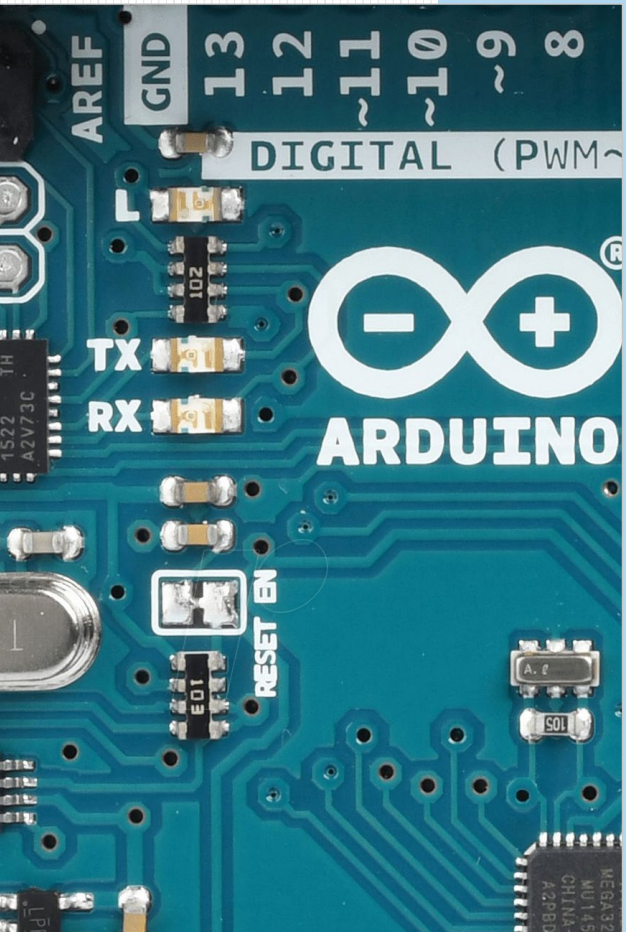
2 Die Programmierung des Arduino

Die als »Sketche« bezeichneten Programme für den Arduino werden in der Arduino-Entwicklungsumgebung (IDE) in einer C/C++-ähnlichen, auf Wiring basierenden Programmiersprache entwickelt. Wir führen kurz in die Besonderheiten dieser Programmierung ein, bevor wir mit der Konstruktion und der Steuerung der ersten Modelle beginnen.



2.1 Grundsätzliches

Die in Java programmierte und aus der am MIT entwickelten Programmierumgebung *Processing* hervorgegangene Entwicklungsumgebung (IDE) des Arduino wird als Teil des Open-Source-Projekts kontinuierlich weiterentwickelt und auf der Arduino-Webseite[1] für die Betriebssysteme Windows, Linux, Linux ARM und Mac OS X zum kostenlosen Download angeboten. Sie umfasst einen Programmeditor und einen auf dem Gnu-C-Compiler basierenden C/C++-Compiler.[2] In dieser IDE wird der Arduino in einer der Programmiersprache C/C++ sehr ähnlichen, aus *Wiring* abgeleiteten Sprache programmiert.[3]

Alle in diesem Buch vorgestellten Modelle können mit den abgedruckten und von der Webseite zum Buch herunterladbaren Sketchen gesteuert werden. Daher ist ein tieferes Verständnis der Programmiersprache für den Nachbau der Modelle nicht erforderlich.

Wer aber die Sketche erweitern oder variieren möchte, sollte einige grundlegende Dinge über die Programmierung des Arduino wissen. Daher möchten wir in diesem Kapitel auf einige Besonderheiten des Arduino etwas näher eingehen, die auf die Tatsache zurückzuführen sind, dass wir es beim Arduino mit einem *Embedded System* mit zahlreichen I/O-Schnittstellen, aber ohne Monitor und Tastatur zu tun haben.

