Programmierung mit Arduino

Dieses Dokument soll einen grundsätzlichen Überblick über die Arduino-Programmiersprache geben. Es ist in keiner Hinsicht vollständig und ebenso nicht dazu gedacht, die Sprache zu lernen (dazu fehlen zu viele Erklärungen)! Eher sollte es als Referenz für Einsteiger zum Nachschlagen gesehen werden.

Intern verwendet Arduino einen C++-Compiler, wer also C++ beherrscht, dem wird der Übergang sehr leicht fallen.

Programmaufbau

Der grundsätzliche Aufbau eines Arduino-Programms ist sehr simpel.

```
// Präprozesordirektiven
// Globale Variablen hier deklarieren
// Funktionsköpfe

void setup()
{
      // Dieser Code wird einmal ausgeführt.
}

void loop()
{
      // Dieser Code wird immer wieder ausgeführt.
}
```

Die Textzeilen nach dem doppelten Schrägstrich sind Kommentare. Sie werden nicht übersetzt und nicht zum Board hochgeladen; sie dienen nur dem Programmierer, um die Übersichtlichkeit des Programms zu erhöhen. Nach `//` wird der Rest der Zeile zum Kommentar. Eine andere Form der Kommentare sieht so aus:

```
/* Kommentar
Kommentar
Kommentar */
```

Der Kommentar beginnt mit einem `/*` und endet erst bei einem `*/`, Zeilenumbrüche werden ignoriert.

Grundbefehle

Die wichtigsten Befehle bei der Arduino-Programmierung sind hier aufgelistet. Zur Schreibweise muss gesagt werden, dass man das erste Wort (den Rückgabedatentyp) nicht im Programm schreibt. Dieser wird erst wichtig, sobald man zum Beispiel den eingelesenen Wert eines Sensors in einer Variable abspeichern will. Jeder Befehl endet mit einem Semikolon!

```
void pinMode(int pin, int usage);
```

Setzt einen Pin als In- oder Output (INPUT/OUTPUT);

```
void digitalWrite(int pin, boolean state);
```

Setzt einen Pin auf einen bestimmten Wert (HIGH/LOW).

```
void analogWrite(int pin, int value);
```

Setzt einen PWM-Pin (siehe Board!) auf einen bestimmten Wert (0-255).

```
boolean digitalRead(int pin);
```

Liest den Wert eines Pins aus (HIGH/LOW).

```
int analogRead(int pin);
```

Liest den analogen Wert eines Analog-Pins aus (0-1023).

```
void delay(int duration);
```

Wartet die angegebene Zeit in Millisekunden. Dies blockiert den Controller komplett, das heißt, es ist nicht möglich, in dieser Zeit andere Befehle auszuführen.

```
void delayMicroseconds(int duration);
```

Wartet die angegebene Zeit in Mikrosekunden (ansonsten wie `delay`).

Algorithmische Grundstrukturen

```
while(boolean bedingung) {…}
```

Wiederholt den Code in den geschweiften Klammern, solange die Bedingung erfüllt ist (die Prüfung findet beim Eintritt statt).

```
do {…} while (boolean bedingung);
```

Eine endgeprüfte Schleife, das heißt, die Bedingung wird erst am Ende der Schleife überprüft (ansonsten gleich wie `while`). Es ist darauf zu achten, dass diese Schleife als einzige ein Semikolon nach der Bedingung hat!

```
for(int i = 0; i < 10; i++) {…}
```

Wiederholt den Code in den geschweiften Klammern, solange die Bedingung in der Mitte erfüllt ist (die Prüfung findet beim Eintritt statt). Im ersten Teil wird meist einen Variable erstellt und definiert, darauf folgt die Bedingung und als letztes wird der Wert der Variable verringert oder erhöht. Wird meist als gezählte Wiederholung verwendet.

```
if(boolean bedingung) {…}
```

Führt den Code in den geschweiften klammern nur aus, wenn die Bedingung erfüllt ist.

```
if() {…} else {…}
```

