}
```

• — — — —   • • — — —   • • • — —

Rekursion

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

void loop() {
  morse_number(123);
}
```

• – – – –   • • – – –   • • • – –

Stapelspeicher

Stapelspeicher

Friedrich L. Bauer (1924–2015)

Friedrich Ludwig Bauer (* 10. Juni 1924 in Regensburg; † 26. März 2015) war ein deutscher Pionier der Informatik. Er konstruierte in den 1950er Jahren mehrere Verschlüsselungsmaschinen, erfand 1957 das Prinzip des Kellerspeichers,[1] hielt 1967 an der Technischen Universität München die erste offizielle Informatikvorlesung[2] in Deutschland und richtete 1988 die erste Computerausstellung im Deutschen Museum aus. Seine Publikationen zur Kryptologie sind Standardwerke der Informatik.

Zuweisung

- Die Anweisung

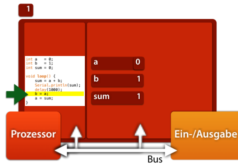
  name = wert

  bewirkt, dass die Variable *name* den neuen Wert *wert* hat.
- Im weiteren Programmablauf liefert jeder spätere Zugriff auf die Variable den Wert *wert* (bis zur nächsten Zuweisung)

Fibonacci

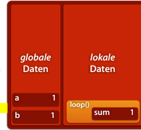
```
int a   = 0;
int b   = 1;
int sum = 0;

void loop() {
    sum = a + b;
    Serial.println(sum);
    delay(1000);
    b = a;
    a = sum;
}
```

Von Neumann-Architektur



Lokale Variablen

```
int a   = 0;
int b   = 1;

void loop() {
    int sum = a + b;
    Serial.println(sum);
    delay(1000);
    b = a;
    a = sum;
```



Handouts

Zuweisung

- Die Anweisung

  name = wert

  bewirkt, dass die Variable *name* den neuen Wert *wert* hat.
- Im weiteren Programmablauf liefert jeder spätere Zugriff auf die Variable den Wert *wert* (bis zur nächsten Zuweisung)

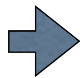
Fibonacci

```
int a   = 0;
int b   = 1;
int sum = 0;

void loop() {
    sum = a + b;
    Serial.println(sum);
    delay(1000);
    b = a;
    a = sum;
}
```

– gibt 1 2 3 5 8 13 … aus

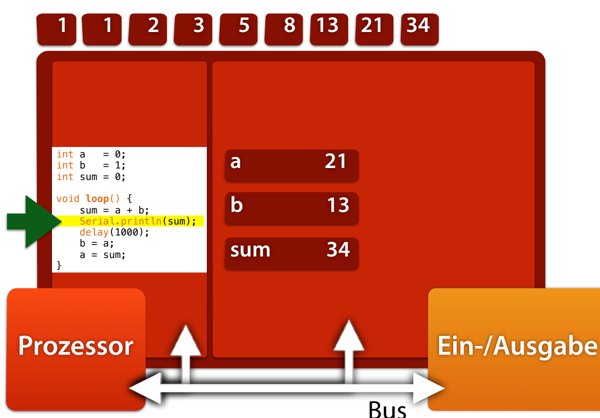
Was passiert hier?

```
int a   = 0;
int b   = 1;
int sum = 0;

void loop() {
    sum = a + b;
    Serial.println(sum);
    delay(1000);
    b = a;
    a = sum;
}
```



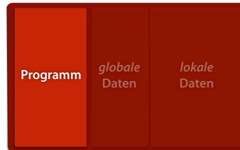
Wir haben das Programm, und wir haben den Rechner. Wie führt der Rechner das Programm aus?

Fibonacci-Folge



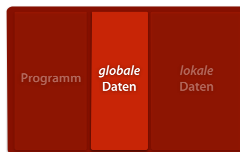
Beobachtung: a ist immer die letzte, b die vorletzte Zahl der Reihe

Das Programm



- Liegt wie Daten im Speicher
- Kann nicht auf sich selbst zugreifen oder verändern
- Das *Betriebssystem* lädt und verwaltet Programme im Speicher

Globale Daten

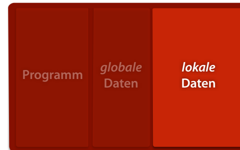


- Enthält Variablen mit *globaler* Sichtbarkeit (= *außerhalb* von Funktionen definiert)
- Von *allen* Funktionen zugänglich

Lokale Daten

- Enthält Variablen mit *lokaler* Sichtbarkeit (= *innerhalb* von Funktionen definiert)
- Lokale Variablen und Parameter existieren nur *während der Ausführung* der jeweiligen Funktion
- Ein *Funktionsrahmen* speichert diese

Funktionsstapel



- Die Funktionsrahmen sind als *Stapel* organisiert
- Der oberste Rahmen ist jeweils *aktiv*
- Bei Rückkehr aus der obersten Funktion wird die darunterlegende (aufrufende) Funktion hinter der Aufrufstelle fortgesetzt