Daraus ergeben sich ein paar spezielle Rahmenbedingungen, die bei der Programmierung des Arduino zu beachten sind. Dazu zählen insbesondere die folgenden fünf Punkte:

- Die Programme (*Sketche*) werden in der Arduino-Entwicklungsumgebung (IDE, Abb. 2–1) auf einem PC oder Laptop entwickelt. Mit einem Mausklick auf »Hochladen« werden sie in einem Schritt kompiliert und, wenn das fehlerfrei gelingt, über eine USB-Verbindung direkt in den Flash-Speicher des Arduino geladen und dort ausgeführt. Dabei werden zuvor auf den Arduino geladene Programme überschrieben. Im Arduino-Speicher ist daher immer genau ein Programm aktiv, das auch nach dem Ausschalten des Arduino erhalten bleibt.

---

1 Download der aktuellen Arduino-IDE: *https://www.arduino.cc/en/Main/Software*

2 Auf Details der Arduino-IDE (Installation, Konfiguration, Bedienung) gehen wir hier nicht weiter ein, denn für versierte Programmierer ist sie weitgehend selbsterklärend und für Einsteiger gibt es bereits hervorragende Bücher und Einführungen. Für einen schnellen Einstieg ist z. B. das »Arduino Special« des Make-Magazins sehr zu empfehlen [7].

3 Auch auf eine Einführung in C/C++ verzichten wir hier: Vielen Lesern wird C bekannt sein, und für alle anderen gibt es im Internet und natürlich auch in Buchform zahlreiche sehr gute Einführungen, von denen wir eine Auswahl auf der Webseite zu diesem Buch verlinkt haben. Speziell für die Programmierung von C auf dem Arduino sind als freie Quellen [9] und [10] zu empfehlen.