Eine Erweiterung zu `if`, der Code in den Klammern hinter `else` wird immer dann ausgeführt, wenn die Bedingung nicht erfüllt war (engl. sonst).

```
if() {…}
else if() {…}
else {…}
```

Eine Erweiterung zu `if`. Erst wird die `if`-Bedingung geprüft, wenn diese nicht erfüllt ist, wird auch noch die `else if`-Bedingung geprüft. Ist keine Bedingung erfüllt, wird `else` ausgeführt. Es ist möglich, beliebig viel `else if`-Abfragen eizubauen, aber es darf immer nur einen `else`-Abschnitt geben (wobei diese auch weggelassen werden kann).

```
switch(int number)
{
case 0: /* Do something */ break;
case 1: /* Do something */ break;
case 2: /* Do something */ break;
default: /* Do something */
}
```

Vergleicht nach einander `number` mit allen `case`-Abfragen (funktioniert nur mit Ganzzahlen). `/* Do something */` kann durch beliebig viel Code ersetzt werden. `default` wird ausgeführt, wenn keine Übereinstimmung gefunden wurde, muss aber immer als letztes geschrieben werden und braucht daher kein `break`. Durch das Weglassen von `break` wird aller nachfolgender Code bis zum nächsten `break` ausgeführt, weitere `case`-Abfragen werden ignoriert. Meist ist dies unerwünscht, sehr selten kann man aber auch bewusst damit arbeiten.

Variablen

Variablen sind Teile des Arbeitsspeichers, die man zum Rechnen oder Zwischenspeichern von Daten verwenden kann. Durch das Schreiben des Schlüsselwortes für den gewünschten Datentyp und darauf folgend einen Namen für die Variable wird die Variable erstellt und Speicherplatz wird reserviert.

```
datentyp nameDerVariable;
```

Liste der wichtigsten Datentypen und deren benötigter Speicherplatz:

Name	**Schlüsselwort**	**Datenbereich**	**Speicher**
Boolean (Wahrheitswert)	bool	true - false	1 Byte
Byte (kleine Zahl)	byte	0 – 255	1 Byte
Charakter (Buchstabe)	char	-128 – 127	1 Byte
Short Integer (Ganzzahl)	Short	-32768 – 32767	2 Byte
Integer (Ganzzahl)	int	-32768 – 32767[1]	2 Byte[1]
Long Integer (große Ganzzahl)	long	-2147483648 – 2147483647	4 Byte
Word (Ganzzahl)	word	0 – 65535	2 Byte
Float (Kommazahl)	float	-3.4028235E+38 – 3.4028235E+38	4 Byte
Double (genaue Kommazahl)	double	-3.4028235E+38 – 3.4028235E+38[1]	4 Byte[1]
String (Zeichenkette)	String	Beliebig viele Buchstaben	

Der Wert einer Variable wird durch die Verwendung eines einfachen `=` gesetzt (dabei ist immer der Datenbereich zu beachten). Es ist möglich, eine Variable schon bei der Deklaration zu definieren, das heißt, ihren Wert sofort zu spezifizieren. Tut man das nicht, sollte man den Wert der Variable auch nicht abrufen, da das zu unerwarteten Ergebnissen führen kann.

```
int var = 0;      // Variable sofort definieren
var = 4;          // Wertzuweisung
```

Durch das Verwenden des Schlüsselworts `const` vor dem Datentyp wird eine Variable als symbolische Konstante gekennzeichnet. Die Variable kann nicht mehr im Verlauf des Programms verändert werden.

```
const int ledPin = 13;
```

Da sich der LED-Pin nicht ändert, wird diese Variable als symbolische Konstante erstellt.

