Mikrocomputertechnik I

Übungen zur Vorlesung

Ausgabe 0.1, 09.09.2013
Autor: Stephan Rupp

Inhaltsverzeichnis

1. Einführung

Die Vorlesung Mikrocomputertechnik ist wie folgt aufgebaut: Teil 1 (vorliegendes Manuskript) befasst sich mit der Programmierung eines Mikrocomputers mit Hilfe einer Hochsprache. Die Vorlesung wird durch praktische Übungen auf dem Arduino Mikrocomputer unterstützt. Ziel dieses Teils ist es, die Teilnehmer in die Lage zu versetzen, eigenständig Programme für Mikrocomputer zu erstellen, Schaltungen zu realisieren und eigenständig tiefer in die Materie vorzudringen. Für diesen Zweck genügt eine einfache Abstraktion der Funktionsweise und des Innenlebens eines Mikrocomputers.

Teil 2 der Vorlesung befasst sich dann genauer mit dem Aufbau vom Mikrocomputern unterschiedlicher Bauart, sowie mit der maschinennahen Programmierung in C und Assembler. Teil 2 der Vorlesung wird durch praktische Übungen unterstützt, bei denen Schaltungen mit dem Mikrocomputer aufgebaut werden, mit der nativen Programmierumgebung des Mikrocomputers programmiert und in Betrieb genommen werden. Hierzu gibt es Bausätze der DHBW mit passender Leiterplatte und den benötigten Bauteilen. Für Teil 2 wird ein eigenes Manuskript herausgegeben, sowie unabhängig vom Vorlesungsmanuskript ein Handbuch mit Anleitungen für die praktischen Übungen.

2. Aufbau eines Mikrocomputers

2.1. Rechnerarchitektur

Grundsätzlich sind alle Rechner nach dem gleichen Prinzip aufgebaut. Man unterscheidet den eigentlichen Prozessor, den Speicher, sowie Schnittstellen für externe Geräte, wie zum Beispiel Tastatur, Maus und Bildschirm. Folgende Abbildung zeigt eine Übersicht.

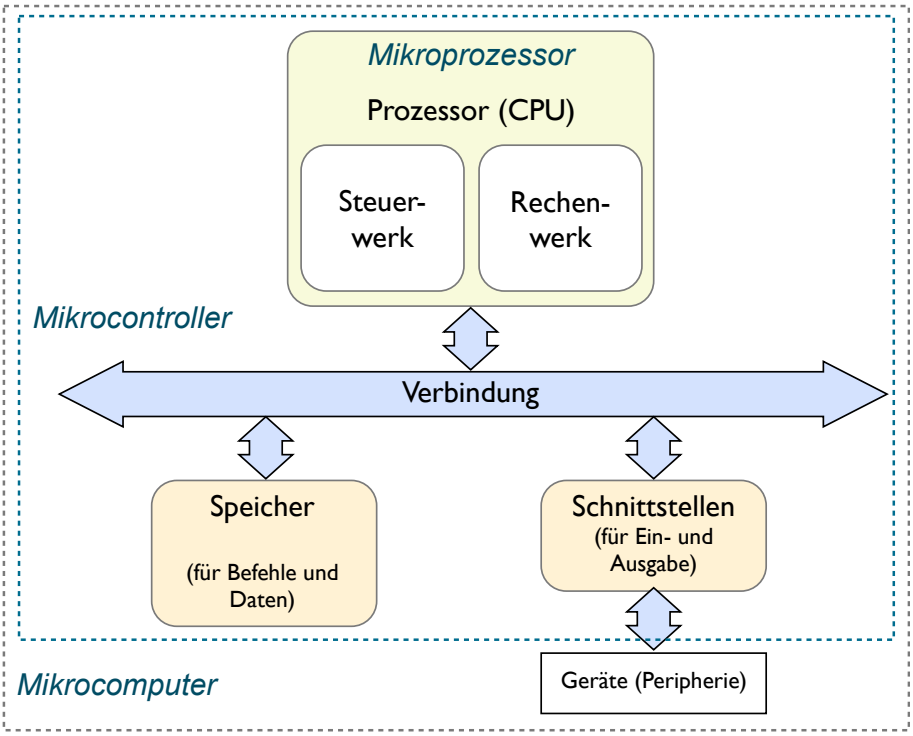


Bild 2.1 Grundsätzlicher Aufbau eines Rechners

Kern des Rechners ist die zentrale Recheneinheit (engl. CPU für Central Processing Unit): der Prozessor oder Mikroprozessor. Der Prozessor enthält eine Recheneinheit und eine Steuereinheit. Mit Hilfe der Steuereinheit nimmt er den nächsten Befehl aus dem Speicher und führt den Befehl aus. Die zur Ausführung benötigten Daten entnimmt er ebenfalls dem Speicher. Umgekehrt schreibt er Ergebnisse mit Hilfe des Steuerwerkes zurück in den Speicher. Das Steuerwerk ist auch für die Ausführung von Sprüngen im Ablauf des Programms verantwortlich, z.B. wenn ein Unterprogramm ausgeführt wird bzw. wenn ein Programm zugunsten einer dringenderen Aufgabe unterbrochen wird.

Das Rechenwerk dient der Berechnung von Ergebnissen, d.h. arithmetischen und logischen Operationen. Um den im Programm vorgegebenen Ablauf abzuarbeiten, benötigt der Prozessor Anweisungen (Befehle), sowie Daten. Sowohl Befehle und Daten sind im Speicher an definierter Stelle abgelegt.

Wie kann der Prozessor aber zu einer Benutzereingabe auffordern, bzw. einen Wert an ein angeschlossenes Anzeigeinstrument schicken? Hierzu wird die dritte in der Abbildung gezeigte Komponente benötigt: Schnittstellenbausteine für externe Geräte. Die externen Geräte adressiert der Prozessor in einem eigenen Adressbereich innerhalb seines Adressraums.

An dieser Stelle seien noch einige Begriffe geklärt: Unter einem Prozessor bzw. Mikroprozessor wird die zentrale Recheneinheit (CPU) verstanden. Hier geht es um den reinen Prozessor mit Steuerwerk und Rechenwerk. Bei einem Mikrocontroller handelt es sich um einen Baustein, auf dem sich zusätzlich zum Prozessor Speicher befindet, sowie einige Schnittstellen für Geräte.

Klapprechner und Rechner für den Schreibtisch verwenden reine Mikroprozessoren. Speicher und Bausteine für Schnittstellen befinden sich auf der Leiterplatte des Rechners (engl. Motherboard). Der Grund für die unterschiedliche Implementierung der Architektur liegt in den unterschiedlichen Anwendungen. Ein Laptop oder Rechner am Arbeitsplatz lädt seine Daten beim Start von einer Festplatte. Er benötigt Flexibilität bzgl. seiner der Ausstattung an Arbeitsspeicher und Prozessorleistung.

Mikrocontroller werden vorwiegend für eingebettete Systeme eingesetzt. Als nicht flüchtiger Speicher sind hier keine Festplatten vorgesehen. Daher werden Programmspeicher (als nicht flüchtiger Speicher) und zumindest ein Teil des Arbeitsspeichers direkt auf dem Baustein untergebracht. Ebenso werden die wichtigsten Schnittstellen direkt auf dem Baustein implementiert. Diese Implementierung ist kompakt und kosteneffizient. Die Architektur bleibt jedoch grundsätzlich gleich. Unter einem Computer (Rechner) bzw. Mikrocomputer wird schliesslich das gesamte System verstanden.

2.2. Ein- und Ausgabesystem

Wie funktioniert nun die Kommunikation des Prozessors mit internen und externen Geräten etwas genauer? Die folgende Abbildung zeigt die Anordnung aus der Perspektive der Hardware mit einigen zusätzlichen Details.

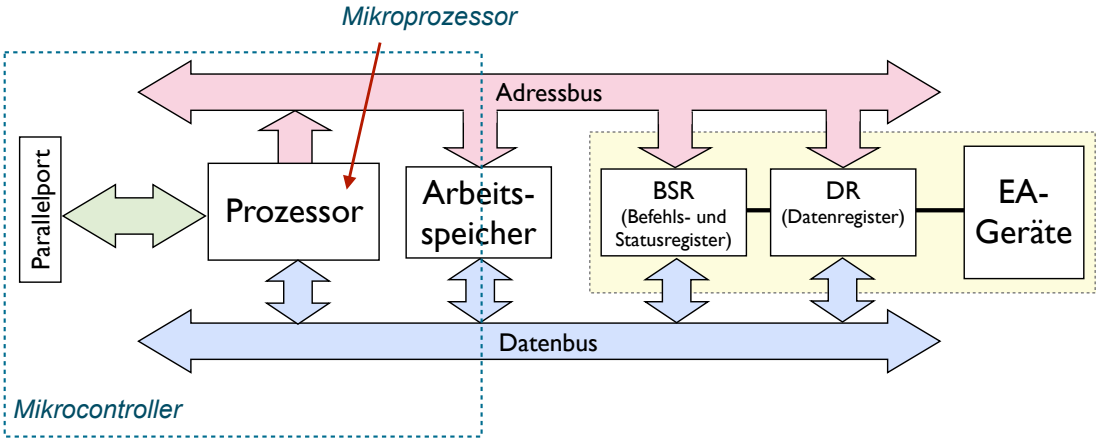


Bild 2.2 Mikrocomputer

Der Prozessor verfügt über einen Adressbus, den er verwendet, um Bereiche im Arbeitsspeicher anzusprechen. Auf diese Weise greift er auf den Ort des nächsten Befehls zu, bzw. auf den Ort zum Laden oder Speichern von Daten. Der angesprochene Speicher stellt den benötigen Befehl oder die benötigten Daten über den Datenbus bereit, bzw. liest die zu speichernden Daten vom Datenbus.

Zur Verbindung mit der Aussenwelt dienen Schnittstellenbausteine, die aus Sicht des Prozessors wie ein Bereich im Speicher verwendet werden. Die Geräte sind in den Adressraum des Prozessors eingebunden. Auf den Schnittstellenbausteinen findet sich ein Register für Instruktionen an das angeschlossene Gerät, bzw. zur Abfrage von Statusmeldungen des angeschlossenen Gerätes (Befehls- und Statusregister). Außerdem findet sich auf dem Schnittstellenbaustein ein Datenregister, das gelesen bzw. beschrieben werden kann. Diese Art der Anbindung überwiegt bei leistungsfähigeren Rechnern. Hier können die Schnittstellenbausteine auch eigenständig direkte Datentransfers in der Arbeitsspeicher beim Prozessor anmelden (engl. DMA, Direct Memory Access)

Bei Mikrocontrollern findet sich eine Besonderheit, die links in der Abbildung dargestellt ist: parallele Ports. Diese Ports sind in der Regel digitale Eingänge bzw. digitale Ausgänge, über die einfache Peripheriegeräte direkt angeschlossen werden können. In diese Kategorie von Geräten fallen Taster und Tastaturen, Relais bzw. Anzeigen (LEDs, LCD-Displays). Über die digitalen Ports lassen sich jedoch auch serielle Schnittstellenprotokolle fahren (wie z.B. SPI oder I^2C), über die externe Baugruppen wie z.B. Ethernet Schnittstellen, WiFi, oder eine Anbindung an Mobilkommunikationsnetze realisiert werden können.

Ebenso lassen sich A/D-Wandler bzw. D/A Wandler und andere Geräte anschliessen. Solche Geräte profitieren von der seriellen Schnittstelle durch die Ersparnis an andernfalls erforderlichen parallelen Leitungen. Viele Mikrocontroller verfügen auch über analoge Ports, an denen also D/A Wandler für analoge Ausgänge, bzw. A/D Wandler für analoge Eingänge angeschlossen sind.

2.3. Ereignisverwaltung

Ein Ereignis, wie z.B. die Bereitschaft eines externes Geräts zur Fortsetzung des Datentransfers, das Erreichen eines vorgegebenen Wertes im Arbeitsprozess, bzw. die Überschreitung eines Grenzwertes müssen unter Umständen zur Unterbrechung des laufenden Programms führen, damit rasch auf das Ereignis reagiert werden kann. Ebenso sollen planmässige Ereignisse, wie

beispiels-weise ein eingestelltes Zeitintervall (Timer Ereignis), zu einer Unterbrechung der laufenden Routine führen.

Ereignisse können entweder als Nachricht (Alarm bzw. engl. Interrupt) an den Prozessor kommuniziert werden, bzw. den Alarm direkt über eine Signalleitung am Prozessor auslösen. Ergebnis ist die Speicherung des Ereignisses in einem Register des Steuerwerks. Das Steuerwerk unterbricht dann das laufende Programm und springt in die für dieses Ereignis vorgesehene Programm. Damit das funktioniert, muss der Programmierer natürlich an dieser Stelle einen passenden Programmcode hinterlegt haben.

Nach Ausführung der Reaktion auf das Ereignis kehrt der Prozessor wieder in das ursprüngliche Program zurück. In der Regel lassen sich Ereignisse mit unterschiedlichen Prioritäten ausstatten. Behandelt wird zunächst das als am dringendsten eingestufte Ereignis, dann die Ereignisse mit untergeordneter Priorität.

Der in dieser Vorlesung verwendete Mikrocontroller verfügt über zwei Signaleingänge, die sich für Ereignisse aktivieren bzw. deaktivieren lassen. Bei der Aktivierung kann man dem betreffenden Signaleingang ein Unterprogramm zuordnen, das bei Eintreffen des Ereignisses aufgerufen wird (engl. Interrupt-Routine). Das Eintreffen des Ereignisses wird hierbei aus dem Signalzustand der verwendeten Leitung abgeleitet und ist konfigurierbar für bestimmte Zustände bzw. Zustandsübergänge an der Signalleitung. Ebenso besitzt der verwendete Mikrocontroller zwei auf ein vorgegebenes Intervall einstellbare Zeitgeber (Timer).

2.4. Kommunikationsschnittstellen

Beim verwendeten Mikrocomputer (Arduino Uno) stehen eine Vielzahl von Erweiterungen für Kommunikationsschnittstellen zur Verfügung (siehe Web-Seite [6] im Literaturverzeichnis, weiter unter der Schaltfläche Produkte oben). Die meisten der verfügbaren Schnittstellenbaugruppen und Schnittstellenbausteine werden über serielle Schnittstellen (SPI und I^2C) angebunden. Diese Schnittstellen werden im Detail in Abschnitt 8 dieses Manuskripts behandelt.

Zu den vielen Baugruppen und Bausteinen bieten die Hersteller fertige Bibliotheken, die die Programmierung auf hohem Abstraktionsniveau unterstützen. In allen anderen Fällen muss man die Schnittstelle auf Basis der vorhandenen Bibliotheken ab Datenblatt selber programmieren. Auch das ist Teil dieser Vorlesung für die genannten Schnittstellen in Abschnitt 8. Eine weitere wichtige Schnittstelle, die in diesem Abschnitt behandelt wird, ist die Einbindung in das Internet sowohl für lokale Netze als auch für Weitverkehrsnetze.

2.5. Zentrale Recheneinheit

Für die Programmierung auf abstraktem Niveau ist ein Modell der zentralen Recheneinheit und ihrer Arbeitsweise hilfreich. Folgende Abbildung zeigt ein solches vereinfachtes Modell. Dargestellt ist der Mikroprozessor mit seinem Steuerwerk und Rechenwerk. Beide sind angeschlossen an den Adressbus und Datenbus. Das Rechenwerk ist nun etwas detaillierter gestellt. Ein wichtiger Bestandteil sind die Register, in die Daten aus dem Arbeitsspeicher übertragen werden.

Die Zugriffszeiten auf die internen Register sind in aller Regel sehr viel schneller als die Zugriffszeiten auf den Arbeitsspeicher. Daher werden benötigte Daten aus dem Arbeitsspeicher in aller Regel vom Steuerwerk erst in die internen Register übertragen. Das Steuerwerk sorgt anschliessend für die Ausführung von Berechnungen in der Arithmetisch Logischen Einheit des Rechenwerkes (engl. ALU, Arithmetic Logic Unit). Ergebnisse werden dann wieder in den internen Registern gespeichert. Ist

ein Rechenvorgang schliesslich abgeschlossen, transferiert das Steuerwerk die Ergebnisse aus dem Register in den Arbeitsspeicher.