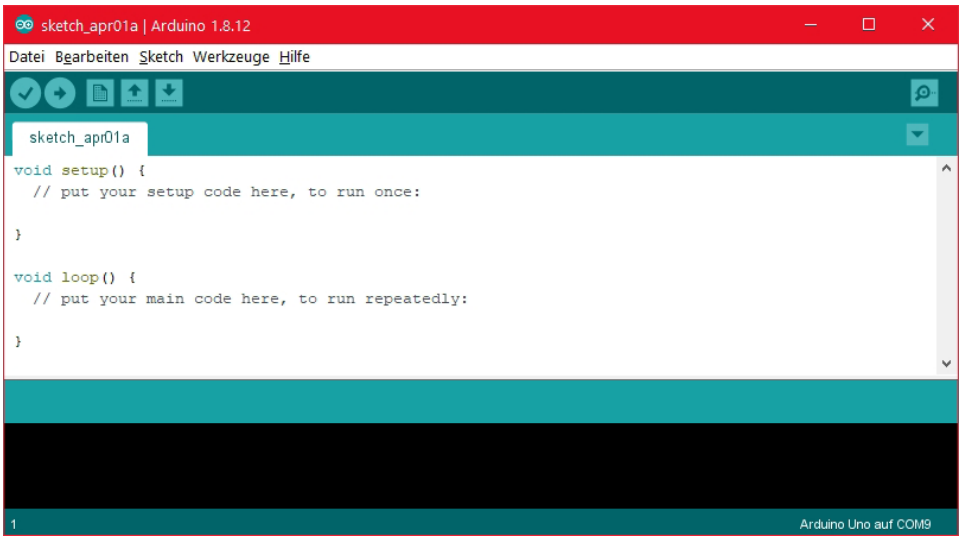


Abb. 2–1 Arduino-IDE mit Default-Sketch

- Ein etwa 0,5 kB großer *Bootloader* im Flash-Speicher des Arduino sorgt dafür, dass das in den Speicher geladene Programm unmittelbar nach der Aktivierung (Anschluss der Stromversorgung oder des USB-Kabels bzw. Abschluss des Uploads) oder nach dem Drücken des Reset-Tasters automatisch auf dem Arduino-Board ausgeführt wird. Will man ein sofortiges Starten (z. B. bei einem autonomen Roboter) verhindern, muss man Roboter und Programm um einen Starttaster erweitern.
- Ein Arduino-Programm kann jederzeit durch das Drücken des Reset-Tasters (rechts oben in Abb. 1–2) auf dem Arduino-Board zum Neustart veranlasst, aber nur durch Unterbrechung der Stromversorgung beendet werden.
- Der sehr begrenzte Hauptspeicher (beim Arduino Uno lediglich 31,5 kByte Flash-Speicher für das Programm und 2 kByte RAM Variablenspeicher) zwingt dazu, mit dem Speicherplatz zu »haushalten«. Dabei hilft der Compiler, der die Speicherbelegung minimiert. Die IDE zeigt nach dem Kompilieren eines Sketches den von Programm und Variablen belegten Speicherplatz an.
- Jedes Arduino-Programm besteht mindestens aus einer `setup()`-Routine, die einmalig zu Programmbeginn ausgeführt wird, und dem Hauptprogramm `loop()`, das anschließend so lange wiederholt wird, bis die Stromversorgung unterbrochen wird (Listing 2–1).

```
void setup() {
  // Initialisierungen (wird zu Programmbeginn einmal durchlaufen)
}

void loop() {
  // Hauptprogramm (wird ad infinitum wiederholt)
}
```

Listing 2–1 Genereller Aufbau eines Arduino-Sketches

Ein Arduino-Programm entspricht in Syntax, Programmstruktur, verwendeten Variablentypen und Operatoren im Wesentlichen der C/C++-Spezifikation, die wir hier als bekannt voraussetzen. Einige wichtige, Arduino-spezifische Eigenschaften und häufig genutzte Befehlsgruppen stellen wir in den folgenden Abschnitten vor.

2.2 Bibliotheken

Für den Arduino gibt es unzählige Codebibliotheken, die (wie in C-Compilern üblich) mit dem Compiler-Kommando `#include` eingebunden werden können und dann nach dem Kompilieren hinzugelinkt werden. Die IDE wird bereits mit einigen Standardbibliotheken ausgeliefert; andere müssen aus dem Internet heruntergeladen und über die Bibliotheksverwaltung als Zip-File in die IDE importiert werden.

```
#include <*.h> // Einbinden einer Bibliothek
```

Bei jedem Start prüft die IDE, ob neue Versionen der installierten Bibliotheken verfügbar sind; das Update kann dann per Mausklick aktiviert werden.

Rund 40 Funktionsbibliotheken finden sich auf Github im »offiziellen« Arduino-Bereich[4]. Die Suche nach »arduino« liefert allein auf Github insgesamt weitere etwa 22.000 Arduino-Code-Repositories. Zu praktisch jedem Sensor oder Aktor, der am Arduino genutzt werden kann, wird inzwischen vom Hersteller eine Arduino-Treiberbibliothek bereitgestellt, oft auf Github, manchmal auch zum Download auf der Webseite des jeweiligen Herstellers. An hilfreichem Programmcode herrscht also kein Mangel. Allein die Auswahl der richtigen – aktuellen, gepflegten und ausgetesteten – Bibliothek oder eines passenden Beispielprogramms für eine bestimmte Anwendung ist manchmal herausfordernd.

Mit der Auswahl unserer Modelle und ihrer Funktionen haben wir uns auf einige zentrale Bibliotheken festgelegt und dabei versucht, deren Anzahl und

4 Quelle für Arduino-Bibliotheken: *https://github.com/arduino-libraries*

Komplexität zu begrenzen. Die folgenden für die Anwendungen und Sketche in diesem Buch wichtigen grundlegenden Arduino-Bibliotheken erläutern wir in den nachfolgenden Abschnitten:

- `SPI.h`: Kommunikation über den SPI-Bus (*Serial Peripheral Interface*)
- `Wire.h`: Kommunikation über das *Two Wire Interface* (TWI, I²C)
- `Servo.h`: Steuerung eines Servomotors
- `Adafruit_MotorShield.h`: Ansteuerung von Gleichstrom- und Schrittmotoren