Ein weiteres wichtiges Schlüsselwort ist `unsigned`. Seine Verwendung führt dazu, dass die Variable keine negativen Werte mehr aufnehmen kann (natürlich nur für die Variablen, welche dies vorher konnten). Der Speicherplatz, der dabei gespart wird, wird im positiven Bereich hinzugefügt, sodass der insgesamt belegte Speicher derselbe bleibt.

```
unsigned int xyz;       // Datenbereich: 0 – 65535
```

[1] Speicherplatz und Datenbereich unterscheiden sich von Board zu Bord.
Bei einem Arduino Due ist der Umfang der Variable doppelt so groß wie angegeben.

Arithmetik

Es ist möglich, mit Variablen und Zahlen zu rechnen, dazu stehen alle Grundrechenarten und einige weitere Befehle zur Verfügung:

```
int a = 10;
int b = a + 7;
a = b – 3;
a = a * 5;
a = 100 / b;
a = (a + 4) / b;
```

Außerdem kann man mit einigen Kurzschreibweisen den Code erheblich übersichtlicher gestalten:

```
a++;        // Increment: a = a + 1;
a--;        // Decrement: a = a - 1;
a += x;     // a = a + x; (mit allen Grundrechenarten möglich)
```

Für weitere Rechenarten wie zum Beispiel Wurzeln, Potenzen, Winkelfunktionen oder Ähnliches gibt es auf der Arduino-Website (www.arduino.cc) Befehle und Beispiele.

Logische- und Vergleichsoperatoren Operatoren

Neben den Rechenzeichen gibt es auch noch Vergleichsoperatoren. Meist verwendet man diese innerhalb einer `if`-Abfrage oder um das Ergebnis in einer `boolean`-Variable zu speichern. Einige Beispiele sind:

```
if(a == 4) {…}    // Gleich
if(a != 4) {…}    // Ungleich
if(a < 5) {…}     // Kleiner als
if(a > 7) {…}     // Größer als
if(a <= 8) {…}    // Kleiner oder Gleich
if(a >= 4) {…}    // Größer oder Gleich
if(!(a < 6)) {…}  // ‘!’ negiert
if(bedingungA && bedingungB)  // Logisches UND
if(bedingungA || bedingungB) // Logisches ODER
```

Weitere Operatoren

Es gibt noch andere Operatoren zum Rechnen oder Vergleichen von Zahlen, beispielsweise Bitoperatoren oder ternäre Operatoren. Da diese selten gebraucht werden und teilweise recht kompliziert sind, werden sie hier nicht im Einzelnen aufgeführt. Weitere Informationen finden sich in der offiziellen Dokumentation der Arduino-Programmiersprache.

Kommunikation

Ein Arduino bietet viele Möglichkeiten, um mit dem Computer oder anderen Sensoren und Boards zu kommunizieren. Dabei gibt es vier Kommunikationsarten zu unterscheiden:

I2C, TWI und Wire

Eine Kommunikationsart, bei der zwei Kabel verwendet werden und die jedes Arduino beherrscht. Die beiden benötigten Pins sind auf dem Board mit SDA (Data-Line) und SCL (Clock-Line) beschriftet. Es ist möglich, beliebig viele Geräte an den Bus anzuschließen, da jedes Gerät eine eigene Adresse hat; dazu werden alle parallel geschaltet. Die Bezeichnungen I2C (richtig geschrieben IIC oder I²C) und TWI sind synonym zu gebrauchen. Wire ist die Bibliothek, die es ermöglicht, mit wenigen Befehlen ein Arduino TWI-fähig zu machen. Dazu muss man ganz an den Anfang des Programms die Zeile `#include <Wire.h>` hinzufügen (Erklärung: siehe Präprozessordirektiven). Ein komplettes Beispielprogramm:

```
#include <Wire.h>

void Setup()
{
      Wire.begin();     // Bus als Master initialisieren
      // Als Slave-Gerät: Adresse in die Klammern schreiben
}



void loop()
{
      Wire.beginTransmission(address);   // Beginnt eine Übertragung
      Wire.write(message);    // Sendet ein Byte oder eine Gruppe von Bytes
      Wire.endTransmission();       // Beendet die Übertragung
      Wire.requestFrom(address, length); // Fordert Daten von einem Slave
      while(Wire.available() < length) {}// Warte auf Eintreffen der Daten
      byte data = Wire.read();      // Daten einlesen, evtl. wiederhohlen
      // Daten auswerten
}
```

Da man nur selten mit einem Arduino als Slave arbeitet, verzichte ich hier auf eine Beschreibung aller Befehle. Um herauszufinden, welche Daten ein Sensor erwartet und was er antworten wird, ist ein Blick in das Datenblatt nötig.