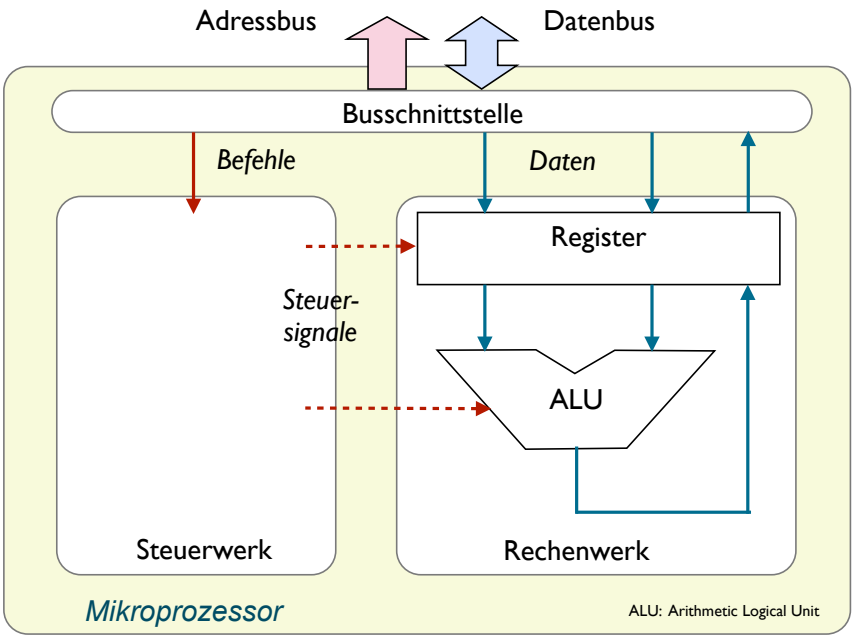


Bild 2.3 Rechenwerk und Steuerwerk

Den Ablauf zur Ausführung eines Befehles kann man insgesamt in zwei Phasen unterteilen: (1) die Phase der Bereitstellung und der Dekodierung des Befehls, (2) die Phase der Ausführung des Befehls. Oben beschrieben wurde die zweite Phase, die Ausführung des Befehls. Zur Ausführung gehören die Bereitstellung der Operanden für das Rechenwerk, die Verarbeitung durch die ALU, sowie die Speicherung des Ergebnisses.

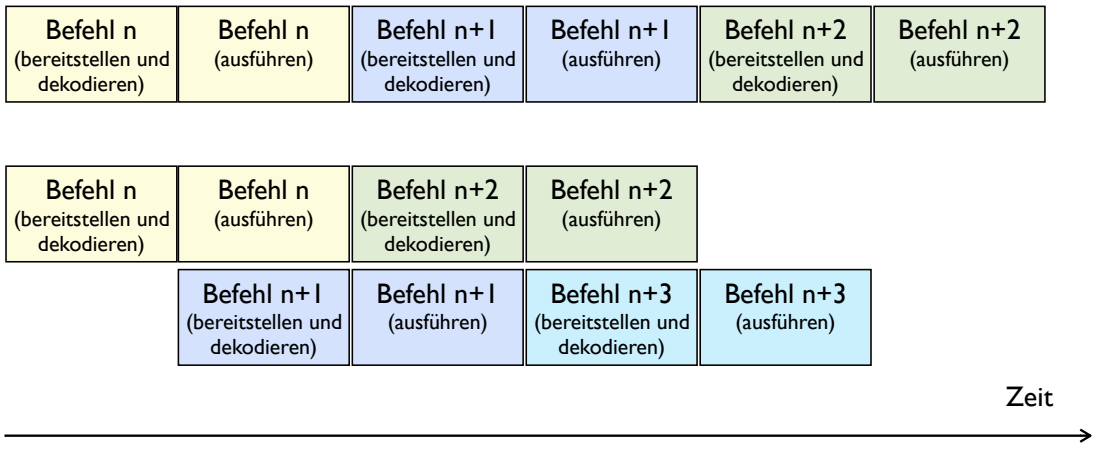


Bild 2.4 Lineare und überlappende Befehlsausführung

Den zeitlichen Ablauf zeigt die Abbildung oben. In dem im oberen Teil der Abbildung dargestellten Ablauf folgen für jeden Befehl abwechselnd die Phasen der Bereitstellung und Dekodierung, sowie der Ausführung des Befehls. Vor der Bereitstellung den nächsten Befehls ist der vorangegangene Befehl ausgeführt. Da jeder Bearbeitungsschritt Taktzyklen des Prozessors

beansprucht, könnte man versuchen, die Effektivität dadurch zu steigern, indem man die Bereitstellung des nächsten Befehle bereits in der Ausführungsphase des laufenden Befehls beginnt, wie unten in der Abbildung gezeigt.

Diese überlappende Verarbeitung wird von zeitgemäßen Prozessoren auch praktiziert und wird unter dem Stichwort „Pipelining“ beschrieben, also Fliessbandverarbeitung. Allerdings ist die Implementierung nicht trivial. Während der lineare Fall ohne Überlappung völlig konfliktfrei abläuft, müssen bei der überlappenden Implementierung Konflikte um Resourcen (gleiche Speicherinhalte) berücksichtigt werden. Ebenso muss der Fall berücksichtigt werden, dass bei einem Sprungbefehl der Ablauf nicht kontinuierlich fortgesetzt wird. Schliesslich sind die Ausführungszeiten nicht für alle Befehle und Rechenoperationen gleich lang.

3. Programmierumgebung

3.1. Von der Hochsprache zur Maschinensprache

Aus dem im vorausgegangenen Abschnitt beschriebenen Modell des Prozessors lässt sich seine Arbeitsweise bei der Ausführung eines Programms unmittelbar ableiten. Betrachtet sei die Ausführung einer arithmetischen Operation, z.B. die Berechnung von c = a + b. Folgende Abbildung zeigt die Ausführung des Programms.

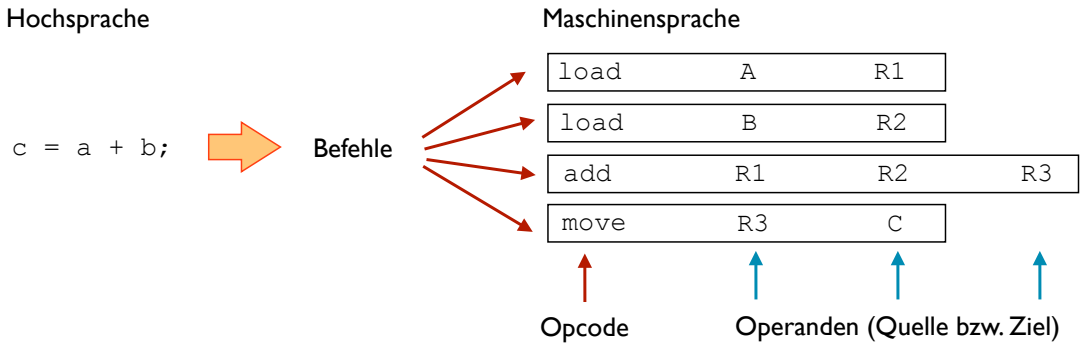


Bild 2.5 Ausführung eines Programms

In der Hochsprache lässt sich die Anweisung zur Berechnung unmittelbar verständlich und in fast mathematischer Form beschreiben (wobei der Operator „=“ hier die Zuweisung des Ergebnisses der Berechnung der Summe a + b an die Variable c bedeutet). Damit der Prozessor die Anweisung verarbeiten kann, benötigt er zusätzliche Informationen: für die Werte der Variablen a, b und c müssen Orte im Speicher gefunden werden, die zugehörigen Speicheradressen A, B und C repräsentieren dann die Variablen.

Anschliessend muss die Berechnung in einzelne Befehle herunter gebrochen werden, wie z.B. rechts in der Abbildung gezeigt. Der Ablauf für den Prozessor ist dann wie folgt: die Werte der Variablen A und B in zwei Register laden, diese beiden Register addieren und das Ergebnis in ein weiteres Register schreiben, schliesslich das Ergebnis an die Adresse C transferieren.

Die Befehle selbst sind auf der Ebene der Maschine strukturiert in den Befehlscode (Opcode), einen binären Code, der für diese Befehle zu vereinbaren ist, sowie in die Operanden. Je nach Befehl mag es unterschiedlich viele Operanden geben, was zu unterschiedlichen Befehlslängen führt. Abhängig von der Art des Befehls sind die Operanden Quellen oder Ziele der Operation.

3.2. Entwicklungsumgebung und Laufzeitumgebung

Die Übersetzung eines in Hochsprache erstellten Algorithmus oder Programms erledigt für uns ein Übersetzungsprogramm: der Compiler. Allerdings ist der Compiler nur ein Teil der insgesamt erforderlichen Entwicklungsumgebung. Üblicherweise arbeitet man heute nicht mehr mit isolierten Komponenten wie Übersetzer (engl. Compiler) und Binder (engl. Linker), sondern mit einer integrierten Entwicklungsumgebung (engl. IDE für Integrated Development Environment). Folgende Abbildung illustriert das Konzept.

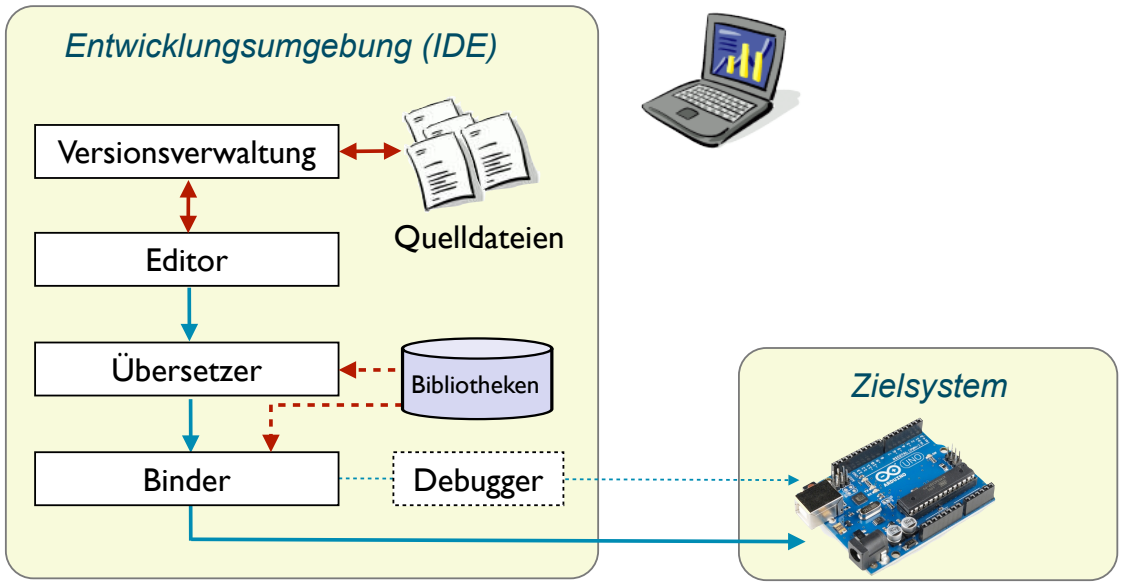


Bild 2.6 Integrierte Entwicklungsumgebung

Die Entwicklungsumgebung läuft nicht auf dem Zielsystem, sondern in aller Regel auf einem Rechner am Arbeitsplatz. Unter einer gemeinsamen Benutzeroberfläche stellt sie für den Anwendungs-programmierer folgende Funktionen beriet: Einen Editor inklusive Verwaltung der Versionen von Programmen und Programmkomponenten (Unterprogramme, Bibliotheken, Projektverzeichnisse etc), den Übersetzer (Compiler) und Binder (Linker), sowie ggf. Programme zur Unterstützung der Fehler-suche (engl. Debugger).

Der Ablauf der Programmierung ist also grundsätzlich gleich geblieben, allerdings bekommt der Anwendungsprogrammierer eine komfortablere Umgebung. In der dargestellten Entwicklungsumgebung wird das fertige Programm auf das Zielsystem (die Laufzeitumgebung) und dort getestet.

In einigen Fällen gibt es vor diesem Schritt auch Simulatoren, die Tests auf der Entwicklungsumgebung ermöglichen. Auf dem Simulator lassen sich leichter logische Fehler und Ungereimtheiten finden als auf dem Zielsystem, das nur sehr eingeschränkte Möglichkeiten zur Fehlersuche bietet. Laufzeitfehler lassen sich allerdings nur sehr beschränkt auf einem Simulator finden.

3.3. Beispiel: Hello World mit LED

In diesem Abschnitt wird die Arduino Entwicklungsumgebung in Betrieb genommen, die Verbindung zur Arduino Baugruppe hergestellt, das erste Programm erstellt und auf die Baugruppe geladen. Das Programm soll eine LED auf der Baugruppe zum blinken bringen.

Übung 3.1: Installieren Sie die Arduino Entwicklungsumgebung (siehe Literaturverzeichnis [4]) auf Ihrem Rechner. Machen Sie sich mit den Funktionen der Entwicklungsumgebung vertraut. Stellen Sie unter dem Menüpunkt „Tools“ das korrekte Arduino Board sowie den korrekten seriellen Port über USB ein (siehe Abbildungen unten).

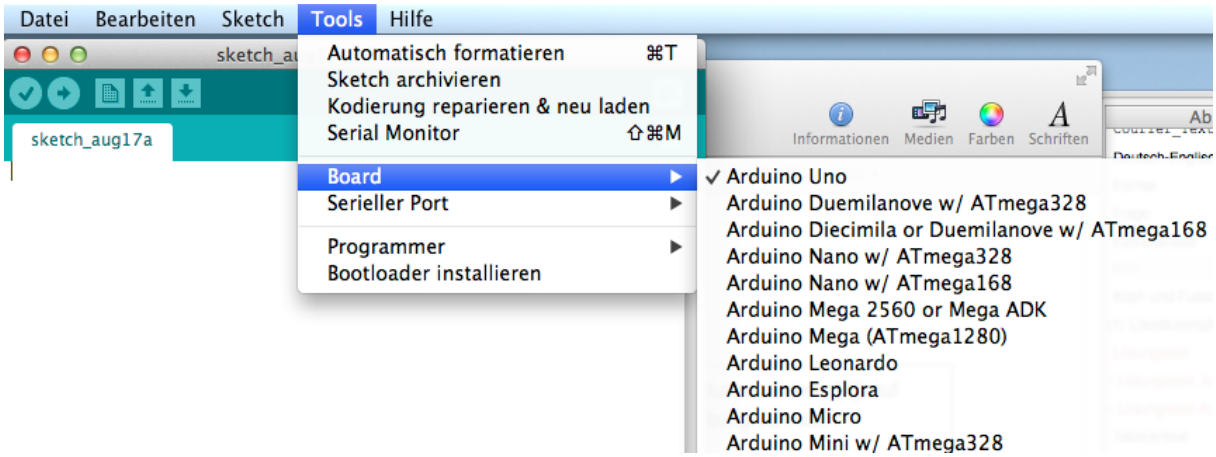


Bild 3.1 Auswahl der korrekten Arduino Baugruppe

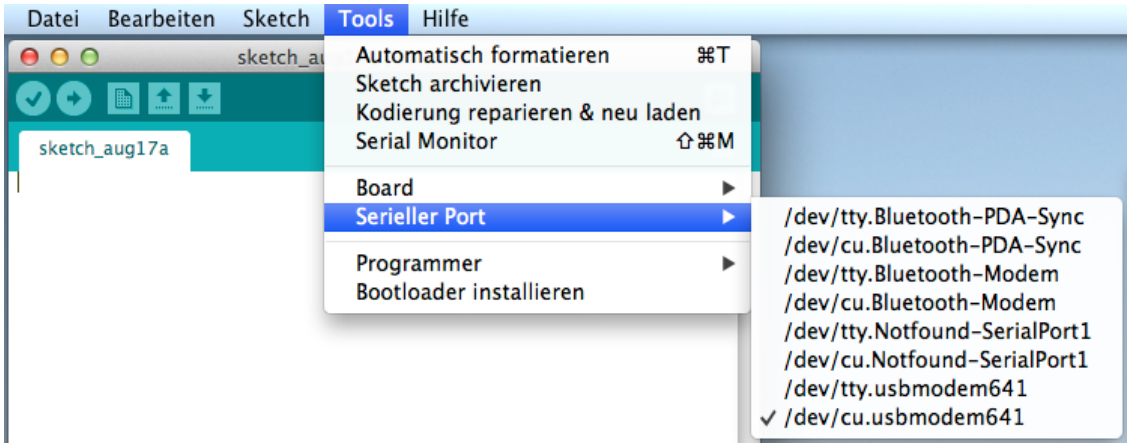


Bild 3.2 Auswahl der korrekten seriellen Schnittstelle zum Arduino Baugruppe

Übung 3.2: Laden Sie aus der Sammlung der Beispielprogramme unter dem Menüpunkt „Öffnen“ under „01.Basics“ das Programm „Blink“. Analysieren Sie den Programmtext (siehe auch unten). Was ist Aufgabe der Funktion setup()? Was ist Aufgabe der Funktion loop()? Ändern Sie das Programm so, dass die LED alle 2 Sekunden für 100 ms blinkt. Übersetzten Sie das Programm („Überprüfen“). Laden Sie das Programm auf das Board („Upload“). Führen Sie das Programm aus.

```
// declare variable name for LED connected to port 13 on the board
int ledPin = 13;

// the setup routine runs once when you press reset:
void setup() {
  // initialize the digital pin as an output.
```

```
  pinMode(ledPin, OUTPUT);
}

// the loop routine runs over and over again forever:
void loop() {
  digitalWrite(led, HIGH);   // turn the LED on (HIGH level)
  delay(100);                // wait for 100 milliseconds
  digitalWrite(led, LOW);    // turn the LED off by (LOW level)
  delay(2000);               // wait for two seconds
}
```

