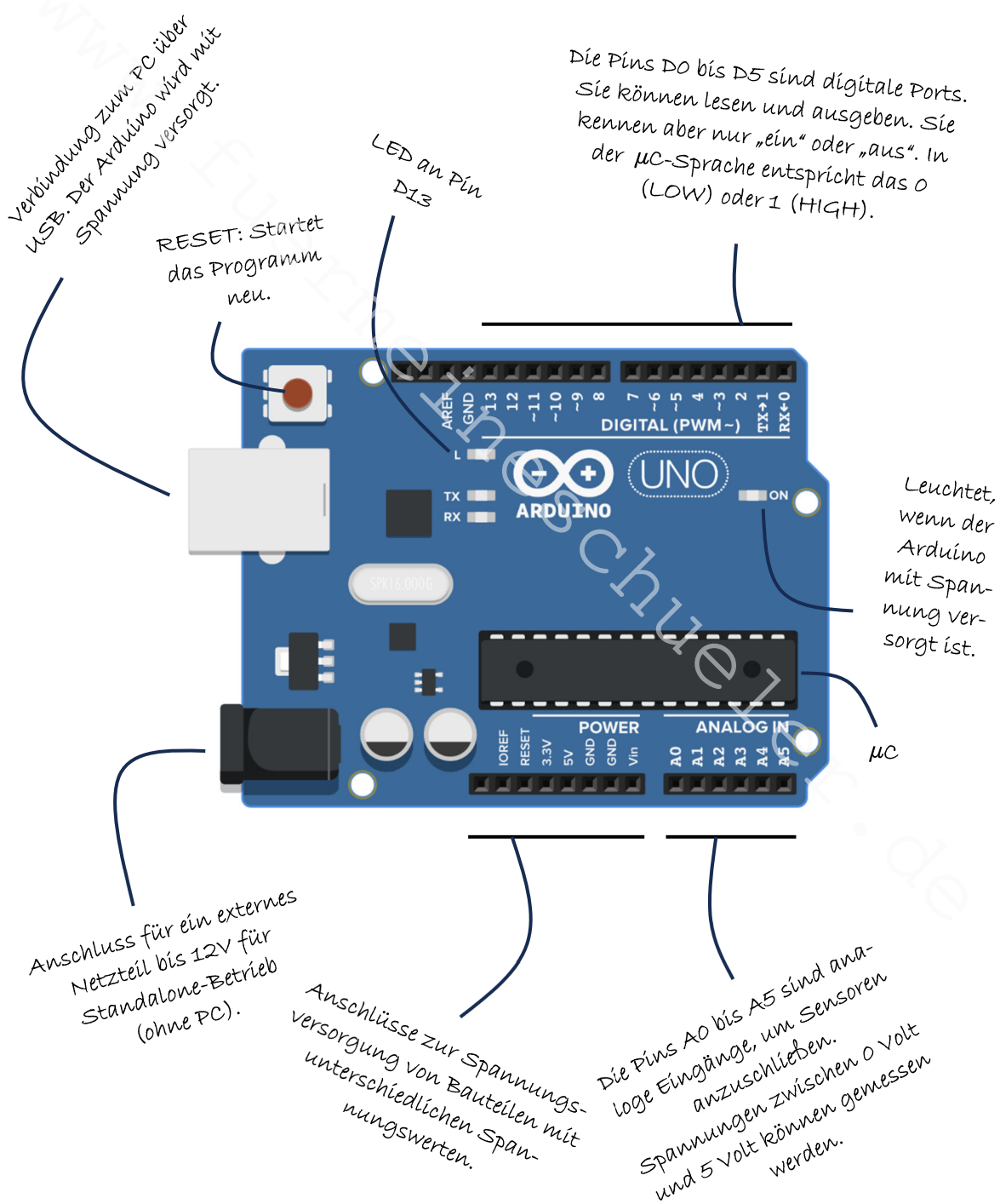
Arduino fundamentals

Fundamentale Grundlagen für die ersten Stunden mit dem Arduino



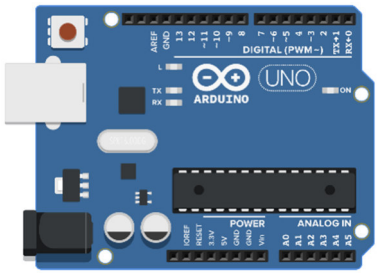
Inhalt

Einleitung

Arduino fundamentals entstand mit Unterstützung von ChatGPT in der Version 3.5. Ziel ist die Darstellung grundsätzlicher Prinzipien. Tiefgreifende und anwendungsbezogene Programme sind hier nicht zu erwarten. Das gezielte Nachschlagen wird durch das Stichwortverzeichnis am Ende des Dokumentes erleichtert. Die Abbildungen sind Screenshots und Ausschnitte der Tinkercad-Oberfläche und der Arduino IDE. Bei Versionspflege fließen typische Fehler aus Unterricht und Klassenarbeiten ein, um eine praxisnahe Hilfestellung zu bieten.

Arduino Uno R3

Arduino Uno R3, Arduino



Der Arduino Uno R3, kurz Arduino, ist eine Open-Source-Plattform für die Entwicklung von elektronischen Projekten. Sie umfasst eine Hardwareplattform mit einem Mikrocontroller und eine leicht zu bedienende Programmierumgebung. Der Arduino ermöglicht es Menschen, auch ohne umfassende Elektronikkenntnisse, eigene interaktive Geräte und Anwendungen zu erstellen.

Arduino Programmierung in der IDE

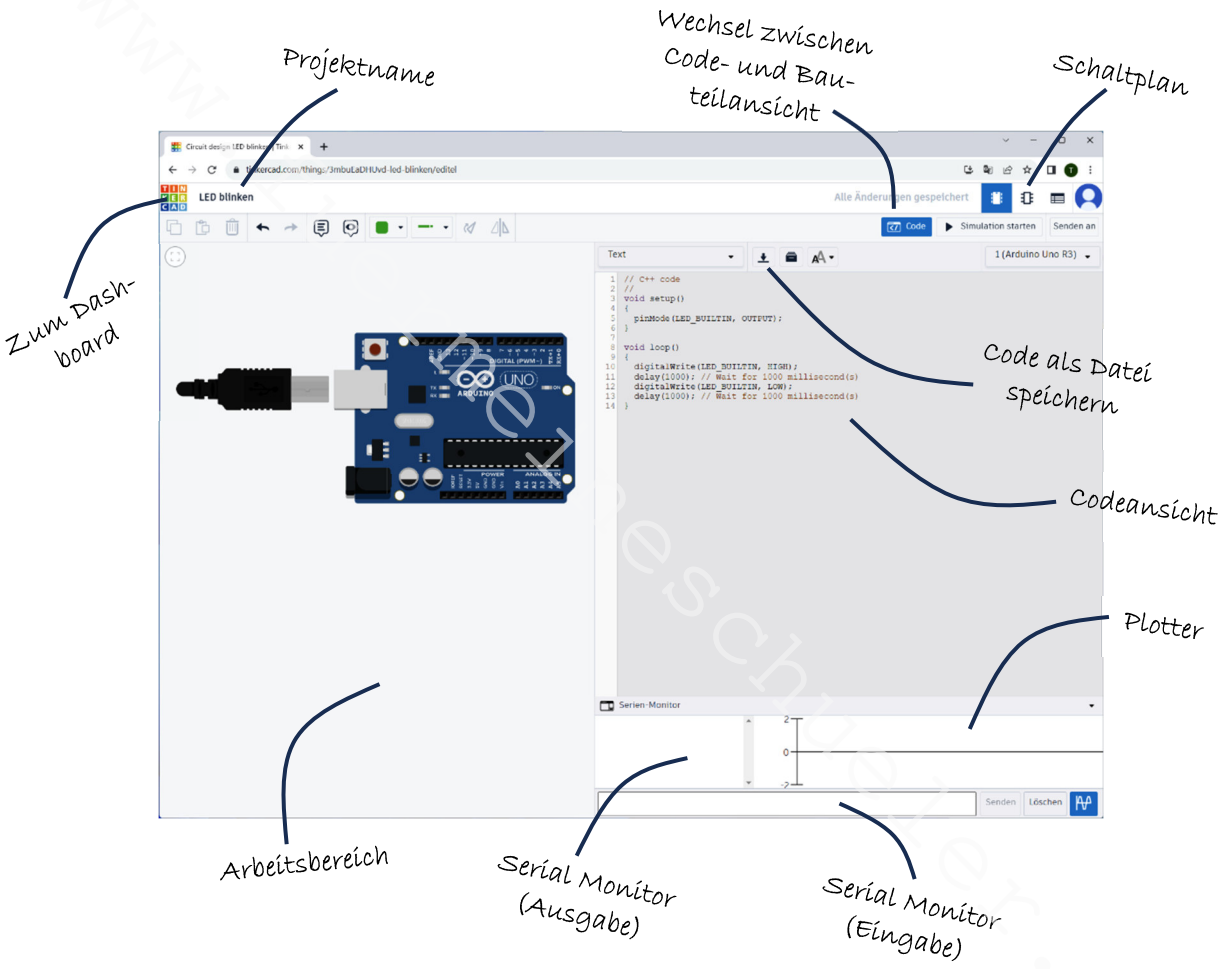
Entwicklungsumgebung, IDE, Arduino DIE, Arduino Weblink