SPI

SPI ist eine Schnittstelle, die auch jedem Arduino zur Verfügung steht, meist wird sie für SD-Karten oder GPS-Module verwendet. Von einer direkten Übertragung per SPI ist allerdings abzuraten, da dies relativ kompliziert ist und trotzdem noch viele Kabel und Pins verbraucht. Für SD-Karten gibt es eine SD-Library, auf die ich später noch zurückkommen werde. Interessant zu wissen ist allerdings,

dass der Controller auf dem Arduino-Board seinen Bootloader normalerweise über diese Schnittstelle zugeschickt bekommt.

Serial (UART) und USB

Es gibt einen großen Unterschied zwischen Serial und USB, aber ein Arduino behandelt wie aus Hardware-technischen Gründen beinahe gleich. Jedes Arduino verfügt auch über mindestens einen seriellen Port, ein Mega sogar über vier. Dieser eine ist aber meist durch einen Adapter belegt, welcher die seriellen Daten zu einem USB-Signal wandelt. Daher wird im Programm auch für die USB-Kommunikation immer ein `Serial`-Befehl verwendet. Wie man die anderen seriellen Schnittstellen verwendet, kann man auf der Arduino-Website nachlesen.

Am wichtigsten ist aber trotzdem die Möglichkeit der Kommunikation mit einem Computer. Dazu muss nur im Setup die Übertragungsgeschwindigkeit eingestellt werden, meist nimmt man 9600 Baud.

```
void setup()
{
      Serial.begin(9600);     // Geschwindigkeit einstellen
}
Void loop()
{
      Serial.print(“Eine Zahl: ”); // Text senden, keine neue Zeile
      Serial.println(123);         // Zahl senden, neue Zeile beginnen
}
```

Zeitgesteuerte Abläufe

Häufig kommt in die Situation, dass man eine Zeit abwarten muss, aber gleichzeitig etwas anderes tun will. Jetzt reicht ein `delay()` nicht mehr aus, da `delay()` das gesamte Programm pausiert (währenddessen sind keine anderen Funktionen verfügbar).

Doch zum Glück gibt es eine Möglichkeit, dies zu umgehen: Man liest die Laufzeit (seit der Controller Strom bekommen hat / seit dem letzten RESET) aus, speichert diese in einer Variable und vergleicht diese mit der Aktuellen. `millis()` gibt die Zeit in Millisekunden zurück, für genauere Steuerung kann auch `micros()` (Zeit in Mikrosekunden) verwendet werden. Sobald die aktuelle Zeit um x größer ist als die Gespeicherte, kann man etwas tun, zum Beispiel mit einer Messung aufhören oder ein besonderes Ereignis dazwischenschieben.

Beispiel auf der nächsten Seite

```
unsigned long prevTime = 0; // Zeitpunkt des vorherigen Ereignisses
void loop()
{
      int time = millis();
      if(time - prevTime >= 10000) // Zeitdifferenz überprüfen
      {
            Serial.println(„Zehn Sekunden vergangen!“)
            prevTime = time;
      }
}
```

Eine andere Möglichkeit, so etwas zu realisieren, sind Interrupts. Auch hierzu sind auf der Arduino-Website Informationen und Beispiele zu finden.