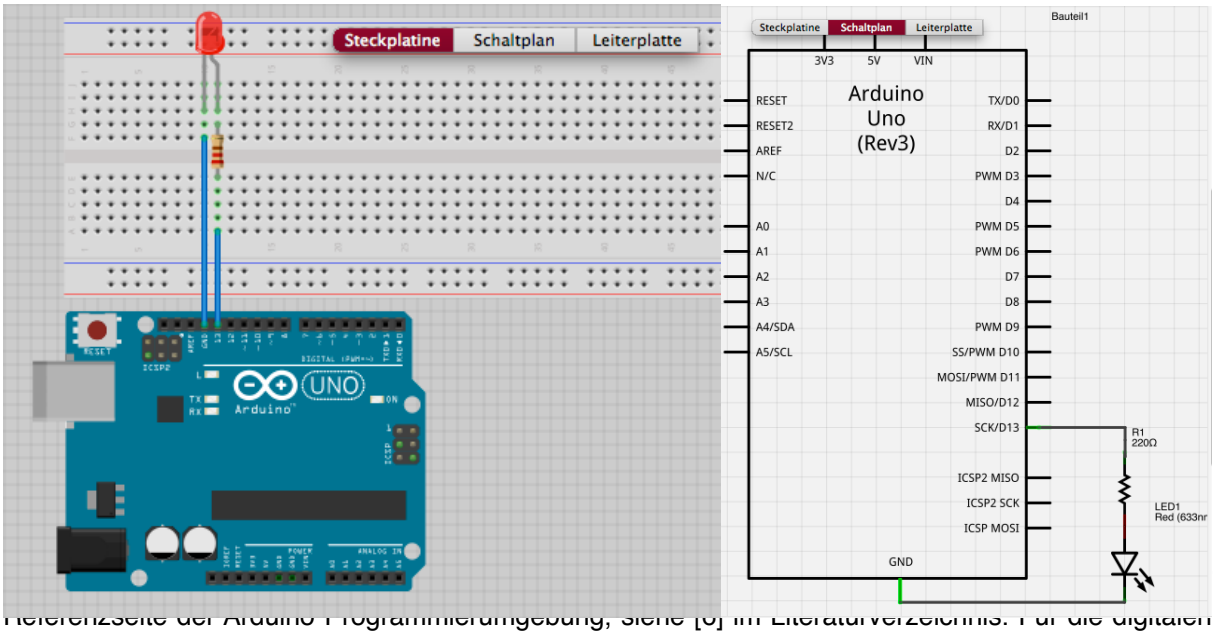
3.4. Externe LED verwenden

In diesem Beispiel wird mit Hilfe des Steckbretts eine externe LED an einem der Ports der Arduino Baugruppe betrieben. Der Aufbau wird hierbei mit Hilfe des Fritzing Editors für Schaltungen nachvollzogen. Dieser Editor gestattet das Übersetzen der Schaltungen auf dem Schaltbrett in Schaltpläne, sowie auch den Entwurf eines passenden Layouts für eine Platine.

Übung 3.3: Laden Sie den Fritzing Editor auf Ihren Rechner (siehe Literaturverzeichnis [5]). Machen Sie sich mit den Funktionen des Editors vertraut. Machen Sie sich mit der internen Verdrahtung des Steckbretts vertraut.

Übung 3.4: Erstellen Sie mit Hilfe des Editors die Schaltung mit Hilfe einer LED aus der Bibliothek, eines Vorwiderstandes (z.B. 220 Ω), sowie der Arduino Baugruppe (zu finden unter Mikrocontroller). Überprüfen Sie Ihre Schaltung in der Ansicht „Schaltplan“ auf Plausibilität.

Übung 3.5: Bauen Sie die Schaltung auf den Steckbrett auf und verbinden Sie sie mit PIN 13 der Arduino Baugruppe. Nehmen Sie die Schaltung mit Hilfe des Programms aus Abschnitt 3.4 in Betrieb.



Referenzseite der Arduino Programmierumgebung, siehe [0] im Literaturverzeichnis. Für die digitalen

Ports finden sich unter dem Stichwort Digital I/O dort folgende Anweisungen. Die Anweisungen werden auch als Funktionen oder Methoden bezeichnet.

```
pinMode();          // schaltet den digitalen Port als Eingang oder
                // Ausgang; Bsp. pinMode(ledPin, OUTPUT);

digitalWrite();     // setzt den logischen Zustand eines Ausgangs-
                // ports auf HIGH oder LOW;
                // Bsp. digitalWrite(ledPin, HIGH);

digitalRead();      // liest den logischen Zustand eines Eingangsports
                    // aus; Bsp. value = digitalRead(inPin)
```

Die ersten beiden Funktionen wurden im Beispiel aus dem vorausgegangenen Abschnitt bereits verwendet. Das Lesen eines digitalen Eingangs funktioniert sinngemäß: (1) Port als Eingang schalten, (2) vom Port lesen. Die Methode digitalRead() wird jedoch anders aufgerufen als die Methode digitalWrite(ledPin, HIGH): Mit value = digitalRead(inPin) wird der Rückgabewert der Funktion an die Variable value übergeben. Diese Schreibweise entspricht der mathematischen Form y = f(x). In der Programmiersprache bedeutet x den Übergabeparameter an die Funktion f(), und y den Rückgabewert der Funktion.

Übung 3.6: Ergänzen Sie die Schaltung aus der letzten Übung um einen Taster. Schreiben Sie ein Programm, das den Taster über einen digitalen Eingang abfragt. Bei gedrücktem Schalter schalten Sie die LED ein, bzw. lassen Sie die LED blinken. Hinweis: Sie benötigen hierzu die Anweisung if().

Für die analogen Ports finden sich auf der Referenzseite (siehe [6]) folgende Funktion unter dem Stichwort Analog I/O.

```
analogRead();       // liest einen Wert vom analogen Eingang;
                // Bsp. value = analogRead(analogPin);
```

Das Format entspricht also der Syntax des Einlesens von einem digitalen Port. Als Rückgabewert wird eine ganze Zahl (integer) beschrieben. Der Wertebereich beträgt 0 bis 1023. Hinter dem analogen Eingangsport verbirgt sich also ein interner 10-Bit Analog/Digitalwandler (A/D Wandler). Das Einlesen eines analogen Wertes funktioniert nach folgendem Schema.

```
int analogPin = 3;   // analog sensor at pin 3
int value = 0;       // sensor value

void setup(){ }      // nothing to set up, analogPin is input already

void loop()
{
  value = analogRead(analogPin);    // read the sensor
}
```

Übung 3.7: Ersetzen Sie den Schalter aus der letzten Übung durch ein Potentiometer. Schreiben Sie ein Programm, das die LED einschaltet, wenn der vom Potentiometer gelesene Wert eine vorgegebene Schwelle überschreitet. Hinweis: Das Potentiometer schalten Sie zwischen Versorgungsspannung (Vcc) und Masse (GND), den Schleifkontakt an einen der analogen Eingänge.

Wie das Blockschaltbild zeigt, verfügt die Arduino Baugruppe über keine analogen Ausgangsports, hat also keine integrierten Digital/Analogwandler (D/A Wandler). Dennoch findet sich unter dem Stichwort Analog I/O auch folgende Funktion.

```
analogWrite();           // schreibt einen analogen Wert per PWM auf
                    // einen digitalen Ausgang;
                    // Bsp. analogWrite(digPin, value);
```

Diese Funktion gibt ein pulsweitenmoduliertes Signal (PWM Signal) aus, wobei das Tastverhältnis dem vorgegebenen analogen Wert entspricht. Allerdings ist ein PWM Signal kein analoges Signal, sondern ein digitales Signal. Folglich ist der Ausgabeport dieser Funktion auch ein digitaler Ausgangsport. Bei einer Sortierung der Funktionen nach der Art der Ausgabe handelt es sich um die Ausgabe eines analogen Wertes. Bei einer Sortierung nach Ports gehört diese Funktion allerdings zu den digitalen Ausgängen.

3.6. Problembehandlung

Was tun, wenn ein Programm sich nicht so verhält, wie man es sich vorgestellt hat? Hierfür ist es hilfreich, unterschiedliche Fehlerkategorien zu kennen:

- Syntaxfehler: Solche Fehler findet der Compiler und und gibt Hinweise. Hierher gehören nicht deklarierte Variable, Zuweisungen von Variablen mit nicht kompatibler Datentypen, fehlende Klammern und Strichpunkte, nicht eingebundene Bibliotheken etc.
- logische Fehler: Hier steckt der Fehler im Konzept: die Formel stimmt nicht, die Adresse einer Komponente ist falsch, eine Schleife ist falsch programmiert, ein Unterprogramm funktioniert nicht etc. Hierfür gibt es gute Lösungsansätze: Das Konzept überdenken, den Programmtext gründlich durchsehen, Kontrollausgaben (Assertions) in das Programm einbauen, um Werte zu überprüfen bzw. den Kontrollfluss zu verfolgen.
- Laufzeitfehler: Hierher gehören Fehler der Sorte Division durch Null, Überschreitung der Grenzen innerhalb einer Datenstruktur, Zeitspanne zur Übertragung der Daten zu kurz gewählt, Speicher füllt sich im Lauf der Zeit bis zum Anschlag etc. Eine Division durch Null oder ein zu weit gegriffener Index führt in aller Regel zu eine Programmunterbrechung mit Fehlermeldung. Andere Laufzeitfehler sind sehr schwer zu finden, da die Logik ja im Prinzip korrekt ist und im Testlauf mit Testausgaben (Assertions) nicht auftreten. Hier helfen im Einzelfall Messungen und das Vorgehen nach dem Ausschlussprinzip, um Fehlerquellen zur an isolieren.

Effektiver als ein krankes Programm zu kurieren ist es, von Anfang an strukturiert vorzugehen. Hierzu hilft die Erstellung eines Konzepts vor der Kodierung, die Zerlegung komplexer Strukturen in beherrschbare Komponenten oder Module, sowie schrittweise vorzugehen. Für letztere Methode kann man beispielsweise eine Aufgabe zerlegen in Tests der Eingänge, Ausgänge und Schnittstellen und andererseits die Logik des Programms, die man mit Testwerten unabhängig von den Eingängen und Ausgängen testen kann.

4. Einfache Programme erstellen

Zur strukturierten Programmierung werden in diesem Abschnitt die Grundlagen vorgestellt. Hierzu gehört eine Übersicht über Algorithmen, Variablen, Datentypen und Anweisungen. Ein wichtiges Konzept ist der Kontrollfluss eines Programms, der sich vor der Kodierung abstrakt beschreiben lässt. Eine Möglichkeit, Systeme in Module zu zerlegen, bieten Unterprogramme bzw. Funktionen.

4.1. Grundlagen der Programmierung

Bei der Programmierung geht darum, ein Problem so exakt zu beschreiben, dass ein Computer es lösen kann. Es ist also eher konzeptionelle Arbeit und Kreativität gefragt, als wildes Kopieren von Programmschnipseln. Vor der Programmerstellung (Kodierung) liegen (1) die Spezifikation der Aufgabenstellung, (2) die Erstellung eines Lösungsverfahrens bzw. Algorithmus.

Unter einem Algorithmus versteht man ein Verfahren, mit dem sich ein Problem schrittweise und präzise lösen lässt. Algorithmen kennt man daher vor allem aus der Mathematik.

Übung 4.1: Erstellen Sie eine Algorithmus für die Berechnung der Summe der Zahlen von 1 bis n. Beschreiben Sie das Vorgehen in Schritten.

Möglicherweise sind Sie bei der Lösung der Aufgabe zu folgendem Ergebnis gekommen:

```
1. Summe <- 0
2. Zahl  <- 1
3. wiederhole, solange wie Summe <= n
 3.1 Summe <- Summe + Zahl
 3.2 Zahl  <- Zahl + 1
```

Was wird nun benötigt, um einen solchen Algorithmus in eine Programmiersprache zu übersetzen? Für die gewählten Platzhalter wie Summe oder Zahl oben müssen Variablen definiert werden. Eine Variable stellt dann eine Behälter für die Werte dar, wie in folgender Abbildung gezeigt.

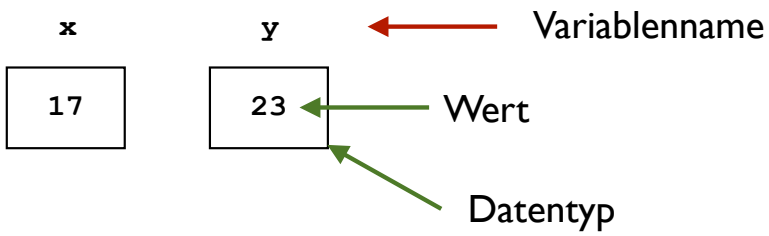


Bild 4.1 Variablen, Werte und Datentypen

Die Variable wird referenziert durch ihren Namen. Der Wert der Variablen kann sich ändern, z.B. durch eine Wertzuweisung. Der Behälter der Werte hat einen bestimmten Datentyp, z.B. eine ganze Zahl, ein Wahrheitswert, bzw. ein Textzeichen. Um einer Variablen den Wert einer anderen Variablen zu übergeben, muss der Datentyp kompatibel sein.

Ein weiterer Bestandteil eines Algorithmus sind die Anweisungen: Hierzu gehören die Wertzuweisung (z.B. y <- 23), die Auswahl (z.B. ist y <= x?) die zu einer Verzweigung führt, sowie Wiederholungen (z.B. solange wie Summe <= N). Für die Wiederholung gibt es auch sogenannte Zählschleifen (z.B. wiederhole 25 mal).

Übung 4.2: Erstellen Sie eine Algorithmus für die Vertauschung zweier Variableninhalte: Am Ende soll die Variable x den Wert der Variablen y haben, und umgekehrt y den Wert der Variablen x. Benennen Sie den Algorithmus tausche(x, y).

Übung 4.3: Erstellen Sie einen Algorithmus, der das Maximum dreier Zahlen ermittelt (imax = maximum (k, l, m)). Versuchen Sie, den Algorithmus anschaulich darzustellen.

Übung 4.3: Erstellen Sie einen Algorithmus, der als Ergebnis n! berechnet (Fakultät, y = faculty(n)).

4.2. Kontrollfluss

Einige Anweisungen führen dazu, dass der Kontrollfluss in einem Algorithmus oder Programm nicht mehr linear verläuft, sondern sich verzweigt. Solche Kontrollstukturen lassen sich abstrakt in Diagrammen darstellen, den sogenannten Aktivitätsdiagrammen. Hierbei wird die Struktur eines Algorithmus oder eines Programms deutlich, ganz unabhängig von den speziellen Aktivitäten innerhalb der einzelnen Stationen. Folgende Abbildung zeigt ein solches Diagramm.