Der Arduino wird mit der Integrated Development Environment (IDE) programmiert. Das Anlegen eines Kontos ermöglicht die Nutzung einer Online-IDE. Die kostenfreie IDE als Download, Dokumentationen und vieles mehr gibt es hier:
https://www.arduino.cc

Tinkercad: Arduino online simulieren

Tinkercad, Simulation, virtuelle Schaltungen

Tinkercad ist eine webbasierte Plattform für Elektronik-Simulation, die es Benutzern ermöglicht, Programme zu erstellen, deren Funktion zu simulieren und virtuelle Schaltungen zu testen. Tinkercad wird für das Erstellen von Schaltkreisen verwendet, insbesondere im Zusammenhang mit Arduino-Projekten. Das Erstellen eines Kontos ist erforderlich. Ein anonymer Zugang wird durch eine Gruppe (Klasse) möglich. Dafür muss „nur“ die Lehrkraft die Hose (E-Mail Adresse) runterlassen. Die Tinkercad-Webseite lautet: https://www.tinkercad.com



Sketch oder Programm?

Sketch, Programm

Ein Arduino Sketch ist das Programm, das auf einem Arduino-Mikrocontroller ausgeführt wird. Es ist im Grunde genommen der Code, den ein Entwickler schreibt, um die gewünschte Funktionalität auf dem Arduino zu implementieren. Der Sketch wird in der Arduino-Entwicklungsumgebung (IDE) geschrieben und kann dann auf den Mikrocontroller hochgeladen werden, um das gewünschte Verhalten auszuführen. Beide Begriffe können gleichberechtigt genutzt werden.

Programmiersprache

Programmiersprache, C++

Die Programmiersprache für Arduino ist eine angepasste Version von C/C++.

Grundsätzliche Struktur eines Sketches

void setup(), void loop(), Struktur

Die grundsätzliche Struktur eines Arduino-Sketches ist:

cpp Copy code

```cpp
void setup() {
  // Initialisierungen
}

void loop() {
  // Wiederholender Code
}
```

Die setup()-Funktion beinhaltet Initialisierungen und wird einmalig ausgeführt.
Die loop()-Funktion enthält das Hauptprogramm, den sich wiederholenden Teil des Programms (Dauerschleife bzw. Endlosschleife).

Für Testläufe kann es wünschenswert sein, das Programm nur einmalig auszuführen. Es bieten sich zwei grundsätzliche Optionen an:

- Das Hauptprogramm in die setup()-Funktion setzen.
- Gezielter Abbruch des Hauptprogramms (siehe Stichwortverzeichnis).

Syntax-Basics

Geschweifte Klammern { und }

geschweifte Klammern, {}

Die geschweiften Klammern werden in einem Arduino-Sketch verwendet, um zusammenhängende Anweisungen als Blöcke von Code zu markieren. Sie zeigen den Anfang und das Ende eines Codeabschnitts an, beispielsweise in setup() und loop().

Semikolon ;

Semikolon, ;

Das Semikolon dient in der Programmierung dazu, das Ende einer Anweisung zu markieren. Es zeigt dem Compiler an, dass die aktuelle Anweisung abgeschlossen ist und die nächste beginnen kann.

Guter Programmierstil

Einzug

Einzug, Programmierstil

Einzüge von Codezeilen verbessern die Übersichtlichkeit von Programmen. Die Lesbarkeit wird erleichtert und die Struktur sichtbarer. Dadurch können Fehler vermieden werden und das Programm ist einfacher zu warten. Sichtbare Programmstruktur erleichtert die Erweiterbarkeit und Anpassung von Code.

Kommentare

Kommentare, // , /* */

In einem Arduino-Sketch setzt man Kommentare mit zwei Schrägstrichen für einzeilige Kommentare oder mit /* und */ für mehrzeilige Kommentare.

cpp Copy code

```cpp
// Dies ist ein einzeiliger Kommentar
```

cpp Copy code

```cpp
/*
Das ist ein mehrzeiliger Kommentar.
Er erstreckt sich über mehrere Zeilen.
*/
```

Kommentare dienen dazu, den Code zu erklären und für andere (und auch für den Entwickler selbst) leichter verständlich zu machen. Sie sind wichtig für die Dokumentation, Fehlerbehebung und Teamarbeit. Kommentare werden vom Compiler ignoriert.

Serial Monitor: Serielle Kommunikation

Serial Monitor, Serial.begin(), Serial.println(), Serial.print(), delay

Der Serial Monitor ist ein Kommunikationstool in der Arduino-IDE, der es ermöglicht, Daten zwischen dem Arduino-Board und einem Computer zu senden und zu empfangen.

cpp Copy code

```cpp
void setup() {
  // Initialisiere die serielle Kommunikation mit 9600 Baud
  Serial.begin(9600);
}

void loop() {
  // Sende den Text über die serielle Schnittstelle
  Serial.println("I am Arduino");
  // Warte für eine Sekunde
  delay(1000);
}
```

Das Programm sendet den Text "I am Arduino" über die serielle Schnittstelle.
Es initialisiert die serielle Kommunikation mit einer Baudrate von 9600. In der Hauptschleife wird der Text gesendet und es wird eine Sekunde gewartet, bevor der Vorgang wiederholt wird.

Die Anweisung Serial.println() erzeugt nach Ausgabe einen Zeilenumbruch. Ist der Zeilenumbruch nicht erwünscht, ist die Anweisung Serial.print() zu nutzen.

Digitale Ausgänge

digitaler Ausgang, digitalWrite(), pinMode(), OUTPUT

Der Arduino besitzt 14 digitale Pins (Pin D0 bis Pin D13). Diese können als Ausgänge verwendet werden.

Die digitalen Pins 0 und 1 sollten vermieden werden, da sie für die serielle Kommunikation (TX und RX) mit dem Computer reserviert sind. Wenn man diese Pins für andere Zwecke nutzt, können Fehlfunktionen auftreten.

Ein digitaler Pin wird als Ausgang mit der Funktion pinMode () konfiguriert.
Ein Beispiel:

cpp Copy code

```cpp
void setup() {
  // Setze Pin 13 als Ausgang
  pinMode(13, OUTPUT);
}

void loop() {
  // Schalte Pin 13 auf HIGH (5V)
  digitalWrite(13, HIGH);
  // Warte 1 Sekunde
  delay(1000);
  // Schalte Pin 13 auf LOW (0V)
  digitalWrite(13, LOW);
  // Warte 1 Sekunde
  delay(1000);
}
```

Das Programm bewirkt, dass die an Pin 13 angeschlossene LED im Sekundentakt ein- und ausgeschaltet wird, was einem Blinken entspricht. Dieses Blinken wird so lange fortgesetzt, wie der Arduino eingeschaltet ist.