Außerdem verwenden wir für einzelne Modellvarianten vier weitere externe Bibliotheken, in deren Nutzung wir in den jeweiligen Kapiteln einführen:

- `ArduinoNunchuk.h`: (Fern-)Steuerung über einen Wii-Nunchuk
- `Pixy.h`: Live-Bildauswertung mit der Pixy-Kamera
- `SD.h`: Lesezugriff auf eine SD-Karte (via SD-Leser)
- `Mouse.h`: Maussteuerung des PCs über die USB-Schnittstelle

2.3 Serieller Monitor

Die IDE des Arduino kennt keine Breakpoints oder andere hilfreiche Funktionen zur komfortablen Fehlersuche (*Debugging*). Da der Arduino ohne ergänzende Hardware auch über keine Ausgabeeinheit wie einen Bildschirm verfügt, lassen sich Inputwerte oder Inhalte von Variablen nicht vom Arduino anzeigen.

Um dennoch eine Überwachung und sogar eine manuelle Steuerung des Programmablaufs zu ermöglichen, haben die Entwickler der Arduino-IDE ein Monitorfenster spendiert, an das der Arduino mit einem einfachen seriellen Protokoll über die USB-Verbindung und eine UART-Schnittstelle Daten übertragen kann (Abb. 2–2). Die Übertragungsgeschwindigkeit kann dabei von 300 bis 2.000.000 Baud gewählt werden. In diesem Buch verwenden wir einheitlich eine Baudrate von 115.200 Bit/s; so muss beim Laden eines anderen Sketches die Rate nicht jedes Mal am seriellen Monitor der IDE angepasst werden.

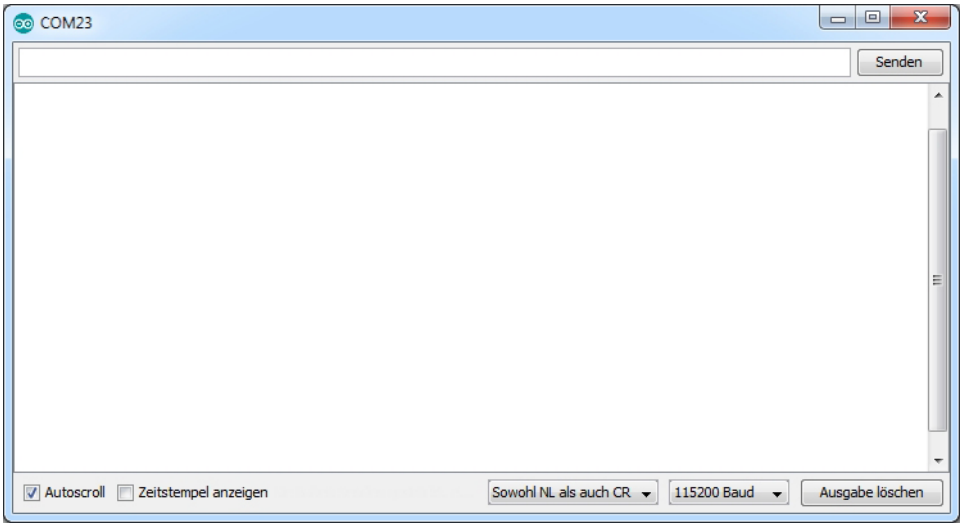


Abb. 2–2 Serieller Monitor der Arduino-IDE

Die empfangenen Daten werden im Monitorfenster der IDE angezeigt (öffnen über das Icon rechts oben in der IDE, Abb. 2–1). Auf demselben Weg können auch Tastatureingaben über das Eingabefenster des Monitors an den Arduino übermittelt werden.