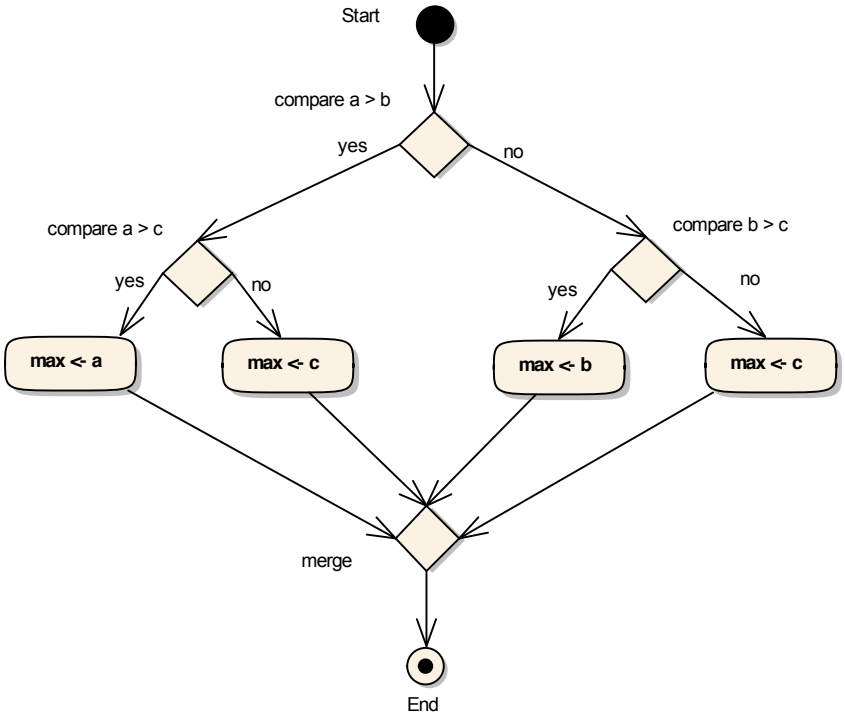


Bild 4.2 Aktivitätsdiagramm

Übung 4.4: Interpretieren Sie das Diagramm zunächst bzgl. der Syntax: Was bedeuten die Rauten? Was bedeuten die abgerundeten Rechtecke? Was bedeuten die Pfeile? Können Sie den Kontrollfluss verfolgen? Hinweis: Finden Sie von der Markierung Start aus die möglichen Wege zum Endpunkt.

Übung 4.5: Welcher Algorithmus ist hier abgebildet?

Kontrollstrukturen lassen sich zerlegen in Aktivitäten und Transitionen zwischen den Wegen. Auf diese Weise entsteht als Diagramm eine Baumstruktur (bzw. mathematisch ausgedrückt ein gerichteter Graph). An den Verzweigungspunkten bzw. Entscheidungspunkten (Rauten) muss man sich für einen der Wege entscheiden: der Kontrollfluss kann nicht in mehrere Wege aufgeteilt werden, die gleichzeitig durchlaufen werden. Die Diagramme kann man wie auf einem Brettspiel mit einer Marke (Spielfigur) vom Startpunkt aus durchschreiten. Der Weg beschreibt hierbei den Kontrollfluss. In der unteren Raute führen mehrere Wege wieder zusammen. Eine Marke wird an dieser Stelle jedoch nur von einem der Wege erwartet.

Die nachfolgende Abbildung beschreibt eine allgemeine Programmstruktur, wie sie auch in den Arduino Projekten in diesem Manuskript Verwendung findet. Hierzu gehören als Aktivitäten die Vorbereitung (Bibliotheken einbinden, Anweisungen an den Compiler geben, globale Variable deklarieren), die Initialisierung (Set-up() Routine), sowie schliesslich die Programmschleife (Loop()).

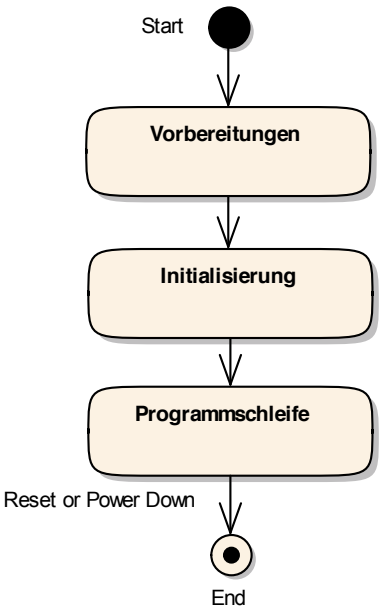


Bild 4.3 Allgemeine Programmstruktur

Wie zu sehen ist, lässt sich diese Methode recht universell für die Struktur von Programmen und Programmausschnitten verwenden. Die Aktivitäten lassen sich, wie in der letzten Abbildung gezeigt, auch erst einmal grob unterteilen und später detaillieren und verfeinern.

In der Arduino Bibliothek findet sich auch eine Methode namens attachInterrupt(). Nachfolgende Abbildung zeigt die Wirkung dieser Methode.

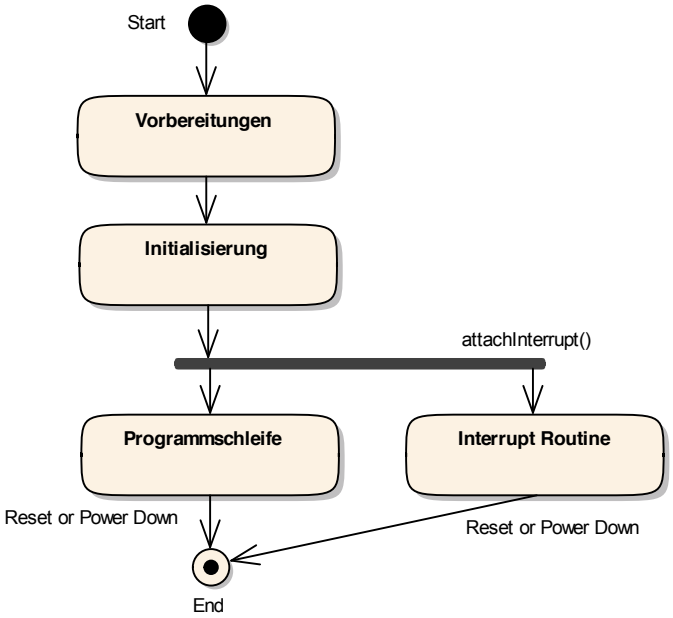


Bild 4.4 Nebenläufigkeit im Kontrollfluss

Übung 4.6: Interpretieren Sie das Diagramm in der Abbildung. Welche Wege lassen sich vom Startpunkt aus durchlaufen? Wie würden Sie mit Spielmarken bzw. Spielfiguren vorgehen? Welche Besonderheit gibt es im Vergleich zu den Verzweigungen in der Abbildung 4.2? Welchen Unterschied machen diese Besonderheiten im Vergleich zum in Abbildung 4.3 gezeigten Ablauf?

Aktivitätsdiagramme zeigen nicht nur die Struktur eines Programms mit seinen Entscheidungsmöglichkeiten, Verzweigungen, Schleifen und Sammelpunkten. Sie ermöglichen auf die Betrachtung nebenläufiger, d.h. parallel zueinander verlaufender Kontrollflüsse. Das ist speziell für Umgebungen mit mehreren kooperierenden Prozessen bzw. mehreren kooperierenden Prozessfäden (Threads) interessant, bzw. für verteile Systeme. Für die Arduino Umgebung beschränkt sich die Nebenläufigkeit auf die Programmierung von Alarmmeldungen (engl. Interrupts).

4.3. Funktionen

Umfangreichere Programmtexte muss man nicht komplett in der Programmschleife unterbringen. Man kann sie auch als Unterprogramm bzw. Funktion am Ende des Programmtextes (hinter der Programmschleife), bzw. in einer separaten Datei unterbringen. Im Hauptprogramm ruft man dann nur noch die Funktion auf. Alle Funktionen bzw. Methoden der Arduino Bibliothek werden so verwendet (siehe z.B. ditalWrite(LedPin, HIGH).

Vom Konzept her betrachtet, fasst man hiermit eine Aktivität im Kontrollfluss in ein Modul zusammen und gewinnt hierdurch im Programmtext an Übersicht. Umgekehrt kann das Aktivitätsdiagramm beim Programmentwurf helfen, eine solche Untergliederung zu finden. Die folgende Abbildung zeigt eine Übersicht über eine solche Struktur.

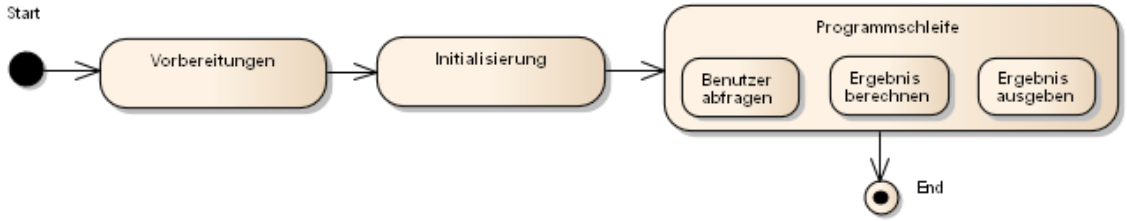


Bild 4.5 Gliederung in Aktivitäten und Unterfunktionen

Übung 4.7: Wie würden Sie den Ablauf innerhalb der Programmschleife in Funktionen untergliedern? Wie werden die Funktionen vom Hauptprogramm aus aufgerufen? Welche Übergabeparameter gibt es? Welche Rückgabewerte gibt es? Wie werden die Funktionen als Unterprogramme geschrieben?

Grundsätzlich haben Funktionen bzw. Methoden die Form y = f(x,y). Hierbei kennzeichnet f() den Namen der Funktion. Die Variablen x und y sind Übergabeparameter. Der Rückgabewert der Funktion ist y. Als Beispiel soll die Funktion $y = x^2 + 2y$ dienen. Für solche Berechnungen ist ein Fliesskommaformat als Datentyp am besten geeignet. Es wird also festgelegt, dass die Übergabeparameter sowie der Rückgabewert Fließkommazahlen vom Typ double sind. Für den Programmtext der Funktion ergibt sich somit:

```
double fun(double a, double b){
   return (a*a + 2*b);
}
```

Hierbei ist durch die Vorgabe der Datentypen bereits alles festgelegt bis auf die Bezeichnung der Übergabeparameter. Mit Hilfe des Schlüsselwortes return wird festgelegt, was als Rückgabewert zurückgegeben wird. Innerhalb der geschweiften Klammern werden die Anweisungen der Funktion spezifiziert. Im Hauptprogramm wird die Funktion dann wie eine Variable vom Datentyp double behandelt. Der Aufruf erfolgt also beispielsweise durch:

```
z = fun(x, y);                  // x, y, z wurden vorher als double
                            // deklariert;
Serial.println(fun(x,y));       // Funktion wird wie eine Variable
                            // vom Datentyp double verwendet;
```

Funktionen ohne Rückgabewert werden mit dem Schlüsselwort void spezifiziert, z.B:

```
void printResult(int i, double x){
 Serial.println("Hier die Ergebnisse");
 Serial.print("Index i: ");
 Serial.print(i);
 Serial.print(" - Wert x: ");
 Serial.println(x);
}
```

Vom Hauptprogramm aus lautet der Aufruf der Funktion dann einfach:

```
    printResult(j, d);             // j wurde vorher als int deklariert
                              // d wurde vorher als double deklariert
```

4.4. Fehlerbehandlung

Bei der Fehlerbehandlung geht es hauptsächlich um das Aufspüren logischer Ungereimtheiten bzw. Ungereimtheiten zur Laufzeit. Hier kann es helfen, die Programmstruktur auch nachträglich einmal bzgl. ihrer Struktur zu analysieren. Wie ist der Ablauf (Aktivitätsdiagramm)? Ist der Kontrollfluss in Ordnung? Auch ein nachträglich rekonstruiertes Aktivitätsdiagramm kann helfen, Ungereimtheiten aufzuspüren.

Eine andere Frage wäre, ob die Module bzw. Unterprogramme vernünftig funktionieren Hier kann es helfen, Kontrollpunkte in das Programm einzubauen und Teile schrittweise mit Hilfe von Testausgaben auf dem seriellen Monitor zu untersuchen. Möglicherweise funktioniert das Programm aber auch ordentlich und der Fehler liegt in der Logik. In diesem Fall muss man das Konzept und die Annahmen nochmals überprüfen.

4.5. Beispiel: Lauflicht

Übung 4.8: Erweitern Sie die Schaltung aus Aufgabe 3.5 auf insgesamt 6 LEDs. Als Anschluss-Pins verwendet Sie beispielsweise die digitalen Ports 8 bis 13.

Übung 4.9: Schreiben Sie ein Programm, das die LEDs in einer vorgegebenen Reihenfolge zyklisch zum Leuchten bringt. Testen Sie das Programm und nehmen Sie die Schaltung in Betrieb. Analysieren Sie Ihre Vorgehensweise.

4.6. Den seriellen Monitor einsetzen

Der serielle Monitor ist eine nützliche Hilfe, um (1) einerseits die Funktion eines Algorithmus unabhängig vom Schaltungsaufbau zu testen, (2) andererseits Programme mit Testausgaben zu versehen und so den Kontrollfluss nachzuvollziehen. In folgenden Programm wurde eine einfache Ausgabe auf den seriellen Monitor verwendet.

```
    // Play with bits & bytes at PORTB (Pins 6 to 13)
    // global variables
    byte lights;

    // setup routine
    void setup() {
      DDRB = B11111100;    // set pins 13 to 8 as output (msb first)
      Serial.begin(9600);  // initialize serial monitor
    }

    // program loop
    void loop() {
       if (lights < 4) {lights = 128; }    // start from 0b100000xx
       Serial.println(lights);             // send test output to serial
```

```
        PORTB = lights;                         // lighting up
        lights = lights >> 1;                   // take one step to the right
        delay(1000);                            // wait a second
    }
```

Für den seriellen Monitor genügen zwei Programmzeilen: (1) in der Set-up Routine wird der serielle Monitor aktiviert (Serial.begin(9600)), (2) in der Programmschleife erfolgt die Ausgabe einer Variablen auf den seriellen Monitor (Serial.println(lights)). Nach Übersetzen und Laden des Programms aktiviert man den seriellen Monitor durch Klicken der Schaltfläche rechts oben in der Entwicklungsumgebung, wie in folgender Abbildung gezeigt.

An den ausgegebenen Werten kann man überprüfen, ob der Algorithmus korrekt arbeitet. Für diesen Schritt ist noch kein Schaltungsaufbau erforderlich. Man kann jedoch nach Überprüfung davon ausgehen, dass der Algorithmus funktioniert. Unabhängig vom Algorithmus kann man die Schaltung mit einem Testprogramm überprüfen, in diesem Fall beispielsweise durch einen Lampentest. Waren beide Schritte erfolgreich, kann man beide Programmteile zusammenfügen und die Funktion nochmals testen. Die Kontrollausgaben lassen sich später auskommentieren bzw. entfernen.