Ausgangsbelastung (Stromstärke)

Stromstärke, Belastung

Ein digitaler Ausgang kann kurzzeitig maximal 40mA ausgeben. Das ist kein Wert für einen Dauerbetrieb. Eine sichere Empfehlung sind 20mA. Die Summe aller Ströme über mehrere Ausgänge sollte 200mA nicht überschreiten. Die Werte gelten auch für die PWM-Pins.

Werden höhere Stromstärken benötigt, ist es ratsam, externe Schaltungen wie Transistoren oder Relais zu verwenden, während der Arduino-Ausgang nur das Steuersignal liefert. Ein Fallbeispiel ist hier zu finden:

https://rotering-net.de/tut/arduino/led-mit-transistor-steuern.html

Der Pin 5V ist unmittelbar mit dem Festspannungsregler NCP1117 des Arduino Uno R3 verbunden und kann Stromstärken ≤ 200mA bereitstellen, wenn andere Pins nicht nenneswert belastet sind.

Variablen: Platzhalter vom Dienst

Variable, Datentyp, int

Eine Variable kann als ein Platzhalter betrachtet werden, der verwendet wird, um Daten in einem Programm zu speichern und später darauf zuzugreifen.

Die Variable hat einen Namen, einen Datentyp und einen Wert. Ähnlich wie ein Etikett auf einem Ordner ermöglicht der Name der Variable dem Programm, auf die gespeicherten Daten zuzugreifen. Der Datentyp bestimmt, welche Art von Informationen in der Variable gespeichert werden können, ähnlich wie ein Etikett angibt, was in einem Ordner gespeichert ist (z.B. Dokumente, Bilder usw.).

Der Wert der Variable kann während der Ausführung des Programms geändert werden, ähnlich wie der Inhalt eines Ordners aktualisiert oder geändert werden kann.

Man könnte eine Variable pinNummer nutzen, die den Wert 13 speichert und den digitalen Pin 13 als Ausgang anspricht.

Ein Beispiel:

cpp Copy code

```cpp
int pinNummer = 13; // Variable für die Pinnummer

void setup() {
  // Setze Pin als Ausgang
  pinMode(pinNummer, OUTPUT);
}

void loop() {
  // Schalte Pin auf HIGH (5V)
  digitalWrite(pinNummer, HIGH);
  // Warte 1 Sekunde
  delay(1000);
  // Schalte Pin auf LOW (0V)
  digitalWrite(pinNummer, LOW);
  // Warte 1 Sekunde
  delay(1000);
}
```

Variablennamen sind zudem leichter zu merken als „nackte“ Zahlen. Clever gewählt sind solche Variablennamen, die Rückschlüsse auf die Verwendung zulassen.

Array: das Sammelbecken

Array, Indizierung

Ein Array ist eine Variable, die mehrere Werte gleichen Datentyps beinhalten kann. Die Werte können über Indizierung abgerufen werden.

Ein Beispiel:

cpp Copy code

```cpp
int quadratzahlen[] = {1, 4, 9, 16, 25};

void setup() {
  Serial.begin(9600);
}

void loop() {
  Serial.println(quadratzahlen[2]);
  delay(1000);
}
```

Das Programm initialisiert ein eindimensionales Array mit dem Namen quadratzahlen[] mit den Werten 1, 4, 9, 16 und 25. Alle Werte sind vom Datentyp int, also ganzzahlige Werte.
Die serielle Kommunikation wird mit einer Baudrate von 9600 Bits pro Sekunde gesetzt. In der Dauerschleife (loop) wird das dritte Element des Arrays über den Serial-Monitor ausgegeben. Danach erfolgt eine Verzögerung um 1 Sekunde.

Der entscheidende Vorteil eines Arrays liegt darin, dass es mehrere Werte des gleichen Datentyps unter einem einzigen Variablennamen speichern kann.
Dies erleichtert die Verwaltung von Daten und vereinfacht den Code. Zudem können Arrays mit einer Schleife durchlaufen werden, um z.B. die Werte zu bearbeiten oder bestimmte Werte zu suchen.

Die Indizierung beginnt mit dem Wert 0.
Das Beispielprogramm gibt daher den Wert 9 aus!

Datentyp

Datentyp, int, long, float, char, String, ASCII

Ein Datentyp definiert die Art der Daten, die in einer Variable gespeichert werden können.

Datentyp	Werte	Wertebereich	Beispiel	Speicherplatz
int	Ganze Zahlen	-32.768 bis 32.767	int zahl = 42;	2 Bytes
float	Gleitkommazahlen	ca. +/- 3.402E+38	float zahl = 3.1415;	4 Bytes
char	Einzelne Zeichen	-128 bis 127	char zeichen = ´A´;	1 Byte
string	Zeichenketten	-	string text = “Hallo“;	Variabel
bool	Wahrheitswerte	true oder false	bool wahr = true;	1 Byte

Häufiger genutzte Datentypen sind:

Warum beginnt der Wertebereich des Datentyps char bei -128?

Einzelne Zeichen vom Typ char werden über den ASCII-7 Code (American Standard Code for Information Interchange) Zahlenwerten zugeordnet. Diese Zuordnung erlaubt die digitale Darstellung von Buchstaben, Ziffern, Sonderzeichen und Steuerzeichen.
Der ASCII-7 Code umfasst 128 Zeichen (7 Bit). 1 Byte (8 Bit) umfasst den Wertebereich 0 bis 255. Das führende Bit wird als Vorzeichenbit genutzt. Somit sind negative ganze Zahlen darstellbar. Der Werteberich ändert sich von 0 … 255 auf -128 … 127.
Der erweitere ASCII Code umfasst 256 Zeichen. Die Darstellung negativer Zahlen ist nicht vorgesehen, dafür besteht die Möglichkeit Zeichen/Symbole anderer Sprachen darzustellen.
Vertiefende Einblicke in die Thematik ASCII Code sind z.B. hier zu finden:
https://www.ascii-code.com/de

Es ist unüblich Zahlen mit dem Datentyp char zu repräsentieren. Die Datentypen int und float sind bezüglich Wertebereich und Lesbarkeit von Code vorteilhafter.
Wenn Zeichenketten verarbeitet werden sollen, ist die Verwendung des Datentyps string sinnvoll, da zusätzliche Funktionen und Operationen speziell für Zeichenketten genutzt werden können. Bei einzelnen Zeichen ist der Datentyp char speichereffizienter.

Datentyp const (Sonderrolle)

const, Konstante

Der const-Qualifizierer wird genutzt, um Variablen als konstant zu deklarieren. Während der Laufzeit kann der Wert nicht geändert werden, bzw. der Compiler gibt beim Versuch einer Änderung bereits vor Programmausführung eine Fehlermeldung aus. const ist in Kombination mit anderen Datentypen zu verwenden. Hier ein Beispiel:

```cpp
const float pi = 3.1415;
```

#define

#define

#define gehört nicht zu den Datentypen, wird aber häufig genutzt, um Aliase einzuführen. Der Code kann dadurch lesbarer werden.

Im vorherigen Abschnitt wurde der Datentyp const für die Initialisierung der Variablen pi mit dem Wert 3.1415 genutzt. Folgendes Beispiel bewirkt selbiges durch Nutzung von #define:

```cpp
#define pi 3.1414
```

Die #define Anweisung führt lediglich zu einer Textersetzung. Das bedeutet, dass während des Kompilierens jedes Vorkommen von pi im Code global durch den festgelegten Wert, in diesem Fall 3.1415, ersetzt wird.
Dies hat keine direkten Auswirkungen auf den Speicher während der Laufzeit des Programms.
Es ist wichtig zu beachten, dass die #define Anweisung keine Typinformationen enthält. Es wird empfohlen, den Datentyp const für Konstanten zu verwenden. Potenzielle Probleme in Ausdrücken (z.B. Berechnungen) mit anderen Typen können dadurch vermieden werden.