Funktionen

Funktionen sind Programmteile, die mit einem Schlüsselwort immer wieder aufgerufen werden können. Dies ist sinnvoll, wenn man zum Beispiel sehr häufig den Wert eines Sensors einlesen muss, was aber viel Code braucht. Funktionen müssen deklariert werden, bevor sie (in `setup()` oder `loop()`) verwendet werden können. Um die Übersichtlichkeit zu erhalten, greift man meist auf eine Vorwärtsdeklaration zurück: Man schreibt nur den Funktionskopf an den Anfang des Programms (gefolgt von einem Semikolon), die eigentliche Funktion steht aber am Ende des Programms. So sagt man dem Compiler, dass die Funktion noch nachgereicht wird und das ihm noch unbekannte Schlüsselwort keinen Fehler auslösen soll.

```
void ledFlash();  // Vorwärtsdeklaration

void setup()
{
      // Setup
      pinMode(13, OUTPUT);
}

void loop()
{
      ledFlash();       // Führe die Funktion aus
      delay(1000);      // Mach was anderes
      ledFlash();       // Führe die Funktion noch mal aus
      delay(5000);      // Mach nochmal was anderes
      ledFlash();       // Führe die Funktion ein letztes Mal aus
      delay(800);       // Mach ein letztes Mal was anderes
}
```

```
void ledFlash()
{
      digitalWrite(13, HIGH);
      delay(50);                    // Wird das ganze Programm aufhalten
      digitalWrite(13, Low);
}
```

Funktionen werden nicht parallel zum normalen Programm ausgeführt, sie werden in dem Programmablauf mit eingebunden.

Es ist möglich, einer Funktion Werte zu übergeben:

```
void ledFlash(int pin, int duration)
{
      digitalWrite(pin, HIGH);
      delay(duration);
      digitalWrite(pin, LOW);
}
```

Natürlich muss dann auch die Vorwärtsdeklaration angepasst werden:

```
void ledFlash(int pin, int duration);
```

Die Funktion wird nun so aufgerufen:

```
ledFlash(13, 50);       // Lasse LED 13 für 50 ms aufblitzen
```

Übergibt man einer Funktion Variablen, übergibt man der Funktion nur eine Kopie der Variable bzw. deren Inhalt. Die ursprüngliche Variable wird nicht verändert!

Insbesondere wenn eine Funktion Messen oder Rechnen soll, muss sie dem Hauptprogramm auch ihr Ergebnis mitteilen können. Dazu gibt es Rückgabewerte: wenn man eine Funktion nicht mit `void`, sondern zum Beispiel mit `int` einleitet, muss die Funktion einen Integer zurückgeben. Sobald ein Wert zurückgegeben wurde, wird die Funktion auch beendet und man kehrt zum Hauptprogramm zurück. Auch hier muss die Vorwärtsdeklaration entsprechend angepasst werden.

```
int messen()
{
      int value = analogRead(A0);
      return value;
      // Code, der hier stehen steht, würde nicht mehr ausgeführt werden
}

// So ruft man die Funktion dann auf:
int value = messen();
```

Präprozessordirektiven

Präprozessordirektiven sind Befehle, welche nicht vom Compiler übersetzt werden. Das bedeutet, dass sie vor dem eigentlichen kompilieren verarbeitet werden. Es gibt zwei wichtige Präprozessordirektiven:

```
#include <Datei.h>
```

Lädt eine Bibliothek in Form einer Header-Datei. Der Quellcode aus dieser Datei wird anstatt der Codezeile eingefügt.

```
#define BEISPIEL 5
```

Jedes Mal, wenn im Programm das Wort „BEISPIEL“ vorkommt, wird es durch „5“ ersetzt. Das ist nützlich, wenn man bestimmte Werte häufig verwendet, aber sie nur einmal festlegt und so schnell ändern kann (z. B. Pins). Natürlich kann man stattdessen auch symbolische Konstanten verwenden (siehe Variablen), aber da Präprozessordirektiven nicht übersetzt und hochgeladen werden, sondern den Code selbst verändern, belegt man so keinen unnötigen Speicher auf dem Controller.