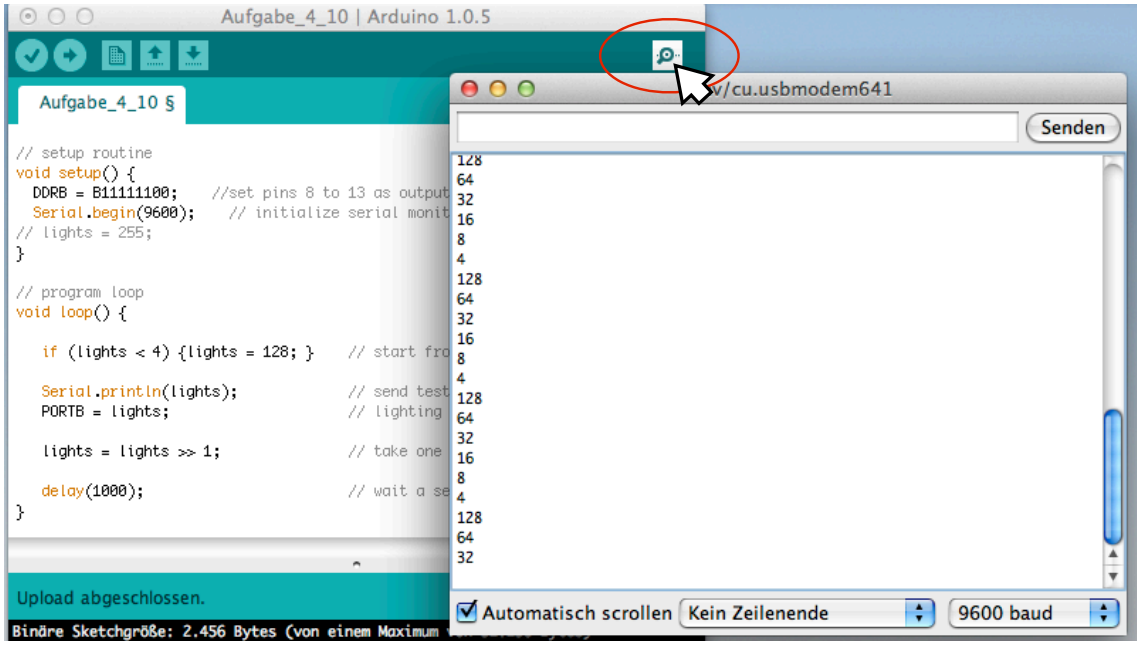


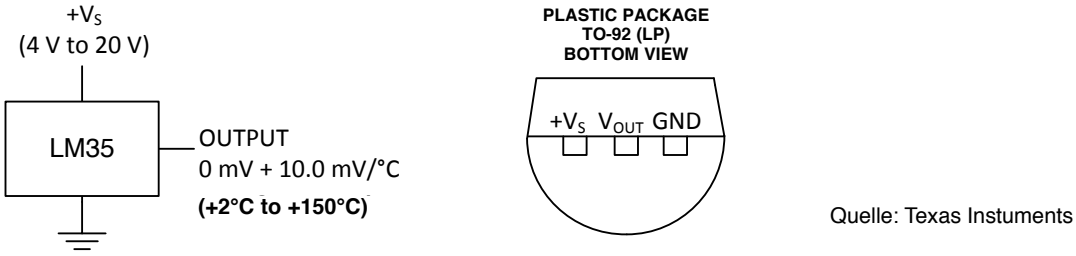
Bild 4.6 Testausgaben auf dem seriellen Monitor

Übung 4.10: Analysieren Sie den Programmcode. Schlagen Sie unbekannte Funktionen bei Interesse auf der Referenzseite [6] nach (siehe auch Port Manipulation).

4.7. Beispiel: Einen Sensor abfragen

Die Abfrage eines Sensors unterscheidet sich programmiertechnisch kaum vom Auslesen eine Potentiometers oder sonst einer Abfrage der analogen Eingangsports. Als Beispiel wird der Temperatursensor LM35 bzw. sonst ein Sensor verwendet. Wie ein Potentiometer, benötigt der Temperatursensor LM35 eine Versorgungsspannung von 4 bis 20V. Folgende Abbildung zeigt einen Ausschnitt aus dem Datenblatt. Der Anschluss erfolgt also ebenfalls wie bei einem Potentiometer: der

mittlere Kontakt liefert die temperaturproportionale Spannung V_{out}. Die beiden äußeren Anschlüsse sind jedoch richtig gepolt zu verwenden, da der Sensor ein Halbleiter-Bauelement ist.



Quelle: Texas Instuments

Bild 4.7 Temperatursensor LM35

Übung 4.11: Bauen Sie eine Schaltung zum Auslesen des Temperatursensors auf. Testen Sie die Schaltung. Hinweis: Die Darstellung der Kontakte in der Abbildung ist die Ansicht von unten (Bottom View). Schreiben Sie ein Programm zur Temperaturmessung. Geben Sie die ausgelesenen Temperaturwerte auf dem seriellen Monitor aus.

Nachdem das Programm läuft und Werte liefert, bleibt die Frage nach der gemessenen Temperatur. Aus dem Datenblatt (siehe Auszug in der Abbildung oben links) ist eine Kennlinie von 10mV pro Grad Celsius (^{0}C) angegeben. Das bedeutet umgekehrt 100 ^{0}C pro Volt. Dieser Wert ist nun noch mit der Referenzspannung und dem Wertebereich des A/D Wandlers zu skalieren. Die Referenzspannung ist in der Standardeinstellung auf 5V eingestellt. Der Wandler ist ein 10-Bit Wandler, liefert also Werte zwischen 0 und 1023. Der Zusammenhang mit der Temperatur lässt sich mathematisch also ausdrücken als

```
temp = (5.0 * analogRead(tempPin) * 100.0) / 1024;
```

Der Kennlinie nach zu urteilen, liefert der Sensor bei 100 ^{0}C eine Spannung von 1 Volt. Die Standardeinstellung der Referenzspannung der A/D Wandler liegt bei 5V, die der Sensor also nie erreichen wird. Somit schöpft man die 10 Bit Auflösung den A/D Wandlers nicht aus, man verschenkt Genauigkeit. Diese Situation lässt sich verbessern, indem man eine weitere Funktion aus der Bibliothek verwendet, die sich unter dem Stichwort Analoge Eingänge und Ausgänge findet.

```
analogReference();              // stellt die Referenzspannung des A/D
                          // Wandlers ein;
                          // default: 5V
                          // INTERNAL: 1.1 V
```

Eine höhere Genauigkeit erreicht man also durch Aufruf von analogReference(INTERNAL).

Übung 4.12: Überarbeiten Sie das Programm, indem Sie auf die interne Referenz des A/D Wandlers umschalten und die Umrechnung in Temperaturwerte anpassen. Geben Sie die Messwerte am seriellen Monitor mit 1 Nachkommastelle in Grad Celsius an.

Das Ergebnis sollte wie folgt aussehen (abhängig von der Temperatur).

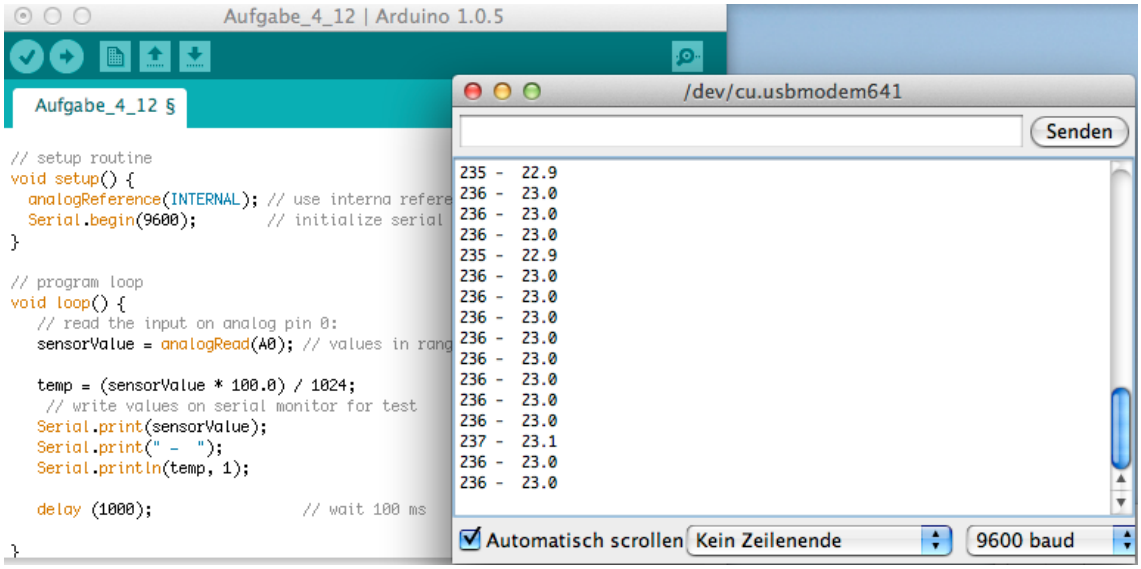


Bild 4.8 Temperaturmessung

5. Zustandsautomaten

Unter einem Zustandsautomaten versteht man eine technische Einrichtung, die abhängig von ihrem inneren Zustand unterschiedlich reagiert. Getränkeautomaten sind beispielsweise so konstruiert, dass sie nur dann ein Getränk auswerfen, wenn vorher das Getränk bezahlt wurde. In diesen lieferfähigen Zustand versetzt man den Automaten, indem z.B. vorher der passende Betrag in Münzen eingeworfen wurde. Ausserhalb dieses Zustandes ignoriert der Automat Aufforderungen zur Getränkeausgabe. Als weiteres Beispiel sei eine Verkehrsampel betrachtet, wie in der folgenden Abbildung gezeigt.

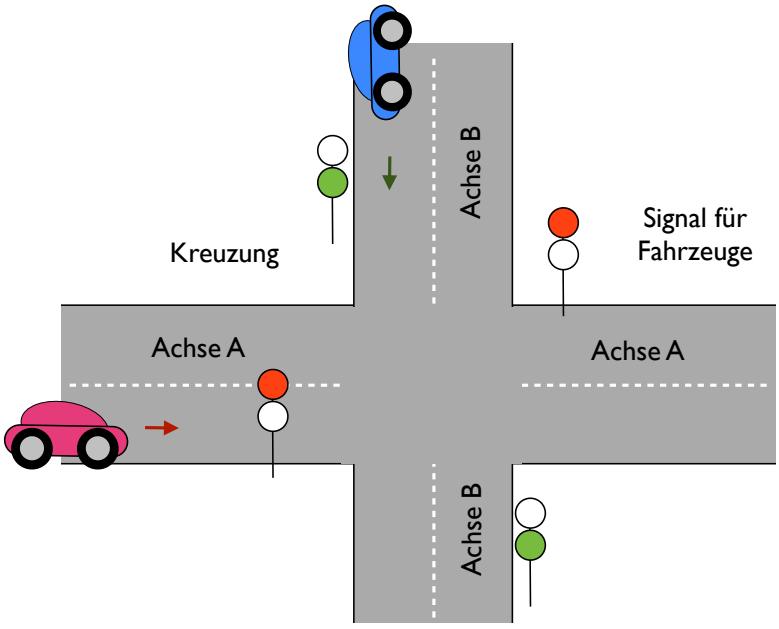


Bild 5.1 Verkehrsampel

Für die Ampelsteuerung als gibt es folgende Zustände: [1] der Verkehr auf Achse A (in der Horizontalen) steht, während der Verkehr auf Achse B (in der Vertikalen) rollt, [2] der Verkehr auf Achse A rollt, während der Verkehr auf Achse B rollt. Zustände dazwischen sind unerwünscht, (Verkehr in beiden Achsen rollt, Verkehr in beiden Achsen steht). Abbiegen in der freigegebenen Richtung ist natürlich erlaubt. Die beiden Zustände der Ampelsteuerung kann man in einem Diagramm darstellen, wie in folgender Abbildung gezeigt.

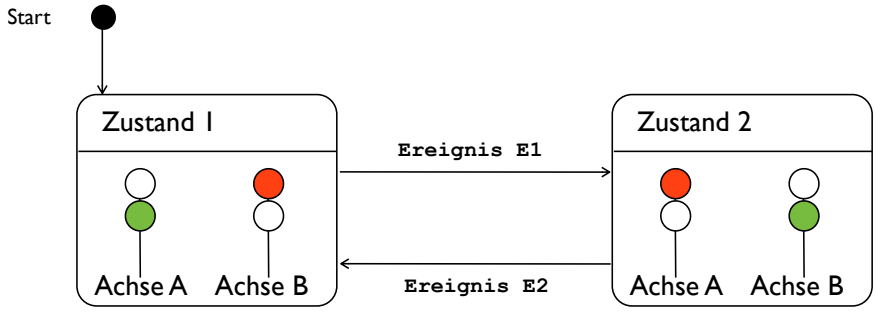


Bild 5.2 Zustandsdiagramm der Ampelsteuerung

Das Zustandsdiagramm (engl. State Diagram) zeigt die beiden Zustände der Ampelsteuerung, sowie die möglichen Zustandsübergänge. Ausserdem zeigt das Diagramm den initialen Zustand, den der Automat nach dem Start (einschalten) ansteuert. Ein Zustandsautomat verbleibt in seinem aktuellen Zustand. Die einzige Möglichkeit, ihn zu einer Zustandsänderung zu bewegen, ist ein passendes äußeres Ereignis. Im Diagramm führt das Ereignis E1 aus dem Zustand 1 in den Zustand 2. Im Zustand 2 reagiert der Automat nicht mehr auf Ereignis E1. Nur durch E2 schaltet er wieder zurück in den Zustand 1.

In der Praxis können die Ereignisse E1 und E2 beispielsweise durch eine Schaltuhr ausgelöst werden. Als Alternative lassen sich interne Timer T1 bzw. T2 verwenden, die die Ereignisse E1 bzw. E2 erzeugen. Die grafische Darstellung im Zustandsdiagramm lässt sich auch in eine Tabellenform übertragen, der sogenannten Zustandsübergangstabelle (engl. State Event Table). Für das Zustandsdiagramm aus der Abbildung oben ergibt sich folgende Zustandsübergangstabelle.

aktueller Zustand	Ereignis (Eingangssignal)	Folgezustand	Ausgangssignal
Z1	E1	Z2	-
Z2	E2	Z1	-

Tabelle 5.1 Zustandsübergangstabelle

Ausgehend von einem beliebigen aktuellen Zustand zeigt die Zustandsübergangstabelle, durch welche Ereignisse Folgezustände erreicht werden. Bei mehr als 2 Zuständen könne mehrere Ereignisse aus den aktuellen Zustand heraus führen. Im Fall der Ampelsteuerung ist zu Tabelle vergleichsweise einfach, enthält jedoch alle Informationen aus dem Diagramm aus dem Zustandsdiagramm. Die Zustandsübergangstabelle lässt sich unmittelbar für den Programm-Entwurf sowie für Tests nutzen. Bei den Tests wird geprüft: (1) ob alle Zustandsübergänge korrekt verlaufen, (2) ob der Automat unempfindlich auf Störungen reagiert.

5.1. Entwurfsmethode

Das Zustandsdiagramm bzw. die eines Automaten lassen sich in eine typische Programmstruktur für eine Automaten übersetzen, die als Entwurfsmuster verwendet werden kann. Die Struktur verwendet das Schlüsselwort switch case (siehe Sprachreferenz [6]). Für den Kern der Steuerung wird diese Struktur wie folgt verwendet:

```
  switch (state) {
    case 1:              // case state equals 1
      // do state actions
      // do state transition (state = nextState)
      break;
    case 2:              // case state equals 2
      // do state actions
      // do state transition (state = nextState)
      break;
    default:
      // if nothing else matches, do the default
      // default is optional
   }
```

Hierbei lässt sich schrittweise vorgehen:

- Zunächst wird nur das Verhalten des Automaten, d.h. die Zustandsübergänge nach dem Zustandsdiagramm programmiert.
- Die Aktionen der Zustände werden definiert zusammen mit den benötigten Datenstrukturen für die Anzeigen der Ampeln.
- Die Aktionen der Zustände werden an passender Stelle in den Zuständen ergänzt.

Für das Zustandsdiagramm aus der Abbildung ergibt sich folgende Programmstruktur. Hinweis: Sie können das Programm per Copy-Paste aus der PDF-Version dieses Manuskripts in Ihre Arduino Entwicklungsumgebung übertragen und dort testen und weiter entwickeln.

```
// Zustandsautomat zur Ampelsteuerung
// global variables
int state;              // state variable
static int T1 = 2000;   // timer for Event 1
static int T2 = 2000;   // timer for Event 2

// setup routine
void setup() {
  state = 1;           // initial state
  Serial.begin(9600);  // initialize serial monitor for tests
}

// program loop
void loop() {

  switch (state) {
    case 1:              // case state equals 1
```

```
          // do state actions:
          // Achse A = grün, Achse B = rot
          delay (T1);       // set timer for Event 1

          // do state transition
          state = 2;        // next state
          break;

        case 2:              // case state equals 2
          // do state actions:
          // Achse A = rot, Achse B = grün
          delay (T2);       // set timer for Event 2

          // do state transition
          state = 1;         // next state
          break;

        default:
          // if nothing else matches, do the default
          Serial.println("undefined state!");
        }

        // write result to serial monitor to test state machine
        Serial.print("state: ");
        Serial.println(state, DEC);

    }
```

Diese Struktur beschreibt nur die Ablaufsteuerung des Automaten. Die Aktionen im jeweiligen Zustand sind nur als Kommentar vermerkt und noch nicht weiter ausgeführt. Die Ablaufsteuerung lässt sich aber bereits testen, wobei die Ausgabe des Zustands über ein Monitorfenster in der Programmierumgebung angezeigt wird. Übersetzen und laden Sie das Programm hierzu auf den Arduino. Aktivieren Sie dann den seriellen Monitor, wie in der folgenden Abbildung dargestellt. Es sollten sich Zustandswechsel mit den per Timer eingestellten Intervallen ergeben.