#define kann auch für Makros genutzt werden. Makros sind komplexe Ausdrücke. Im Beispiel wird jedes Vorkommen von SQUARE(x) im Code durch (x * x) ersetzt.

```cpp
#define SQUARE(x) (x * x)
```

Im Allgemeinen ist für komplexe Ausdrücke die Nutzung von Funktionen sinnvoller, da folgende Eigenschaften vorteilhafter sind:

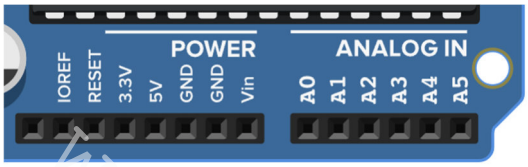
- explizierter Rückgabetyp
- Typeninkompatibilitäten werden während des Kompilierens erkannt
- Notwendigkeit von Klammern, was die Klarheit und Vermeidung von Fehlern verbessert
- begrenzter Gültigkeitsbereich
- explizierter Aufruf und dadurch mehr Kontrolle

#define führt eine Textersetzung durch. Aliase erleichtern die Lesbarkeit von Programmen. In komplexen Ausdrücken können durch Inkompatibilitäten und Gültigkeitsbereiche Probleme auftreten.

Analoge Eingänge: Werte lesen

analoger Eingang, analogRead()

Analoge Eingänge ermöglichen kontinuierliche analoge Signale, wie beispielsweise Spannungen, zu messen.
Diese Eingänge können Werte zwischen 0 und 5 Volt interpretieren und in digitale Werte umwandeln, die der Arduino verarbeiten kann.



Die Umwandlung eines analogen Signals in einen digitalen und gestuften Wert geschieht durch einen sogenannten Analog-Digital-Umsetzer, kurz ADU. Häufig ist auch die Kurzform ADC (C = Converter) zu finden.

Ein analoger Eingang wird mit der Funktion analogRead () eingelesen. Diese Funktion gibt einen Wert zwischen 0 und 1023 zurück, basierend auf der gemessenen Spannung, wobei 0 für 0 Volt und 1023 für die Referenzspannung (5 Volt) steht.
Hier ist ein kurzes Beispiel, wie ein analoger Eingang am Arduino eingelesen wird:

cpp — Copy code

```cpp
// Der analoge Eingang wird am Pin A0 angeschlossen
int analogPin = A0;
// Hier wird der gemessene analoge Wert gespeichert
int analogValue;

void setup() {
  // Initialisierung der seriellen Kommunikation
  Serial.begin(9600);
}

void loop() {
  // Liest den analogen Wert
  analogValue = analogRead(analogPin);
  // Gibt den Wert über die serielle Schnittstelle aus
  Serial.println(analogValue);
  // Kurze Pause
  delay(1000);
}
```

Der Code misst den Wert der analogen Größe, die an Pin A0 angeschlossen ist, und gibt diesen über die serielle Schnittstelle aus. Der Arduino wiederholt diesen Vorgang in der loop()-Funktion in einer Endlosschleife.

Der Pin Vin ist kein analoger Eingang! Er dient der Versorgung des Arduino über eine externe Spannungsquelle, wie z.B. einer Batterie. Für die meisten Arduino-Boards sind Spannungen im Bereich 7V … 12V zulässig.

ADU (bzw. ADC)

ADU, ADC, Analog-Digital-Umsetzer, Auflösung

Der ADU (Analog-Digital-Umsetzer) steckt hinter den analogen Eingängen und setzt analoge Signal in gestufte Werte um. Diese Eingänge ermöglichen es dem Arduino, elektrische Signale von Sensoren zu lesen, die physikalische Größen wie beispielsweise Helligkeit, Temperatur oder Feuchtigkeit messen.

Die Auflösung des ADU gibt an, wie fein ein analoges Eingangssignal in digitale Werte umgesetzt werden kann. Sie wird in Bits gemessen.
Der Arduino mit einem 10-Bit-ADU besitzt eine Auflösung von 1024 (weil $2^{10} = 1024$).
Das Eingangssignal wird in 1024 Schritte unterteilt (0 bis 1023).

Der Arduino arbeite mit einer Referenzspannung von 5V. Daher entspricht die Auflösung 4,88mV (5V:1024 = 0,0048828125V).

Die analogen Eingangssignale sollten Werte größer 5 Volt nicht annehmen können. Dies ist durch die Sensoren oder anderweitige elektronische Beschaltung der analogen Eingänge sicherzustellen. Werte größer 5 Volt werden pauschal mit dem Wert 1023 repräsentiert und gefährden den Mikrocontroller.

Beispiel 1:

Die Eingangsspannung an einem analogen Eingang beträgt 1V. Der ADU liefert einen Wert von ca. 204:

$$ADU\_Wert = \frac{analoge\ Eingangsspannung}{Referenzspannung} \cdot 1023 = 204$$

Je nach Genauigkeit und Stabilität der Referenzspannung kann der ADU-Wert etwas abweichen. Bei auffällig größeren Abweichungen ist die Versorgung zu überprüfen.
Häufig wird die Versorgung über die USB-Schnittstelle eines PC hergestellt. Die USB-Nennspannung wird mit 5V angegeben, nach Spezifikation für USB 3.x sind aber Werte im Bereich 4,45V … 5,25V.
Tipp: Steckernetzteil mit geprüfter Spannung 7V … 12V nutzen! Die 5V werden dann über den Spannungsregler des Arduino erzeugt.

Beispiel 2:
Der ADU gibt einen Wert von 720 aus. Die analoge Eingangsspannung besitzt einen Wert von 3,51V:

$$Eingangsspannung = 0{,}00488V \cdot 720 = 3{,}5136V$$

Ein Beispiel:

cpp Copy code

```cpp
void setup() {
  // Initialisiere die serielle Kommunikation mit 9600 Baud
  Serial.begin(9600);
}

void loop() {
  // Lies den Wert vom analogen Pin A0
  int analogWert = analogRead(A0);

  // Rechne den Wert in Spannung um (5V als Referenzspannung)
  float spannung = analogWert * (5.0 / 1023.0);

  // Gib den Text "Analogwert: " aus
  Serial.print("Analogwert: ");
  // Gib den gelesenen analogen Wert aus
  Serial.println(analogWert);

  // Gib den Text "Spannung: " aus
  Serial.print("Spannung: ");
  // Gib die Spannung auf 2 Dezimalstellen genau aus
  Serial.print(spannung, 2);
  // Gib den Text "V" und einen Zeilenumbruch aus
  Serial.println("V");

  // Warte 1 Sekunde, bevor die nächste Messung durchgeführt wird
  delay(1000);
}
```

Der Arduino Sketch liest einen analogen Wert von Pin A0 ein. Dieser Wert entspricht einer Spannung, die an diesem Pin anliegt.
Der eingelesene Wert wird in eine Spannung umgerechnet und über den Serial Monitor ausgegeben. Zuerst wird der rohe Analogwert angezeigt, gefolgt von der umgerechneten Spannung mit zwei Dezimalstellen Genauigkeit.
Das Programm wartet 1 Sekunde, bevor die nächste Messung durchgeführt wird. Dieser Zyklus wiederholt sich kontinuierlich.

Analoge Ausgänge

analoges Ausgangssignal, PWM, Puls-Weiten-Modulation, analogWrite()

Der Arduino kann analoge Signale erzeugen. Dafür werden PWM-Ausgänge (digitale Ausgänge mit einem ~) genutzt, um ein analoges Signal zu imitieren.

Der Arduino erzeugt ein pulsierendes Signal (Abfolge von Ein- und Ausschalten), bei dem die Pulsweite (HIGH) variiert. Das PWM-Signal besitzt die Werte 0 Volt (LOW) oder 5 Volt (HIGH). Der durchschnittliche Spannungswert ist quasi-analog und für viele Anwendungen ausreichend.