Die Befehle zur Ansteuerung des seriellen Monitors sind integriert und müssen nicht über eine zusätzliche Bibliothek eingebunden werden. Der serielle Monitor muss lediglich mit Angabe der Übertragungsgeschwindigkeit der Daten (Baudrate in Bit/s) aktiviert werden:

```
void Serial.begin(long baud);
```

Dabei kann als Baudrate `baud` einer der folgenden Werte gewählt werden: 300, 1.200, 2.400, 4.800, 9.600, 19.200, 38.400, 57.600, 74.880, 115.200, 230.400, 250.000, 500.000, 1.000.000 oder 2.000.000.[5] Sie muss mit der im Fenster des seriellen Monitors unten rechts ausgewählten Baudrate übereinstimmen, damit die Zeichen korrekt empfangen und dargestellt werden. In den Sketchen in diesem Buch verwenden wir, wie erwähnt, einheitlich die Baudrate 115.200.

Das serielle Protokoll des Monitors arbeitet mit acht Datenbits, keinem Paritäts- und einem Stoppbit. Mit dem erweiterten Kommando

```
void Serial.begin(long baud, int config);
```

5 Grundsätzlich sind auch andere Baudraten möglich; für den seriellen Monitor können in der IDE jedoch nur diese ausgewählt werden.

können auch andere Protokollvarianten gewählt werden, wie z. B. sechs Datenbits, ein Bit für gerade Parität und zwei Stoppbits. Der serielle Monitor der IDE erwartet jedoch die Standardeinstellung.

Die Aktivierung des seriellen Monitors, durch die die Anschlüsse an den Pins D0 und D1 als Empfangs- (RX) und Sendeleitung (TX) belegt werden und daher im Sketch nicht für andere Zwecke genutzt werden dürfen, erfolgt üblicherweise in der `setup()`-Routine. Datenübertragungen werden von den LEDs RX und TX auf dem Arduino-Board angezeigt.

Will man den seriellen Monitor nur temporär aktivieren, lässt er sich mit der folgenden Funktion auch wieder deaktivieren; damit werden zugleich die Pins D0 und D1 für die Nutzung als digitale I/O-Pins freigegeben:

```
void Serial.end();
```

Ausgabe

Die Ausgabe von Daten (eines numerischen Wertes oder eines Textstrings) auf dem seriellen Monitor erfolgt mit folgenden Kommandos:

```
long Serial.print(string text);
long Serial.print(int var, byte type);
long Serial.print(float var, int len);
```

Dabei gibt `type` die Darstellung der Zahl an: `DEC`, `HEX`, `OCT` oder `BIN`. Bei Fließkommazahlen wird mit dem zweiten Parameter `len` die Anzahl der anzuzeigenden Nachkommastellen festgelegt; fehlt die Angabe, werden als voreingestellter Defaultwert zwei Nachkommastellen angezeigt. Die Funktion liefert die Anzahl der geschriebenen Zeichen zurück.

Der folgende Befehl schließt die Ausgabe außerdem mit einem Zeilenumbruch ab:

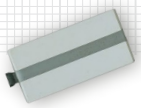
```
long Serial.println(string text);
```

Eingabe

Auch die Übergabe von Zeichen oder Daten an ein Arduino-Programm ist mit dem seriellen Monitor möglich. Dazu werden Zahlen oder eine Zeichenfolge in das Eingabefeld des Monitorfensters eingetragen und abschließend der »Senden«-Knopf (rechts oben) gedrückt.

Der Arduino kann eine Eingabe mit der folgenden Funktion feststellen:

```
int Serial.available();
```



Die Funktion gibt die Zahl der im 64 Byte großen Eingangspuffer vorliegenden Zeichen (`char`) zurück. Aus diesem Puffer können die Daten mit

```
int Serial.read();
```

jeweils einzeln als ASCII-Zeichen ausgelesen werden. Liegt kein Zeichen vor, liefert die Funktion den Wert -1 zurück. Ganzzahlige, durch Leerzeichen getrennte Werte und Fließkommazahlen können alternativ mit den folgenden Funktionen aus dem Puffer gelesen werden:

```
int Serial.parseInt();
float Serial.parseFloat();
```

Dabei wird der jeweils nächste Integerwert bzw. die nächste Fließkommazahl aus dem seriellen Puffer ausgewertet; andere Zeichen (Buchstaben, Leerzeichen) werden übersprungen.