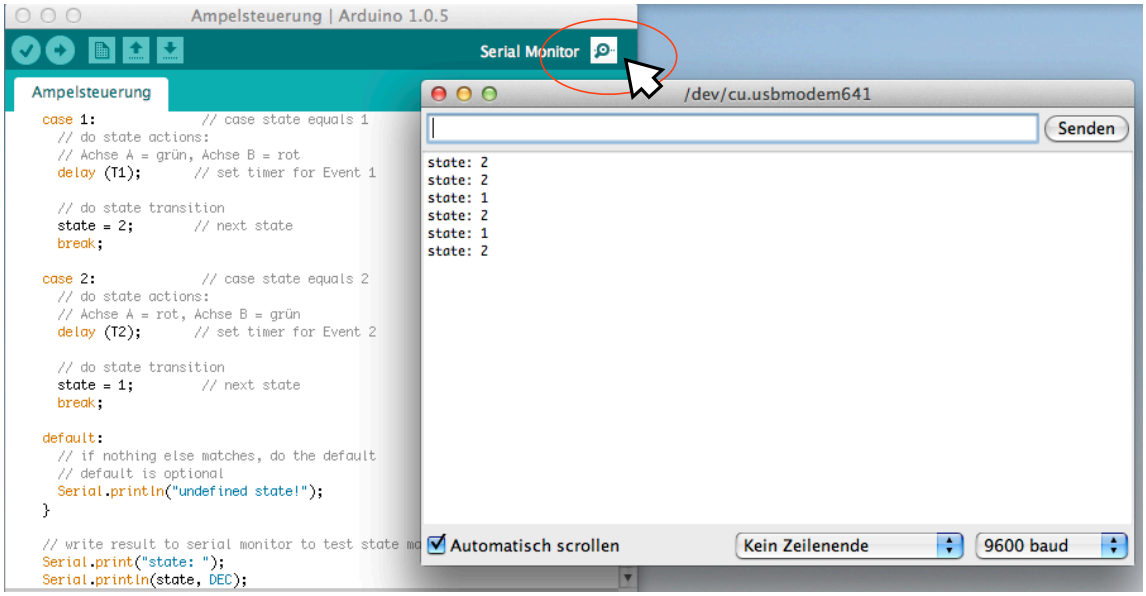


Bild 5.3 Test des Zustandsautomaten

Als nächstes werden nun die Aktionen der Zustände näher definiert und in passende Datenstrukturen zur Ansteuerung der Ampeln übersetzt. Da beide Ampeln jeder Achse gleich angesteuert werden, genügt eine Beschreibung für Achse A und für Achse B. Für beide Ampeln ist eine geeignete Datenstruktur zu wählen, die den Zustand der beiden Lampen wieder gibt. Im folgenden Programmtext wurde hierfür ein Integer-Array mit 2 Feldern gewählt. Jedes Feld repräsentiert eine Lampe der Ampel. Der Wert des Feldes signalisiert des Zustand an (= 1) bzw. aus (= 0).

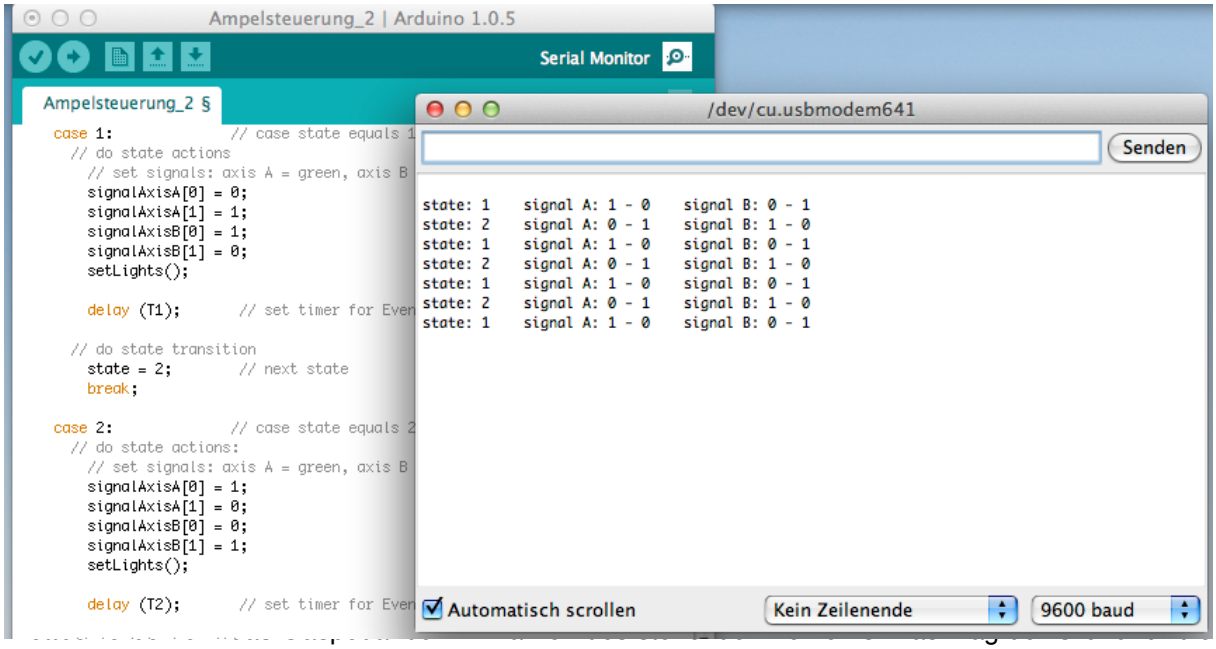
```
// Zustandsautomat zur Ampelsteuerung mit Zustandsaktionen
// global variables
int state;                // state variable
static int T1 = 2000;     // timer for Event 1
static int T2 = 2000;     // timer for Event 2

int signalAxisA[2];       // traffic light at axis A
int signalAxisB[2];       // traffic light at axis B

int pin1AxisA = 7;        // red LED of axis A connects to this pin
int pin2AxisA = 8;        // green LED of axis A connects to this pin

int pin1AxisB = 12;       // red LED of axis B connects to this pin
int pin2AxisB = 13;       // green LED of axis B connects to this pin

// setup routine
void setup() {
  state = 1;          // initial state

  // set pinmodes to Output for all LEDs
  pinMode(pin1AxisA, OUTPUT);
  pinMode(pin2AxisA, OUTPUT);
```

```
  pinMode(pin1AxisB, OUTPUT);
  pinMode(pin2AxisB, OUTPUT);

  // initialize serial monitor for tests
 Serial.begin(9600);
}

// program loop
void loop() {

  switch (state) {
    case 1:               // case state equals 1
      // do state actions
        // set signals: axis A = green, axis B = red
        signalAxisA[0] = 0;
        signalAxisA[1] = 1;
        signalAxisB[0] = 1;
        signalAxisB[1] = 0;
        setLights();

        delay (T1);       // set timer for Event 1

      // do state transition
        state = 2;        // next state
        break;

    case 2:               // case state equals 2
      // do state actions:
        // set signals: axis A = red, axis B = green
        signalAxisA[0] = 1;
        signalAxisA[1] = 0;
        signalAxisB[0] = 0;
        signalAxisB[1] = 1;
        setLights();

        delay (T2);       // set timer for Event 2

      // do state transition
      state = 1;          // next state
      break;

    default:
      // if nothing else matches, do the default
      Serial.println("undefined state!");
    }

    // write result to serial monitor to test state machine
    Serial.print("state: ");
    Serial.print(state, DEC);
```

```
        Serial.print("    signal A: ");
        Serial.print(signalAxisA[0], DEC);
        Serial.print(" - ");
        Serial.print(signalAxisA[1], DEC);
        Serial.print("    signal B: ");
        Serial.print(signalAxisB[0], DEC);
        Serial.print(" - ");
        Serial.println(signalAxisB[1], DEC);
    }

    void setLights(){
      digitalWrite (pin1AxisA, signalAxisA[0]);
      digitalWrite (pin2AxisA, signalAxisA[1]);
      digitalWrite (pin1AxisB, signalAxisB[0]);
      digitalWrite (pin2AxisB, signalAxisB[1]);
    }
```

Vor der Set-up Routine sind noch die Anschluss-PINs für die LEDs der beiden Ampeln zu definieren. Innerhalb der Set-up Routine werden diese Anschlüsse dann als Ausgänge geschaltet. Die Logik der Zustandsaktionen findet sich dann in der Programmschleife: in jedem Zustand wird der Zustand der Lampen beider Ampeln korrekt gesetzt. Das Schalten der Zustände ist dann in ein Unterprogramm verlagert, das sich am Ende des Programmtextes findet.

Ein wichtiges Merkmal eines Zustandsautomaten ist die Unabhängigkeit des Schaltwerks mit den zustandsabhängigen Zustandsübergänge von den individuellen Aktionen jedem Zustand. Auf diese Weise kann erst das Gerüst des Schaltwerks entworfen und getestet werden. Nach Ergänzung der Zustandsaktionen ergibt ein Test das in folgender Abbildung gezeigte Ergebnis..



Verwendung dreier Signallampen in Verkehrsampeln sein: Eine dritte Lampe signalisiert dem rollenden Verkehr in Orange, dass gleich auf Rot umgeschaltet wird. Ebenso signalisiert die dritte Lampe dem stehenden Verkehr den Beginn der grünen Phase.

Hierdurch ergeben sich weitere Zustände, die schonend zwischen den Zuständen Z1 und Z2 vermitteln. Die folgende Abbildung zeigt das diesbezüglich erweiterte Zustandsdiagramm. Zustände, die von Z10 (bisher Z1) nach Z20 (bisher Z2) vermitteln, wurden mit Z11, Z12 und Z12 nummeriert. Sinngemäß ergibt sich der Übergang von Z20 zurück nach Z10.

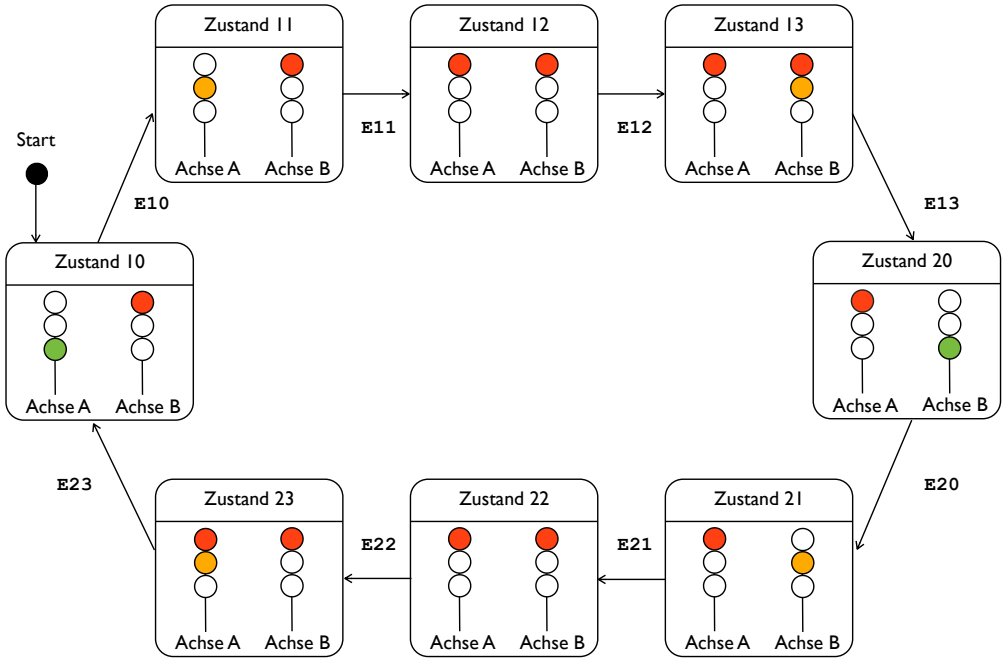


Bild 5.3 Erweitertes Zustandsdiagramm

Bei diesem etwas aufwändigeren Design sollte sich der Vorteil eines Entwurfsmusters zeigen: es kann als Schablone für umfangreichere Strukturen, wie im obigen Zustandsdiagramm, verwendet werden. Insgesamt ergeben sich nun 8 Zustände, die zyklisch durchlaufen werden.

Übung 5.1: Programmieren Sie die Ampelsteuerung und testen Sie die Funktion mit Hilfe des seriellen Monitors.

Übung 5.2: Bauen Sie die erweiterte Ampelsteuerung auf dem Steckbrett auf. Testen Sie die Funktion der Schaltung. Hinweis: Es genügt jeweils eine Ampel für Achse A und eine Ampel für Achse B.

5.3. Eine Chance für Fußgänger

Für die Autofahrer an der Kreuzung ist nun gesorgt. Wie aber schaut es mit Fußgängern aus? Damit Fußgänger planmäßig die Strasse passieren können, wird auf der Achse A ein Fußgängerüberweg gebaut. Es wird außerdem eine Anlage mit zwei Fußgängerampeln installiert, die sich mit einem Taster aktivieren lassen, um den Straßenverkehr bei Bedarf zu unterbrechen. Die folgende Abbildung zeigt die Anordnung. Nun muss die Steuerung der Anlage überarbeitet werden.

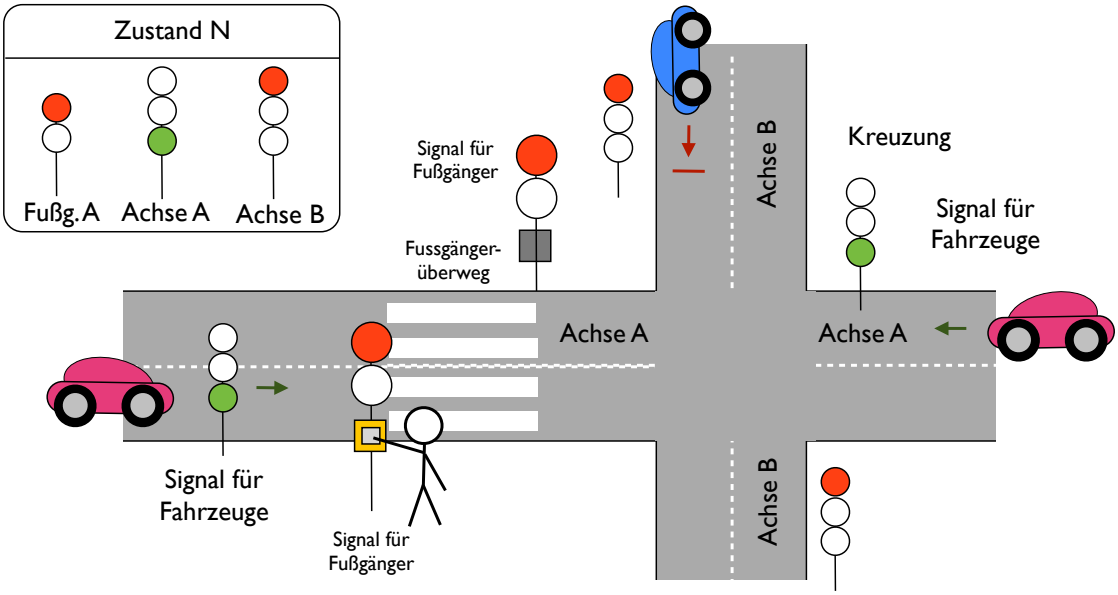


Bild 5.4 Erweiterung um eine Fußgängerampel

Übung 5.3: Erweitern Sie die Ampelsteuerung aus dem letzten Abschnitt um eine Fussgängerampel in Achse A. Verwenden Sie ein Zustandsdiagramm. Testen Sie die Funktion mit Hilfe des seriellen Monitors bzw. durch Aufbau der Schaltung auf dem Steckbrett.