Hier ein Beispiel:

cpp Copy code

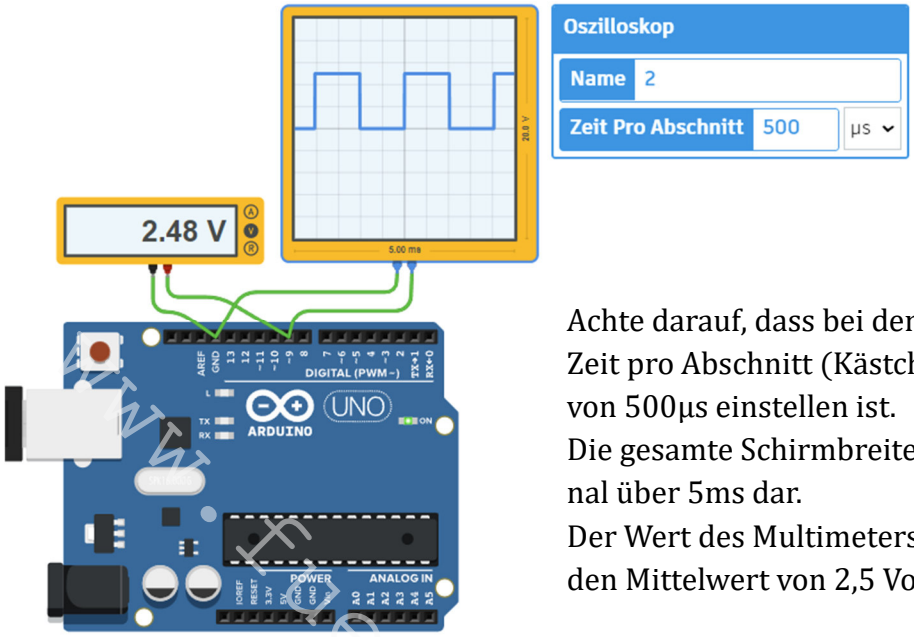
```cpp
int pwmPin = 9;

void setup() {
  pinMode(pwmPin, OUTPUT);
}

void loop() {
  analogWrite(pwmPin, 128);
}
```

Das Programm konfiguriert den Pin 9 als Ausgang und erzeugt mit der Anweisung analogWrite() ein PWM-Signal mit einem Tastgrad von 50% (Wert 128), was einer mittleren Spannung von 2,5 Volt entspricht.

Tastgrad 0%:	mittlere Spannung: 0 Volt	analogWrite(pwmPin, 0);
Tastgrad 50%	mittlere Spannung: 2,5 Volt	analogWrite(pwmPin, 128);
Tastgrad 100%:	mittlere Spannung: 5 Volt	analogWrite(pwmPin, 255);

Der Tastgrad (quasi mittlere Spannung) wird über die Werte 0 bis 255 eingestellt, wobei die mittlere Spannung zwischen 0% bis 100% von 5V beträgt.

Tinkercad: Mittlere Spannung und pulsierendes Signal passend zum Beispiel-Code.



Achte darauf, dass bei dem Oszilloskop die Zeit pro Abschnitt (Kästchen) auf einen Wert von 500µs einstellen ist.
Die gesamte Schirmbreite stellt das PWM-Signal über 5ms dar.
Der Wert des Multimeters schwankt etwas um den Mittelwert von 2,5 Volt.

Aufgabe: Variiere die Werte in der Anweisung analogWrite() und beobachte die Anzeige des Multimeters (Spannungsmessgerät) und das pulsierende Signal.

Arithmetik und vergleichende Operatoren

Funktionen

Funktionen, Modularität, spezifische Aufgaben

Funktionen sind aus mehreren Gründen nützlich.
In der Regel wird Code zusammengefasst, der mehrmals ausgeführt werden muss. Eine solche Funktion kann immer wieder aufgerufen werden. Eine Wiederverwendbarkeit ist leicht möglich.
Die Modularität zerlegt das Programm in kleinere Teile mit spezifischen Aufgaben. Die Lesbarkeit des Programms wird erleichtert.
Wenn ein Fehler auftritt, ist es einfacher, in der Funktion selbst zu suchen, anstatt den gesamten Programmcode durchzugehen.

Funktionen ohne Rückgabewert

Rückgabewert

Funktionen ohne Rückgabewert (kurz Rückgabe), sind Zusammenfassungen von Codezeilen, die wiederholt genutzt werden können.
Funktionen ohne Rückgabe besitzen einen Datentyp. Der Typ void, übersetzt „Leere“, gibt an, dass die Funktion keinen Wert zurückliefert. Verrückt: Ein Datentyp für etwas, was es nicht gibt.
void setup() und void loop() sind Funktionen ohne Rückgabe.

Ein Beispiel:

cpp Copy code

```cpp
// Pin, an dem die LED angeschlossen ist
int ledPin = 13;

// Initialisierung der LED
void setup() {
  pinMode(ledPin, OUTPUT);
}

// Funktion zum Blinken der LED
void blinkLED() {
  digitalWrite(ledPin, HIGH); // LED einschalten
  delay(1000); // 1 Sekunde warten
  digitalWrite(ledPin, LOW); // LED ausschalten
  delay(1000); // 1 Sekunde warten
}

// Hauptprogramm
void loop() {
  blinkLED(); // Die blinkLED-Funktion aufrufen
}
```

In diesem Beispiel gibt es eine Funktion namens blinkLED(), die keine Rückgabe hat (void). Diese Funktion schaltet eine LED ein, wartet eine Sekunde, schaltet die LED aus und wartet erneut eine Sekunde.

Die Funktion setup() initialisiert den Pin, an dem die LED angeschlossen ist, und gibt an, dass dieser Pin als Ausgang verwendet wird. Die Funktion loop() ruft einfach blinkLED() auf, um die LED blinken zu lassen.

Funktionen mit Rückgabewert

Rückgabewert

Ein Beispiel:

cpp Copy code

```cpp
int berechneSumme(int a, int b) {
  int summe = a + b;
  return summe;
}

void setup() {
  Serial.begin(9600);
}

void loop() {
  int zahl1 = 5;
  int zahl2 = 3;
  int ergebnis = berechneSumme(zahl1, zahl2);

  Serial.print("Summe: ");
  Serial.println(ergebnis);

  delay(1000);
}
```

In loop() werden zwei Variablen (zahl1, zahl2) mit den Werten 5 und 3 initialisiert. Beide Zahlen sind vom Datentyp integer (int), also ganzzahlig.
Die Funktion berechneSumme(int a, int b) nimmt durch ihren Aufruf in loop() die beiden Zahlen zahl1 und zahl2 entgegen und weist sie „funktionsintern" den Variablen a und b zu. Durch die Anweisung return() wird die Summe der beiden Zahlen zurückgegeben.

Die Funktion berechneSumme() muss nicht vor setup() definiert werden. Nach loop() wäre auch eine Option.
In diesem Beispiel werden nur ganze Zahlen genutzt, so dass alle Werte vom Datentyp int sind.

Zusammenfassend berechnet der Sketch die Summe von 5 und 3 und gibt diese über den Serial Monitor aus. Dieser Vorgang wird in einer endlosen Schleife wiederholt.

Zufallsfunktion

Zufallsfunktion, Zufallswerte

Der Zufallszahlengenerator im Arduino basiert auf einem Algorithmus. Dass bedeutet, dass er unter gleichen Bedingungen immer die gleichen Zahlen erzeugen würde. Im folgenden Sketch wird die Anweisung randomseed() verwendet, um den Zufallszahlengenerator zu initialisieren.

Ein Beispiel:

cpp Copy code

```cpp
void setup() {
  Serial.begin(9600);
  // Initialisiere den Zufallszahlengenerator
  randomSeed(analogRead(A0));
}

void loop() {
  // Zufallszahl von 0 bis 100
  int zufallszahl = random(101);
  Serial.print("Zufallszahl: ");
  Serial.println(zufallszahl);
  delay(1000); // Warte 1 Sekunde
}
```

Die Werte eines offenen analogen Einganges (hier A0), sind nicht konstant, sondern ändern sich ständig. Diese Unbeständigkeit wird genutzt, um bei jedem Aufruf des Zufallszahlengenerators andere Startbedingungen zu besitzen. Die Zufälligkeit der generierten Zahlen ist dadurch verbessert.