Über einen speziellen Event-Handler lässt sich die Bearbeitung von Eingaben im seriellen Monitor sogar aus dem Hauptprogramm ausgliedern:

```
void serialEvent() {
  // Behandlung von Eingaben über den seriellen Monitor
}
```

Der Event-Handler wird automatisch aufgerufen, wenn eine Eingabe vorliegt; er ersetzt das kontinuierliche Abfragen von `Serial.available()`.

Debugging

Die Nutzung des seriellen Monitors vereinfacht in vielen Fällen die Fehlersuche: Variableninhalte und Sensorwerte können ausgegeben und durchlaufene Programmabschnitte angezeigt werden.

Allerdings kosten die Befehle Laufzeit und knappen Programmspeicher. Wird der serielle Monitor nur für die Fehlersuche benötigt, dann will man – insbesondere bei zeitkritischen Abläufen – die Monitorbefehle aus der endgültigen Version des Programms entfernen.

Das gelingt leicht, wenn man die Funktionsaufrufe mit Compiler-Bedingungen versieht – damit spart man sich später ein manuelles Auskommentieren oder Löschen der Monitorbefehle. Beim folgenden Codebeispiel erfolgt die jeweilige Ausgabe nur, wenn `DEBUG` definiert ist. Wird die einleitende Definition `#define DEBUG` auskommentiert, sind die Aufrufe des seriellen Monitors im kompilierten Programmcode nicht mehr enthalten.

```
#define DEBUG

#ifdef DEBUG
  #define Baud 115200
#endif

void setup() {
#ifdef DEBUG
  Serial.begin(Baud);
#endif
  // setup-Kommandos
}

void loop() {
#ifdef DEBUG
  Serial.println("Debugging");
#endif
  // Programmcode
}
```

Listing 2–2 Compiler-Bedingungen für Debugging-Kommandos

Die Sketche des Buches enthalten diese Codefolge immer dann, wenn der Sketch zeitkritische Programmzweige enthält, die durch die Kontrollausgaben auf dem seriellen Monitor spürbar verlangsamt werden.

2.4 I/O-Ports

Digitale Ports

Die 14 von 0-13 durchnummerierten I/O-Pins (oben in Abb. 1–1) sind digitale Ein- oder Ausgänge, die wir im Folgenden mit D0 bis D13 bezeichnen. Sie werden via Software als Eingang oder Ausgang konfiguriert. Die »Betriebsart« eines digitalen I/O-Pins wird – üblicherweise in der `setup()`-Routine – mit folgendem Befehl festgelegt:

```
void pinMode(byte pin, byte mode);
```

`pin` gibt die Nummer (0-13) des Digitalpins an. Als Betriebsart (`mode`) können `INPUT`, `OUTPUT` oder `INPUT_PULLUP` gewählt werden. `INPUT` ist beim Start des Arduino voreingestellt, daher kann man in der `setup()`-Routine eigentlich auf die Einstellung des `INPUT`-Modus verzichten. Meist macht man es dennoch, damit beim

Blick in den Sketch sofort klar ist, welche Pins zu welchen Zwecken verwendet werden.

Im `INPUT`-Modus liefert der Port entweder `HIGH` (hohes Potenzial, 5 V) oder `LOW` (niedriges Potenzial, 0 V):

```
int digitalRead(byte pin);
```

Wird ein digitaler Pin im `OUTPUT`-Modus verwendet, können mit dem folgenden Befehl ein `HIGH`- oder ein `LOW`-Signal (5 V/ 0 V) ausgegeben werden:

```
void digitalWrite(byte pin, byte value);
```

Im `INPUT_PULLUP`-Modus wird der Eingang intern über einen 20-kΩ-Widerstand mit 5 V verbunden, also auf 5 V »gezogen« – daher auch die Bezeichnung *Pull-up*-Widerstand. Ein angeschlossener Taster liefert dann im offenen Zustand ein definiertes `HIGH`-Signal – anderenfalls wäre der Wert undefiniert und Störströme können das anliegende Signal beeinflussen. Wird der mit GND verbundene Taster geschlossen, liegt ein definiertes `LOW`-Signal an.