Übung 5.4: Die Schaltung verbessert die Chancen der Fussgänger, hat aber für die Autofahrer unter Umständen den Nachteil, dass der Verkehr jederzeit sofort durch einen Fussgänger unterbrochen werden kann, sofern Sie die Schaltung in dieser Weise realisiert haben. Ändern Sie die Schaltung ggf. so, dass der Verkehr nicht ständig unterbrochen werden kann.

6. Strukturierte Programmierung

Wenn Programme umfangreicher werden, benötigt man Methoden, mit denen sich die Aufgaben so strukturieren lassen, dass überschaubare Module entstehen. Die Aufteilung sollte so sein, dass Änderungen nur begrenzte Auswirkungen auf den gesamten Programmkode haben.

Eine Methode ist die Verwendung bereits fertiger Bausteine, die sich in Programm-Bibliotheken finden. Diese Sammlung lässt sich durch eigene Bibliotheksfunktionen erweitern. Auch ein Element der Bibliothek benötigt eine geeignete Struktur.

6.1. Verwendung von Bibliotheken

Für viele Aufgaben existieren Lösungen, die die Entwickler anderen als Bibliotheksfunktionen zur Verfügung stellen. Unter dem Menüpunkt „Sketch/Library importieren“ finden sich fertige Bausteine für den Anschluss verschiedener externer Komponenten, z.B. mit Hilfe einer seriellen Schnittstelle.

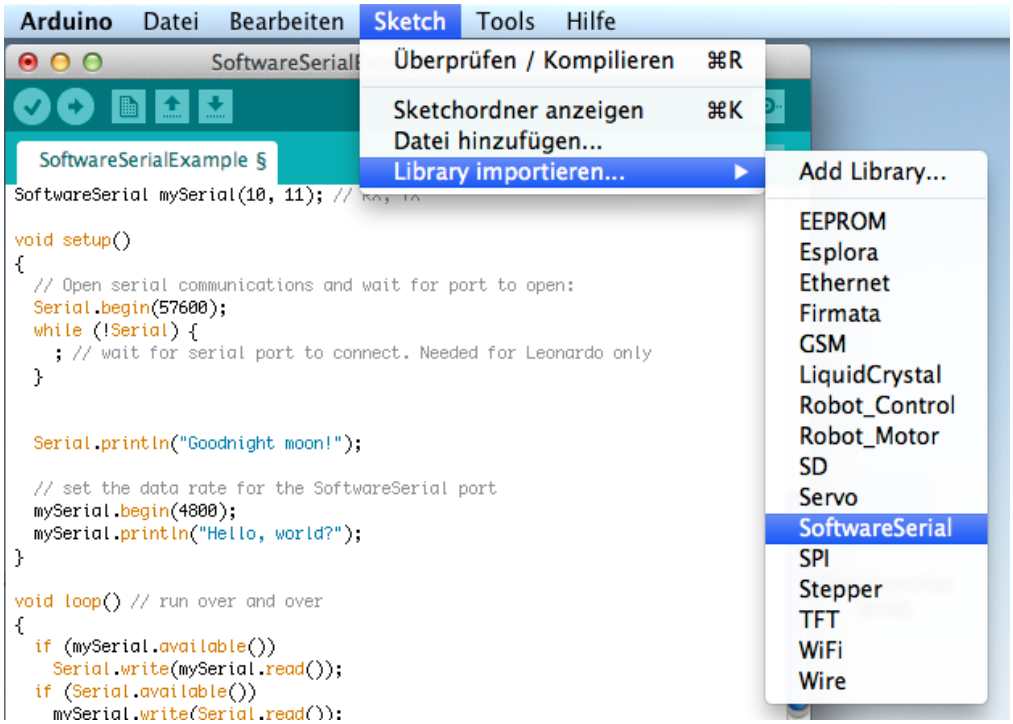


Bild 6.1 Arduino Bibliotheken importieren

Die Bibliothek zur seriellen Schnittstelle besteht hier aus einer kommentierten Beispiel-Anwendung mit allen benötigten Deklarationen, der setup() Routine, sowie der Programmschleife loop(). Für eigene Anwendungen lässt sich die Anwendung geeignet erweitern und anpassen.

Innerhalb der Anwendung aus der Bibliothek finden sich fertige Funktionen zur Initialisierung der seriellen Schnittstelle, zur Prüfung der Verfügbarkeit der Schnittstelle, sowie zum Senden (Schreiben) einer Information über die Schnittstelle (bzw. zum Empfangen bzw. Lesen von der Schnittstelle).

Übung 6.1: Analysieren Sie die Anwendung „SoftwareSerial“ aus der Bibliothek. Welchen Zweck haben die Deklarationen? Was wird initialisiert? Was geschieht in der Programmschleife?

Eigene Programmbeispiele lassen sich ebenfalls in einer Bibliothek unterbringen. Ebenso lassen sich weitere Programmbibliotheken importieren. Eine Anleitung hierfür findet sich in den Anleitungen z.B. unter http://arduino.cc/en/Guide/Libraries.

6.2. Funktionen zur Lösung von Teilaufgaben

Einfache Berechnungen innerhalb eines Programms können direkt im Programmfluss erfolgen. Die in der verwendeten Programmiersprache gegebenen Datentypen sind hierfür meist ausreichend. Für komplexere Berechnungen, bzw. für häufig benötigte Aufgaben lassen sich Funktionen (Unterprogramme) verwenden, bzw. auch eigene Datentypen einsetzen.

Übung 6.2: Schreiben Sie ein Programm, das die Werte a und b vom Benutzer abfragt und das Ergebnis $c = \sqrt{(a^2 + b^2)}$ ausgibt. Hinweis: Verwenden Sie für die Eingabe und Ausgabe den seriellen Monitor (zu aktivieren unter dem Menüpunkt „Tools“).

```
//Aufgabe 6_2
// global variables
int a,b;
double c;

// setup routine
void setup() {
  // initialize serial monitor
 Serial.begin(9600);
}

// program loop
void loop() {

    Serial.println("Bitte Zahl a eingeben: ");
    // read from serial port when data are available
    while (Serial.available() == 0){};
    a = Serial.read() - 48;
    Serial.print("Echo a: ");
    Serial.println(a, DEC);


    Serial.println("Bitte die Zahl b eingeben: ");
    // read from serial port when data are available
    while (Serial.available() == 0){};
    b = Serial.read() - 48;
    Serial.print("Echo b: ");
    Serial.println(b, DEC);

    // calculate and write result to serial monitor
    c = sqrt(a*a+b*b);
    Serial.print("Wurzel(a2+b2): ");
    Serial.println(c, 4);
    Serial.println();

}
```

Übung 6.3: Schreiben Sie das Programm so um, dass zum Lesen der Zahlen sowie zur Ausgabe Unterprogramme bzw. Funktionen verwendet werden. Welche Rolle haben bei Funktionen die Begriffe Übergabeparameter und Rückgabewert? Wie lautet die zugehörige Syntax?

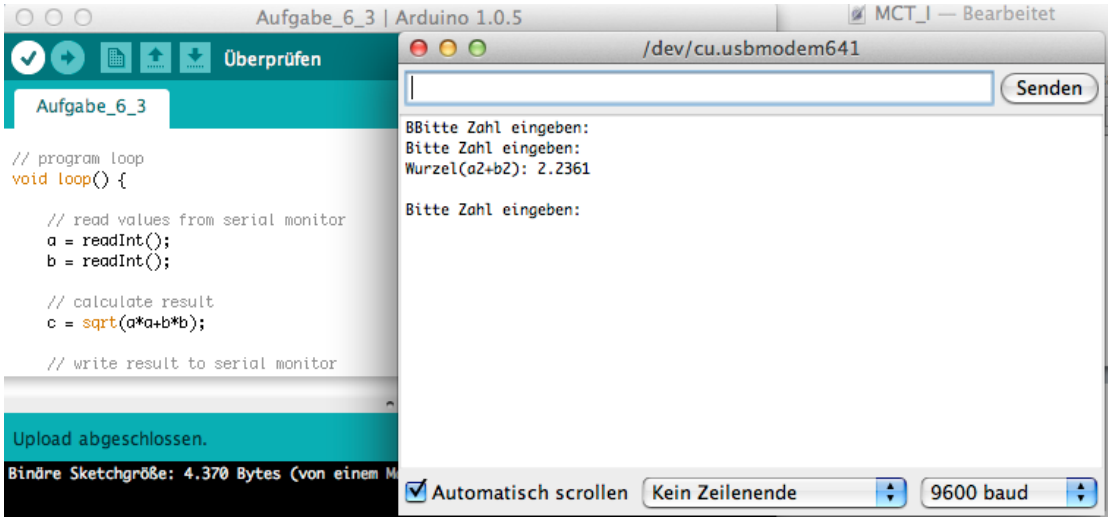


Bild 6.2 Verwendung des seriellen Monitors

```
// Aufgabe 6_3
// global variables
int a,b;
double c;

// setup routine
void setup() {
  // initialize serial monitor
 Serial.begin(9600);
}

// program loop
void loop() {

    // read values from serial monitor
    a = readInt();
    b = readInt();

    // calculate result
    c = sqrt(a*a+b*b);

    // write result to serial monitor
    writeResult(c);

}

// Subroutines (Functions)
int readInt() {
    Serial.println("Bitte Zahl eingeben: ");
    // read from serial port when data are available
    while (Serial.available() == 0){};
    return (Serial.read() - 48);
```

```
}

void writeResult(double d){
    Serial.print("Wurzel(a2+b2): ");
    Serial.println(d, 4);
    Serial.println();
}
```

Übung 6.4: Beim Einlesen der Werte wurden bisher nur einstellige Zahlen verwendet, die von der sogenannten ASCII Kodierung in eine Zahl umgerechnet wurden. Ändern Sie das Programm so, dass sich mehrstellige Werte eingeben lassen. Hinweis: Vereinbaren Sie hierzu ein Zeichen als Hinweis auf das Ende der eingegebenen Zeichen, bzw. verwenden Sie die Option „Neue Zeile (NL)“ des seriellen Monitors (New Line bzw. Line Feed ist in ACSII als 10 kodiert).

Durch die Verwendung der Methoden ist die Programmschleife in Übung 6.3 deutlich übersichtlicher geworden im Vergleich zu Übung 6.2. Ein weiterer Vorteil der modularen Vorgehensweise unter Verwendung von Funktionen sollte sich bei Übung 6.4 zeigen: Die Anpassung der Eingabe betrifft nur die eigene Funktion readInt(). Der übrige Programmtext bleibt unverändert.

Übung 6.5: Folgendes Programm verwendet Zufallszahlen aus der Standard-Bibliothek. Testen Sie das Programm und erkunden Sie die Funktion random().

```
// Aufgabe 6_5
// global variables
long strength, snow;

// setup routine
void setup() {
  // initialize serial monitor
 Serial.begin(9600);
}

// program loop
void loop() {

  Serial.print("Geben Sie die Schneestärke ein [0 - 9]: ");
  while (Serial.available() == 0){};
  strength = Serial.read() - 48;
  delay(20);

  Serial.println();
  for (int j = 0; j < 20; j++){
    for (int i = 0; i < 80; i++){
   snow = random(0, 9);
   if (snow < strength) Serial.print("*");
   else Serial.print(" ");
 } // loop i
```

```
          Serial.println();
          } // loop j

    }
```

6.3. Objektorientierte Programmierung

Ein weiteres Strukturmerkmal neben den Funktionen wird mit dem Begriff der Klasse bezeichnet. Eine Funktion kann einer Klasse zugeordnet werden. Ein Beispiel hierzu fand sich bereits im vergangenen Abschnitt, nämlich die Funktionen der Klasse Serial. Die Funktionen der Klasse werden zusammen mit dem Klassennamen aufgerufen, z.B. Serial.println(„Hallo“), wobei der Klassennamen vor dem Funktionsnamen genannt wird und durch einen Punkt vom Funktionsnamen getrennt ist.

Klassen sind ein Konzept der objektorientierten Programmierung. Klassen entsprechen hierbei Einheiten wie die bereits genannte serielle Schnittstelle, bzw. auch besonderen Datenstrukturen. Als besondere Datenstruktur kämen in der Elektrotechnik z.B. komplexe Zahlen in Frage. Eine Klasse besitzt Eigenschaften (bzw. Attribute), wie bei einer komplexen Zahl z.B. der Realteil und Imaginärteil, sowie die bereits genannten Funktionen, die in der objektorientierten Umgebung Methoden genannt werden.

Wegen der Attribute geht das Konzept der Klassen über das der Unterprogramme (bzw. Funktionen) hinaus. Wenn die Attribute, wie z.B. Realteil und Imaginärteil einer komplexen Zahl, nur über die Klassenfunktionen geändert werden können (z.B. durch die Funktionen get() zum Lesen des Attributes, bzw. set() zum Setzen des Attributes), spricht man von Kapselung der Daten. Statt globaler Variablen und beliebiger Unterprogramme halten Klassenattribute und Klassenfunktionen den Programmierer zu einer klaren Gliederung des Programmtextes an.

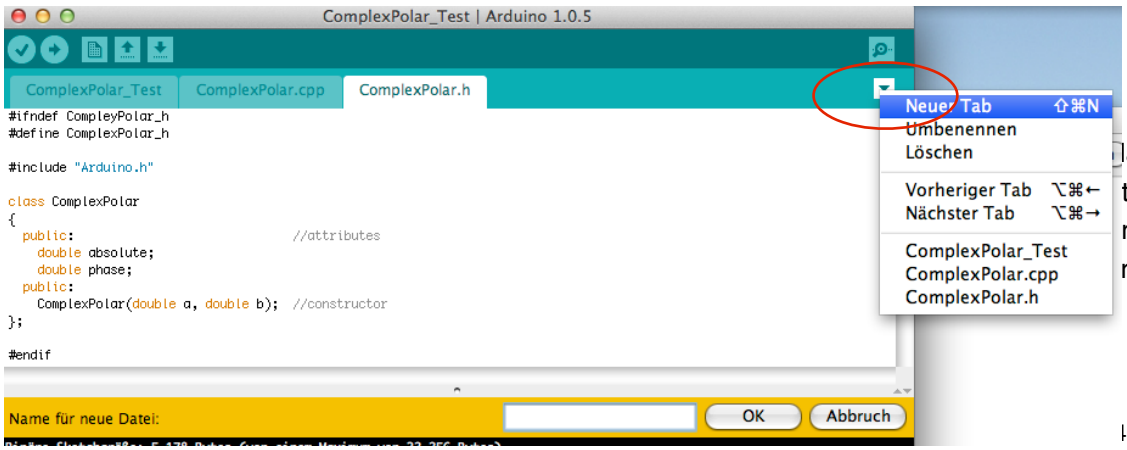
Übung 6.6: Schreiben Sie ein Methode, die zu einer in kartesischen Koordinaten gegebenen komplexen Zahl (d.h. Realteil und Imaginärteil) den Betrag ausrechnet. Benutzen Sie die Methode in einem Programm, das Real- und Imaginärteil vom Benutzer abfragt und den Betrag ausgibt.