Im Sketch generiert die Anweisung random(101) eine Zufallszahl im Bereich 0 bis 100 und gibt diese über den Serial Monitor aus.

Der Vorteil der Anweisung random() gegenüber einem offenen Eingang ist, dass der Wertebereich vorgegeben werden kann.

Programmablauf kontrollieren

Sequenzielle Ausführung, Kontrolle

Grundsätzlich wird ein Sketch von Anfang bis Ende in einer linearen Abfolge abgearbeitet, Anweisung für Anweisung, Zeile für Zeile, … man spricht von einer sequenziellen Ausführung.

Ein Nachteil ist die begrenzte Flexibilität. Es ist schwer möglich, auf sich ändernde Bedingungen zu reagieren, wenn der Code einer vordefinierten Reihenfolge folgt. Eine Gliederung und Strukturierung des Codes in Funktionsbereiche ist nicht möglich.

Um diese Nachteile zu überwinden, gibt es Techniken, um den Programmablauf zu kontrollieren. Analog zu Wanderwegen werden Weggabeln benötigt, mit eindeutigen und zu prüfenden Bedingungen und dazugehörige eindeutige Anweisungen.

Hier ein bildhaftes Beispiel:

Bedingung	**Anweisung**
Du bist fit!	Lauf den langen Weg! 
Du hast kurze Beine!	Nutze das Fahrrad! 
Deine Lunge pfeift!	Setz dich hin! 

Bedingte Anweisung: if … else

if … else, bedingte Anweisung

Die if … else Anweisung ermöglicht Entscheidungen im Programmablauf zu treffen, basierend auf bestimmten Bedingungen.

Der nachfolgende Sketch überwacht einen am Pin 2 angeschlossenen Taster.
Wird der Taster gedrückt, leuchtet die auf dem Arduino befindliche LED an Pin 13. Ist der Taster nicht gedrückt, bleibt die LED ausgeschaltet. Eine ausführlichere Beschreibung ist auf nachfolgender Seite zu finden.

Hier der Sketch:

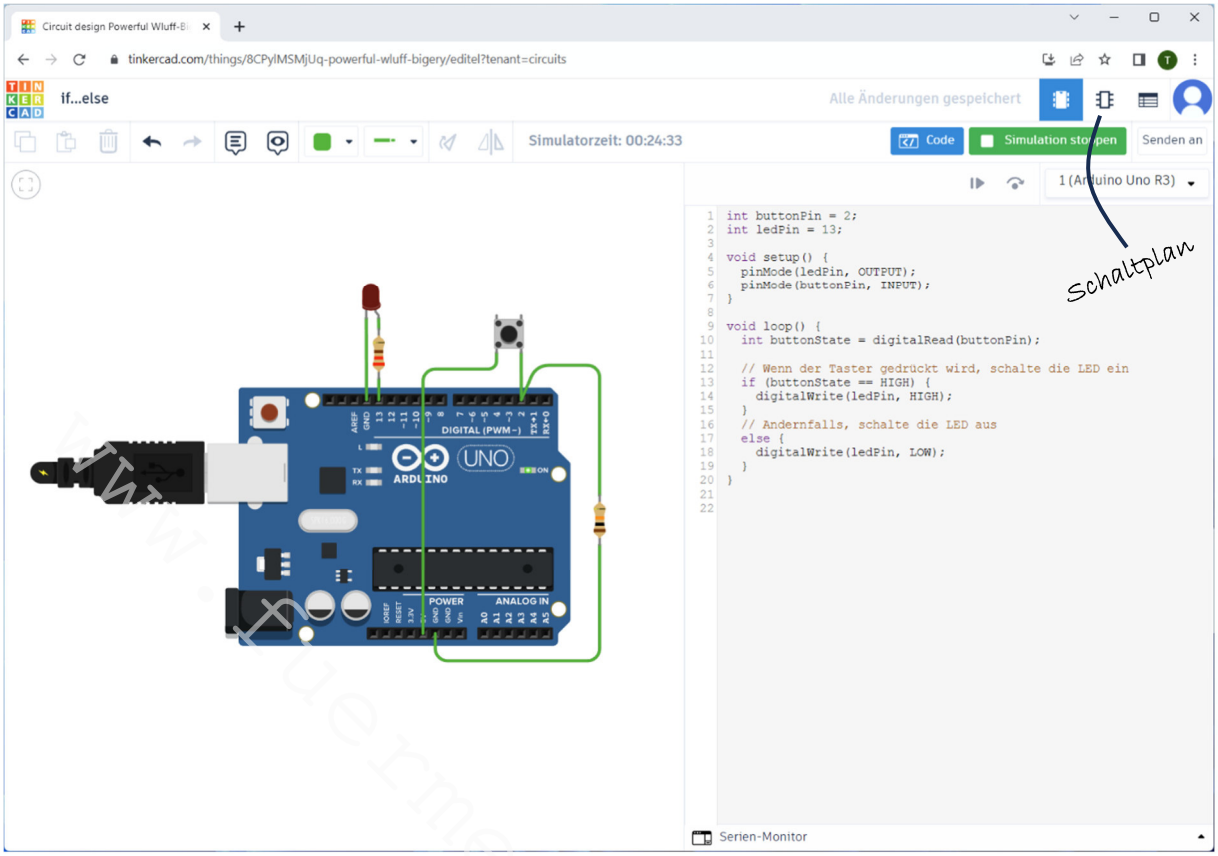
cpp Copy code

```cpp
int buttonPin = 2;
int ledPin = 13;

void setup() {
  pinMode(ledPin, OUTPUT);
  pinMode(buttonPin, INPUT);
}

void loop() {
  int buttonState = digitalRead(buttonPin);

  // Wenn der Taster gedrückt wird, schalte die LED ein
  if (buttonState == HIGH) {
    digitalWrite(ledPin, HIGH);
  }
  // Andernfalls, schalte die LED aus
  else {
    digitalWrite(ledPin, LOW);
  }
}
```

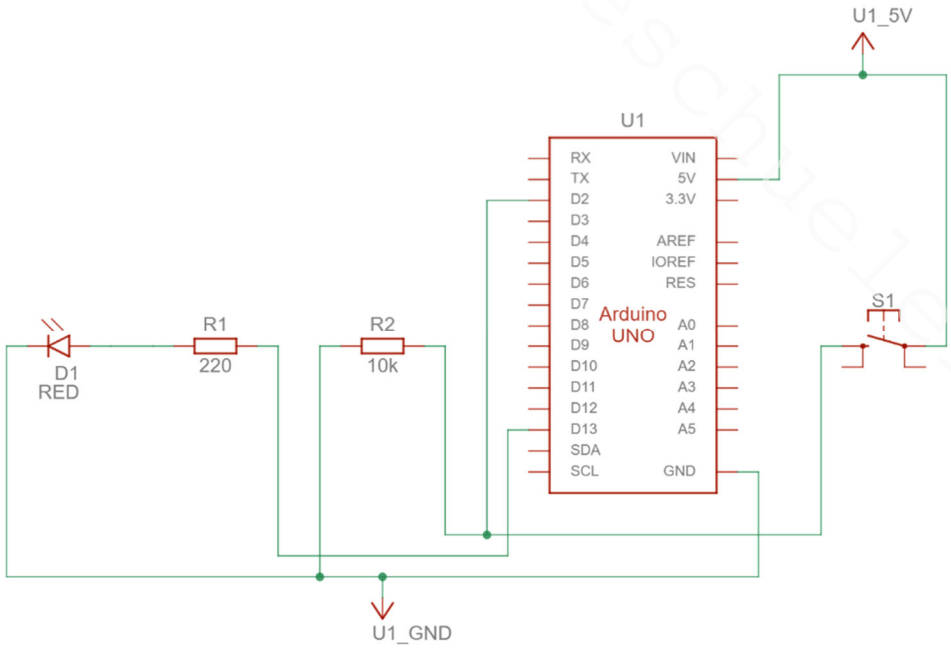
Pin D2 fungiert als Eingang. Bei Betätigung des Tasters wird die Spannung von 5 Volt auf Pin D2 geführt. 5 Volt werden als HIGH interpretiert (logisch 1) und der Variablen buttonState wird dieser Zustand zugeordnet. In diesem Fall wird der als Ausgang arbeitende Pin D13 auf HIGH (5 Volt) gesetzt. Pin D13 führt dann selbst 5 Volt. Die angeschlossene LED leuchtet.
Im anderen Fall (Taster nicht gedrückt) wird Pin D13 auf LOW (logisch 0) gesetzt, die LED ist aus.

if … else kann mehr als nur eine Bedingung prüfen. An dieser Stelle der Verweis:
https://www.arduino.cc/reference/en/language/structure/control-structure/else/

Nachfolgend die Darstellung der Simulation in Tinkercad.
An Pin D13 ist über einen zusätzlichen Widerstand (220Ω) eine baulich „dickere“ rote LED angeschlossen. Zwischen Pin D2 und GND liegt ein sogenannter Pull-down Widerstand (10kΩ). Auf elektrotechnische Grundlagen wird hier nicht weiter eingegangen.