Alternativ kann auch ein externer Pull-up-Widerstand verwendet werden. Durch die Nutzung des `INPUT_PULLUP`-Modus spart man sich jedoch in der Schaltung einen Widerstand und den 5-V-Anschluss. Da der Strom bei offenem Input über den Pull-up-Widerstand nach GND abfließt, hat unsere Schaltung unvermeidlich eine Verlustleistung. Damit diese möglichst gering ausfällt, wird ein großer Widerstand als Pull-up verwendet.

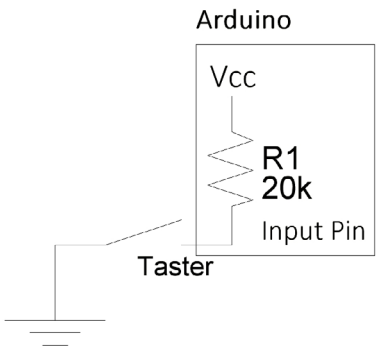


Abb. 2–3 *Interner 20-kΩ-Pull-up-Widerstand R1 für ein definiertes Signal bei offenem Input (z. B. einem geöffneten Taster)*

Die Verlustleistung des internen Pull-up-Widerstands des Arduino liegt damit bei:

$$P = U \cdot I = \frac{U^2}{R} = \frac{25V^2}{20.000\Omega} = 1{,}25\text{mW}$$

Die Funktion `digitalWrite()` ermöglicht außerdem, im `INPUT`-Modus den internen 20-kΩ-*Pull-up*-Widerstand ein- (`HIGH`) und auszuschalten (`LOW`). Das entspricht einer Umschaltung des Pin-Modus von `INPUT` auf `INPUT_PULLUP` (bzw. umgekehrt).

Einigen der digitalen Pins sind spezielle Funktionen zugeordnet. Werden diese Funktionen genutzt, stehen die entsprechenden Pins für andere Aufgaben nicht zur Verfügung:

- Die Pins D0 und D1 sind für die RX/TX-Signale zum seriellen Monitor zuständig und belegt, wenn der serielle Monitor aktiviert ist, um Ausgaben auf ihm oder Eingaben über ihn vorzunehmen (siehe Abschnitt 2.3).

- Die Pins D2 und D3 können mit externen Interrupts belegt werden (siehe Abschnitt 2.5).
- Servomotoren benötigen die PWM-Pins D10 (Servo1) bzw. D9 (Servo2).
- Die Pins D10-D13 sind mit den Signalleitungen des seriellen SPI-Protokolls belegt: SS, MOSI, MISO und SCK (siehe Abschnitt 2.7).
- Schließlich ist Pin D13 direkt mit einer On-Board-LED verbunden, die über das Anlegen eines `HIGH`- oder `LOW`-Signals angesteuert werden kann.

Pin	Funktion	PWM
D0	RX	
D1	TX	
D2	INT0	
D3	INT1	~ (Timer2)
D4		
D5		~ (Timer0)
D6		~ (Timer0)
D7		
D8		
D9	Servo2	~ (Timer1)
D10	Servo1, SS (SPI)	~ (Timer1)
D11	MOSI (SPI)	~ (Timer2)
D12	MISO (SPI)	
D13	SCK (SPI)	

Tab. 2–1 Spezielle Funktionen der digitalen Pins D0 bis D13 (Arduino Uno)

Analoger Output

Der ATmega328P verfügt über keinen D/A-Wandler. Dennoch kann über die Digitalpins mit dem folgenden Befehl auch ein analoger 8-Bit-Wert (0-255) ausgegeben werden:

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

Dabei wird der Integerwert in ein *Pulse-Width-Modulation*-Signal (Pulsweitenmodulation, PWM) umgewandelt: Der Ausgang sendet ein »gepulstes« Signal