Tinkercad bietet die Möglichkeit der elektrotechnischen Dokumentation, indem eine schematische Ansicht (Schaltplan) erstellt werden kann.



Bestimmte Wiederholung: for-Schleife

for-Schleife, bestimmte Wiederholung

Die for-Schleife wiederholt einen bestimmten Block Codezeilen, solange eine bestimmte Bedingung erfüllt ist. Sie besteht aus drei Hauptteilen:

- Initialisierung
- Bedingung
- Inkrementierung

In der Initialisierung wird einer Variablen ein Startwert zugewiesen. Diese Variable wird verwendet, um den Durchlauf der Schleife zu steuern.

In der Bedingung wird überprüft, ob die Schleife weiterhin ausgeführt werden soll. Solange diese Bedingung wahr (true) ist, wird die Schleife ausgeführt. Wenn die Bedingung falsch (false) wird, endet die Schleife.

Die Inkrementierung aktualisiert die Variable, um auf das nächste Element oder den nächsten Zustand in der Schleife zu zeigen. Die Inkrementierung beeinflusst, wie oft die Schleife durchlaufen wird.

Ein Beispiel:

cpp Copy code

```cpp
for(int i = 0; i < 5; i++) {
  // Code, der wiederholt ausgeführt wird
}
```

Initialisierung:	Die Variable i startet mit dem Wert 0.
Bedingung:	Es wird geprüft, ob $i < 5$ zutrifft.
Inkrementierung:	Die Variable i wird um 1 erhöht.

Die Schleife wird fünfmal durchlaufen, für die Fälle $i = 0$ bis $i = 4$.

Die for-Schleife ist nützlich, um Aufgaben zu automatisieren, beispielsweise das Durchlaufen von Arrays oder das Steuern von Sensoren mit wiederholten Abfragen.

Bestimmte Wiederholung: while-Schleife

bestimmte Wiederholung, while-Schleife

Die while-Schleife ähnelt der for-Schleife. Anders als die for-Schleife, besitzt die while-Schleife keine explizite Initialisierung oder Inkrementierung.

Ein Beispiel:

```cpp
int i = 0;
while (i < 5) {
  // Code, der wiederholt ausgeführt wird
  i++;
}
```

Initialisierung: int i = 0 initialisiert die Variable i mit dem Wert 0 außerhalb der Schleife.

Bedingung: while (i < 5) überprüft, ob i kleiner als 5 ist. Solange diese Bedingung wahr (true) ist, wird der Code innerhalb der Schleife ausgeführt.

Inkrementierung: i++ erhöht i nach jedem Durchlauf der Schleife um 1.

Die Schleife wird fünfmal durchlaufen, für die Fälle i = 0 bis i = 4.
Hier ein konkreteres Beispiel:

```cpp
int zahl = 1;
int summe = 0;

void setup() {
  Serial.begin(9600);
}

void loop() {
  while (zahl <= 100) {
    summe = summe + zahl;
    zahl++;
  }

  Serial.print("Die Summe der Zahlen von 1 bis 100 ist: ");
  Serial.println(summe);

  delay(1000);  // Eine Sekunde warten
}
```

Der Sketch berechnet in der while-Schleife die Summe der Zahlen 1 bis 100. Nach dem 100sten Durchlauf ist die Variable zahl = 101, so dass der folgende Durchlauf (101ste) abgebrochen wird. Der Sketch setzt die Ausführung mit der Ausgabe auf dem Serial Monitor fort.

Bestimmte Wiederholung: do … while-Schleife

bestimmte Wiederholung, do … while-Schleife

Die do … while Schleife garantiert, dass der Code mindestens einmal ausgeführt wird, selbst wenn die Bedingung von Anfang an falsch ist. Die Begründung ist einfach: Die Bedingung wird erst am Ende der Schleife geprüft.

Die Struktur der do … while Schleife:

cpp Copy code

```cpp
void setup() {
  Serial.begin(9600);
}

void loop() {
  int counter = 0;

  do {
    Serial.println("I am an Arduino!");
    counter++;
  } while (counter < 5);

  delay(1000);  // Eine Sekunde warten
}
```

Die Ausgabe „I am an Arduino“ erfolgt mindestens einmal, aber nicht mehr als fünfmal.

Die do … while Schleife ist nützlich, wenn sichergestellt werden soll, dass ein bestimmter Code mindestens einmal ausgeführt wird, unabhängig von der Bedingung.
Sie wird oft verwendet, wenn eine Benutzereingabe verarbeitet werden soll und sichergestellt sein muss, dass die Eingabe zumindest einmal durchgeführt wurde, bevor die Bedingung überprüft wird.

Programm/Schleife stoppen/verlassen

Stopp, Verlassen, return, break, exit(0), while(1)

Der Stopp bzw. das Verlassen von Schleifen oder des Programms kann an bestimmten Stellen notwendig sein.

Wird innerhalb einer Schleife ein bestimmtes Ziel erreicht und weitere Durchläufe sind nicht nötig, ist ein Verlassen der Schleife sinnvoll.

Unter anderen Bedingungen kann es notwendig sein, den Programmablauf zu stoppen oder abzubrechen.

Die folgend dargestellten Möglichkeiten besitzen unterschiedliche Eigenschaften und sollten bedacht eingesetzt werden.

- **return**

Wenn der Sketch in einer Schleife arbeitet, kann durch die Anweisung return das Hauptprogramm loop() verlassen werden.

Ein Beispiel:

```cpp
void setup() {
  Serial.begin(9600);
}

void loop() {
  for (int i = 0; i < 5; i++) {
    if (i == 3) {
      Serial.println("Schleife wird bei i = 3 verlassen.");
      return; // Verlasse die loop-Funktion
    }
    Serial.print("Iteration: ");
    Serial.println(i);
  }
}
```

Die for-Schleife läuft von 0 bis 4. Erreicht i den Wert 3, wird die if-Bedingung erfüllt. Die return-Anweisung beendet die loop-Funktion, was zur Folge hat, dass das Programm über void setup() zum Anfang der loop-Funktion zurückkehrt.

- **break**

Ein allgemein gehaltenes Beispiel:

```cpp
void loop() {
  for (int i = 0; i < 10; i++) {
    if (bedingung) {
      // Hier wird die Schleife vorzeitig beendet
      break;
    }
    // Weitere Anweisungen innerhalb der Schleife
  }
  // Weitere Anweisungen nach der Schleife
}
```

Die for-Schleife wird vorzeitig beendet, wenn die Bedingung erfüllt ist. Der Rest des Codes innerhalb der Schleife wird übersprungen. Die Programmausführung wird nach der Schleife fortgesetzt.

- **exit(0)**

exit(0) beendet das Programm sofort. Es wird nicht wieder neu aufgenommen.

Ein allgemein gehaltenes Beispiel:

```cpp
void loop() {
  if (bedingung) {
    exit(0); // Beendet das Programm
}
```

Wenn die Bedingung bedingung erfüllt ist, wird exit(0) aufgerufen.

Das gesamte Programm wird beendet und der Mikrocontroller wird in einen Zustand versetzt, in dem er erneut programmiert oder neu gestartet werden muss (Reset-Taster).

- **while(1)**

while(1) ist eine Endlosschleife. Das Programm bleibt an der Stelle stecken.

Ein allgemein gehaltenes Beispiel:

```cpp
void loop() {
  if (bedingung) {
    while(1) {
      // Code bleibt hier stecken
    }
  }
  // Weitere Anweisungen, wenn Bedingung nicht erfüllt
}
```

Es gibt einen Ausweg aus der Endlosschleife durch Anwendung der break-Anweisung.

```cpp
void loop() {
  while(1) {
    if (bedingung) {
      break; // Verlasse die Schleife
    }
  }
  // Code nach der Schleife
}
```

Es ist darauf zu achten, dass die if-Bedingung innerhalb der Schleife irgendwann erfüllt ist, sonst bleibt das Programm in der Endlosschleife stecken.

Mathematische Operationen (Arithmetik)

mathematische Operationen, Arithmetik, Berechnungen

Unter mathematische Operationen sind alle Berechnungen zu verstehen.
Hier ein einfaches Beispiel, das keiner Erläuterung bedarf:

cpp Copy code

```cpp
int a = 5;
int b = 3;
int sum = a + b; // sum wird den Wert 8 haben
```

Grundlegende mathematische Operatoren:

Funktion	Operator
Addition	+
Subtraktion	-
Multiplikation	*
Division	/
Potenz	pow()
Inkrement	++
Dekrement	--
Exponentialfunktion	exp()
Natürlicher Logarithmus	log()
10er Logarithmus	log10()
Wurzelfunktion	sqrt()

Logische Operationen

logische Operationen, UND, ODER, NICHT, true, false, bool

Logische Operationen behandeln Wahrheitswerte (true oder false). Grundsätzlich unterscheiden wir zwischen den logischen Operationen UND, ODER und NICHT. Logische Operationen überprüfen Bedingungen und helfen den Programmablauf zu steuern.

Funktion	Operator
UND	&&
ODER	||
NICHT	!

Hier drei Beispiele für logische Operationen:

```cpp
bool a = true;
bool b = false;
bool result = a && b; // result wird den Wert false haben
```

```cpp
bool a = true;
bool b = false;
bool result = a || b; // result wird den Wert true haben
```

```cpp
bool a = true;
bool result = !a; // result wird den Wert false haben
```

Datentypen, Operationen und logische Fehler

Syntaxfehler, logische Fehler, Cast, völlig irre werden

Wir rechnen anders als der Arduino. Damit ist die Art und Weise der Rechenvorgänge gemeint. Was für uns implizit verständlich und/oder logisch erscheint, ist es für den Arduino nicht. Der Arduino kann nur explizit rechnen, also nach festen und eindeutigen Regeln und Vorschriften der Programmiersprache C++.
Wir müssen explizit daran denken, unsere impliziten Rechenvorgänge dem Arduino im Quellcode explizit deutlich zu machen. Das erfordert Konzentration und ist schwierig.

Auftretende Probleme sind keine Syntaxfehler, die uns der Compiler mit Angabe der Codezeile meldet. Es sind logische Fehler ohne Hinweis. Die Fehlersuche kann schwierig sein.

Hier ein leicht verständliches Beispiel mit drei ganzzahligen Werten:

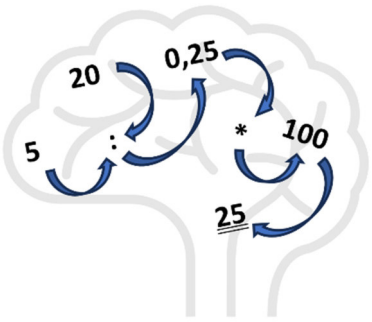
```
1  void setup() {
2    Serial.begin(9600);
3  }
4
5  void loop() {
6    int a = 5;
7    int b = 20;
8    int c = 100;
9    float result = (a / b) * c;
10   Serial.println(result);
11   delay(1000);
12 }
```

Serien-Monitor

```
0.00
0.00
0.00
0.00
0.00
0.00
```

Das Ergebnis ist result = 0 ?

Die Lösung sollte 25 sein. Wo verbirgt sich der logische Fehler?

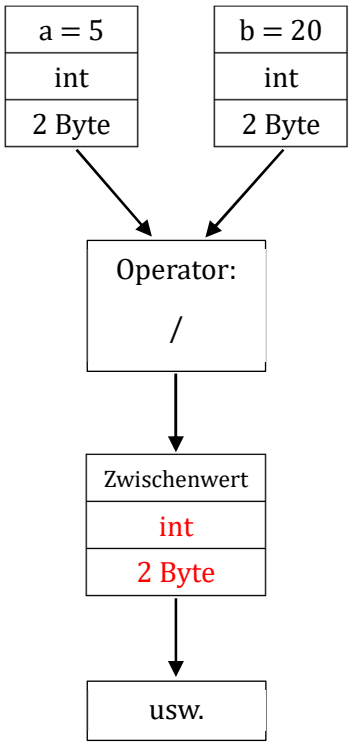


Kopfrechnen ist ein sequenzieller (schrittweiser) Vorgang.

Dabei kümmert sich das menschliche Gehirn nicht um Datentypen und Speicherplatz.
Das Zusammenspiel von Synapsen und Neuronen „macht es schon irgendwie“. Alles automatisch!

Wie sieht es beim Arduino aus?

Der Compiler berücksichtigt die Datentypen der Variablen und „reserviert“ den entsprechenden Speicherplatz im SRAM. Das „Reservieren“ ist bei allen Systemen mit begrenzten Ressourcen wichtig. Der SRAM des Arduino ist eine stark begrenzte Ressource (wie eigentlich alles) und beträgt beim Arduino Uno nur 2KB.
Programmierer müssen sich bewusst sein, wie viel Speicher die Variablen benötigen, um sicherzustellen, dass der verfügbare SRAM ausreicht, um z.B. Programmabstürze zu vermeiden.



Bei Betrachtung der Division ist uns sofort klar, dass der Zwischenwert eine Gleitkommazahl ist. Dem Arduino ist das nicht klar!

Der Compiler muss für den Zwischenwert eine Temporärvariable verwenden, der ein Datentyp und ein Speicherplatz zugewiesen werden muss.
Für die Wahl des Datentyps wird die sogenannte implizite Datentypfestlegung genutzt: Der Compiler legt die Datentypen der beiden verrechneten Variablen a und b zu Grunde.
Die Temporärvariable besitzt somit ebenfalls den Datentyp int.
Vom richtigen Zwischenwert 0,25 werden die Nachkommastellen abgeschnitten und der Wert 0 wird für die weitere Berechnung genutzt. Das Ergebnis der gesamten Berechnung beträgt dann null.
Das ist ein logischer Fehler und liegt in der Verantwortung des Programmierers.

Der Umgang mit Datentypen und Operationen kann zu logischen Fehlern führen.
Es ist sorgfältig darauf zu achten, wie Variablen deklariert und verarbeitet werden.

Mögliche Lösungen:

Für nummerische Variablen pauschal den Datentyp float (Gleitkommazahlen) zu nutzen ist keine sinnvolle Lösung. Neben dem größeren Speicherbedarf wird für die Verarbeitung von Gleitkommazahlen mehr Rechenleistung beansprucht. Die Ausführungsgeschwindigkeit von Programmen nimmt ab.

Eine geeignetere Lösung wäre die Variable a oder die Variable b vom Typ float festzulegen. Die implizite Datentypvergabe führt dann zum Datentyp float für den Zwischenwert.

Eine weitere Möglichkeit bietet die explizite Datentypumwandlung, auch „Cast“ genannt. Im Beispiel wird die Variable zahl1 explizit in float umgewandelt, bevor die Division erfolgt.

cpp Copy code

```cpp
int zahl1 = 5;
int zahl2 = 2;

float ergebnis = float(zahl1) / zahl2;
```

Stichwortverzeichnis

P

R

S

T

U

V

W

Z