Übung 6.7: Erweitern Sie das Programm um eine weitere Methode, die zu einer in kartesischen Koordinaten gegebenen komplexen Zahl (d.h. Realteil und Imaginärteil) die Phase ausrechnet. Erweitern Sie das Programm um die Ausgabe der Phase.

```
// Complex numbers using methods
// global variables
static double pi = 3.1415926535897932; //static = constant value

// setup routine
void setup() {
 // initialize serial monitor
 Serial.begin(9600);
}

// program loop
void loop() {
```

```
    // ask user to enter real part and imag. part over serial monitor
    Serial.println("Pls. enter real part of complex number: ");
    while (Serial.available() == 0){};
    double real = Serial.read() - 48;

    Serial.println("pls. enter imaginary part of complex number: ");
    while (Serial.available() == 0){};
    double imag = Serial.read() - 48;

    // write amplitude and phase to serial monitor
    Serial.println();
    Serial.print("Betrag : ");
    Serial.println(absoluteValue(real, imag), 4);

    Serial.print("Phase : ");
    Serial.print(phase(real, imag), 1);
    Serial.println(" Grad");
    Serial.println();
  }

  // methods = functions

  // calculate absolute value from real part and imag.part
  double absoluteValue(double a, double b){
          return sqrt(a*a + b*b);
  }

  //calculate phase from real part and imag. part
  double phase(double a, double b){
          if(a!=0) return ((90*2/pi)*atan(b/a));
   else return 0;
  }
```

Bis hierher brachte die Nutzung von Methoden bzw. Funktionen nichts Neues. Die Gliederung des Programmtextes in die Hauptschleife, in der die am Ende des Textes aufgeführten Funktionen aufgerufen werden, gab es bereits im vorausgegangenen Abschnitt. Das soll sich jetzt ändern. Hierzu wird ein neuer Datentyp für komplexe Zahlen definiert. Öffnen Sie hierzu bitte einen neuen Reiter in der Arduino-Entwicklungsumgebung, wie in der folgenden Abbildung gezeigt. Geben Sie als Dateinamen „ComplexPolar.h“ ein.



lann als
tion der
r wurde
rag und

```
#ifndef CompleyPolar_h
#define ComplexPolar_h

#include "Arduino.h"

class ComplexPolar
{
  public:                                 //attributes
    double absolute;
    double phase;
  public:
    ComplexPolar(double a, double b);  //constructor
};

#endif
```

Vor den Attributen findet sich das Schlüsselwort public, das kennzeichnet, dass die Attribute ausserhalb der Klasse lesbar und beschreibbar sind. Auf die Attribute greift man in einem Programm, genau wie beim Methodenaufruf, mit Nennung des Klassennamens zu, hier also z.B. mit ComplexPolar.phase. Eine Alternative zum Schlüsselwort public wäre das Schlüsselwort private. In diesem Fall wäre der direkte Zugriff auf die Attribute nicht möglich, sondern nur mit Hilfe der Klassenmethoden.

Den Attributen folgen der Konstruktor der Klasse, sowie die Methoden. Hier ist wiederum das Schlüsselwort public vorangestellt, damit man aus einem Programm heraus auf den Konstruktor zugreifen kann. Der Konstruktor ist eine Methode mit dem Namen der Klasse, hier also ComplexPolar(), die als Übergabeparameter Werte für die Attribute hat. Methoden sind im genannten Beispiel keine definiert. Der Konstruktor dient dazu, in einem Programm ein Objekt der Klasse zu erzeugen, hier also eine komplexe Zahl. Dieser Vorgang entspricht der Deklaration einer Variablen.

Ausserhalb der Klassendefinition finden sich noch einige Angaben, die mit einem Raute-Symbol (#) beginnen. Diese Angaben sind Instruktionen für den Präprozessor und unabhängig von der Klassendefinition. Die ersten beiden Zeilen dienen dazu, zu verhindern, dass die Daten der Header-Datei ComplexPolar.h mehrfach eingebunden werden. Die letzte Zeile (#endif) ist die abschließende Klammer zur ersten Zeile (#ifndef). Außerdem wird in Zeile vier eine Arduino Header-Datei eingebunden.

In der Header-Datei wurde nur die Bezeichnung der Klasse mit ihren Attributen, dem Konstruktor, sowie ggf. der Methoden und der Datentypen definiert. In einer weiteren Datei wird nun spezifiziert, wie der Konstruktor der Klasse und die Methoden zu implementieren sind. Öffnen Sie hierzu einen weiteren Reiter (Tab) und benennen Sie die Datei ComplexPolar.cpp (wobei die Dateiendung cpp auf C++ hin deuten soll).

Folgender Programmtext zeigt die Implementierung des Konstruktors. Zunächst wird die zugehörige Header-Datei eingebunden (sowie die Arduino Header Header-Datei). Dann folgen hinter dem Methodennamen mit den beiden Übergabeparametern in geschweiften Klammern die Schritte zur Implementierung. Hier werden also beide Attribute der Klasse mit den Werten aus den Übergabe-parametern initialisiert. Eine Alternative wäre gewesen, die Attribute immer mit den Werten Null zu initialisieren. Sofern die Klasse Methoden besitzt, folgt die Implementierung der Methoden. Durch die Verwendung einer eigenen Datei bleibt der Programmtext des Hauptprogramms schlank.

```
#include "Arduino.h"
#include "ComplexPolar.h"

// implement constructor
ComplexPolar:: ComplexPolar(double a, double b)
{
  absolute = a;
  phase = b;
}

// implement methods
// ...
```

Nach der Definition der Klasse in der Header-Datei ComplexPolar.h und nach der Implementierung des Konstruktors (und ggf. der Klassenmethoden) kann die Klasse nun in einem Hauptprogramm verwendet werden. Öffnen Sie hierzu einen weiteren Reiter (Tab) und benennen Sie ihr Hauptprogramm. Folgender Programmtext zeit ein Beispiel.

```
// Complex numbers using complex class without methods

#include "Arduino.h"
#include "ComplexPolar.h"

// global variables
static double pi = 3.1415926535897932; //static = constant value

ComplexPolar cp(0,0); // create new complex polar cp
                // by calling the constructor

// setup routine
void setup() {
 // initialize serial monitor
 Serial.begin(9600);
}

// program loop
void loop() {

  // ask user to enter real part and imaginary part over serial mon.
  Serial.println("Pls. enter real part of complex number: ");
  while (Serial.available() == 0){};
  double real = Serial.read() - 48;

  Serial.println("pls. enter imaginary part of complex number: ");
  while (Serial.available() == 0){};
  double imag = Serial.read() - 48;
```

```
    // calculate amplitude and phase
    cp = convert(real, imag);

    // write amplitude and phase to serial monitor
    Serial.println();
    Serial.print("Betrag : ");
    Serial.println(cp.absolute, 4);

    Serial.print("Phase : ");
    Serial.print(cp.phase, 1);
    Serial.println(" Grad");
    Serial.println();
}

// Methods
ComplexPolar convert(double a, double b){
    cp.absolute = sqrt(a*a + b*b);
    if (a!=0) cp.phase = (90*2/pi)*atan(b/a); //Realteil = 0
    else {
 if (b>=0) cp.phase = 90; //real part=0 and imag. part positive
 else cp.phase = -90; //real part=0 and imag. part negative
    }
    return cp;
 }
```

Im Vergleich mit dem Programm aus Übung 6.7 zeigen sich folgende Unterschiede:

- Bei der Deklaration der globalen Variablen wird der Konstruktor der Klasse aufgerufen , der das Objekt cp mit Datentyp ComplexPolar erzeugt (siehe ComplexPolar cp(0,0)). Der Objektname ist hierbei beliebig. Der Datentyp des Objekts entspricht seiner Klasse, daher wird hier der Klassenname als Typenbezeichnung angegeben.
- In der Programmschleife werden die vom Benutzer abgefragten Werte der Methode convert() übergeben. Diese Methode ist nun vom Typ ComplexPolar. Daher kann man den Rückgabewert der Methode dem Objekt cp direkt übergeben (siehe cp = convert(real, imag)).
- Für die Ausgabe der Werte werden anschließend die Attribute der Klasse verwendet (siehe Serial.println(cp.absolute, 4) und Serial.print(cp.phase, 1)).

Übung 6.8: Führen Sie das Programm aus und prüfen Sie auf Korrektheit der Berechnung.

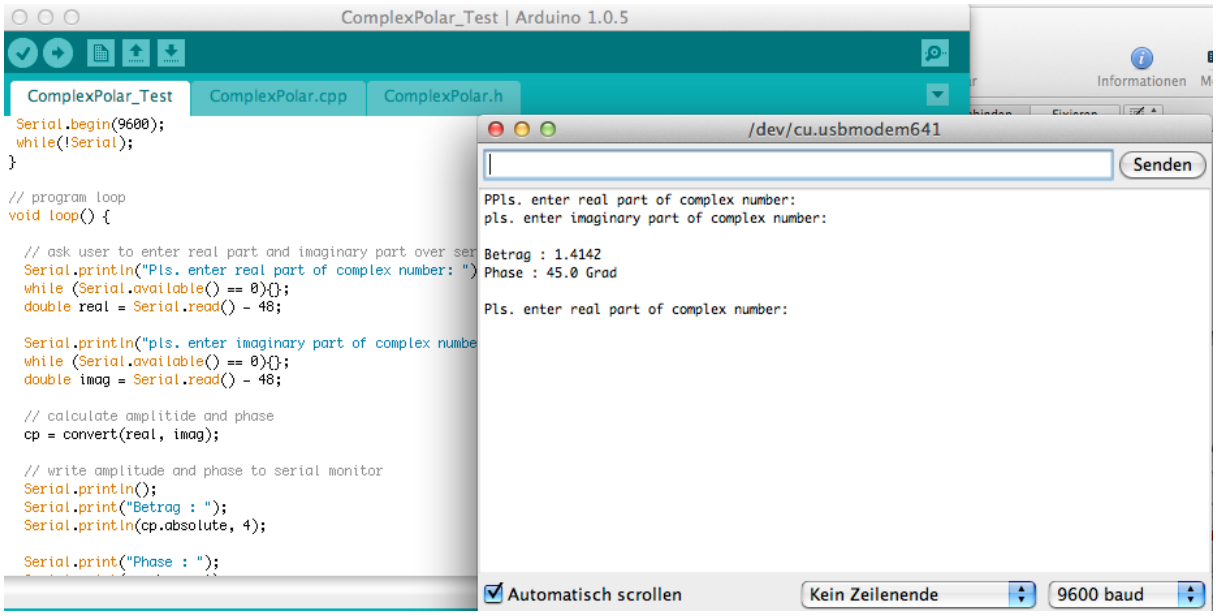


Bild 6.3 Komplexe Zahlen als Klasse

Welchen Vorteil hat die Verwendung von Klassen? Bei Verwendung mehrerer komplexer Zahlen wäre immerhin die Zugehörigkeit der Attribute Betrag und Phase zu den jeweiligen Variablen klar geordnet. Einen größeren Fortschritt Richtung Übersichtlichkeit hat die Verwendung der Klasse bisher allerdings nicht gebracht. Hierzu wären statt der Methode am Programmende die Methoden zur Berechnung in der Klasse als Klassenmethoden zu implementieren.

Übung 6.9: Implementieren Sie die Berechnung von Betrag und Phase aus dem Realteil und dem Imaginärteil als Klassenmethode. Ändern Sie das Programm entsprechend und testen Sie seine Funktion. Hinweis: Erweitern Sie zunächst die Header-Datei um die Methodennamen. Implementieren Sie dann die Methoden in der Datei mit Endung _.cpp. Verwenden Sie dann die Klassenmethoden im Hauptprogramm. Hinweis: bei der Implementierung der Methode achten Sie bitte auf den Bezug zur Klasse (korrekte Syntax: void ComplexPolar::convert(double re, double im)).

Übung 6.10: Ändern Sie den Status der Klassenattribute von public auf private. Ergänzen Sie Methoden zum Setzen der Attribute (set...()) und zum Lesen der Attribute (get...()). Ändern Sie die Programmdateien und testen Sie das Programm.

Wie der folgende Programmtext zeigt, ist das Programm nun durch die Verwendung von Klassen und Klassenmethoden deutlich übersichtlicher geworden. Die Konversion ist wird nun als Klassenmethode aufgerufen (siehe cp.convert(real, imag)). Die Abfrage der Klassenattribute geschieht durch „getter“-Methoden (siehe Serial.println(cp.getAmp(), 4) und Serial.print(cp.getPhase(), 1)).

```
// Complex numbers using complex class with class methods

#include "Arduino.h"
#include "ComplexPolar2.h"

// global variables
```

```
ComplexPolar cp(0,0); // create new complex polar cp
                      // by calling the constructor

// setup routine
void setup() {
 // initialize serial monitor
 Serial.begin(9600);
}

// program loop
void loop() {

  // ask user to enter real part and imaginary part over serial mon.
  Serial.println("Pls. enter real part of complex number: ");
  while (Serial.available() == 0){};
  double real = Serial.read() - 48;

  Serial.println("pls. enter imaginary part of complex number: ");
  while (Serial.available() == 0){};
  double imag = Serial.read() - 48;

  // calculate amplitude and phase
  cp.convert(real, imag);

  // write amplitude and phase to serial monitor
  Serial.println();
  Serial.print("Betrag : ");
  Serial.println(cp.getAmp(), 4);

  Serial.print("Phase : ");
  Serial.print(cp.getPhase(), 1);
  Serial.println(" Grad");
  Serial.println();
}
```

Für Klassen mit ihren Attributen und Methoden, sowie auch für Beziehungen von Klassen zueinander gibt es auch eine grafische Schreibweise, das sogenannte Klassendiagramm. Folgende Abbildung zeigen Klassendiagramme für komplexe Zahlen.

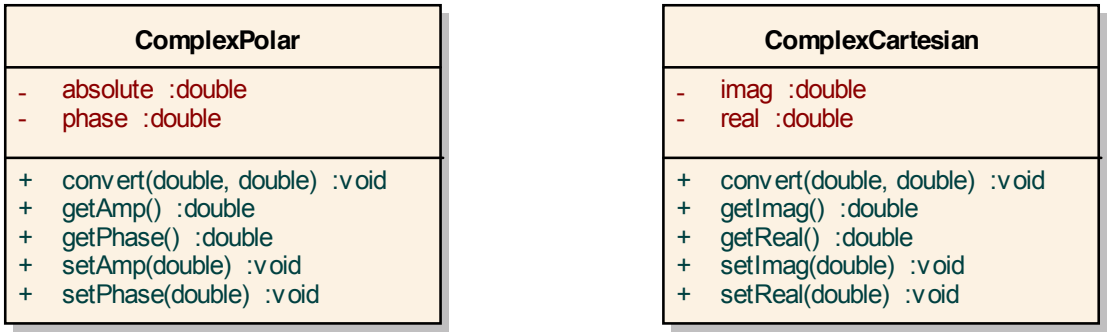


Bild 6.4 Klassendiagramme

Übung 6.11: Interpretieren Sie das Diagramm auf der linken Seite. Wo finden sich Klassenname, Attribute und Methoden? Wo finden sich die zugehörigen Datentypen? Wie sind private und öffentliche (public) Deklarationen gekennzeichnet?

Übung 6.12: Interpretieren Sie das Diagramm auf der rechten Seite der Abbildung.

Diagramme wie das in der Abbildung gezeigte Klassendiagramm sind in einem Standard namens UML (Unified Modelling Language) spezifiziert. Für UML gibt es auch Editoren zum Erstellen der Diagramme. Die Diagramme lassen sich mit Hilfe des UML-Editors unmittelbar in Quellcode für verschieden Programmiersprachen wie Java oder C++ übersetzen. Folgender Text wurde aus dem Diagramm in der Abbildung automatisch erzeugt. Wie man sieht, lässt sich der Text sofort als Header-Datei verwenden, sowie als Vorlage für die Implementierung der Methoden.

```
//////////////////////////////////////////////////////////
//  ComplexCartesian.h
//////////////////////////////////////////////////////////

class ComplexCartesian
{

public:
 ComplexCartesian();
 virtual ~ComplexCartesian();

 void convert(double am, double ph);
 double getImag();
 double getReal();
 void setImag(double b);
 void setReal(double a);

private:
 double imag;
 double real;

};
```

Der Text enthält übrigens noch ein weiteres Element der objektorientierten Programmierung: unterhalb des Konstruktors ComplexCartesian() findet sich der Destruktor ~ComplexCartesian(). Ein Aufruf des Destruktors löscht das erzeuge Objekt wieder aus dem Arbeitsspeicher. Diese Funktion wird bei umfangreicheren Programmen verwendet, um nicht mehr benötigte Objekte zu löschen. Bei Programmende werden alle erzeugten Objekte automatisch gelöscht und der Arbeitsspeicher freigegeben.

Übung 6.13: Schreiben Sie ein Programm zur Berechnung der Impedanz Z der Parallelschaltung zweier komplexer Impedanzen Z_1 und Z_2. Geben Sie Z als Realteil und Imaginärteil aus, sowie in Betrag und Phase.

7. Signalgenerator

7.1. Direkte Digitale Synthese (DDS)

In diesem Abschnitt wird ein etwas komplexeres Programm erstellt: Ein Signalgenerator nach dem DDS-Verfahren (Direkte Digitale Synthese). Bei diesem Verfahren wird das Signal rein numerisch erzeugt. Die Signalform wird in einer Tabelle abgelegt, z.B. ein kosinusförmiges Signal mit 512 Stützstellen, wie in der folgenden Abbildung gezeigt. Wenn man die Tabelle nun wiederholt ausliest, wiederholt sich das Signal periodisch.

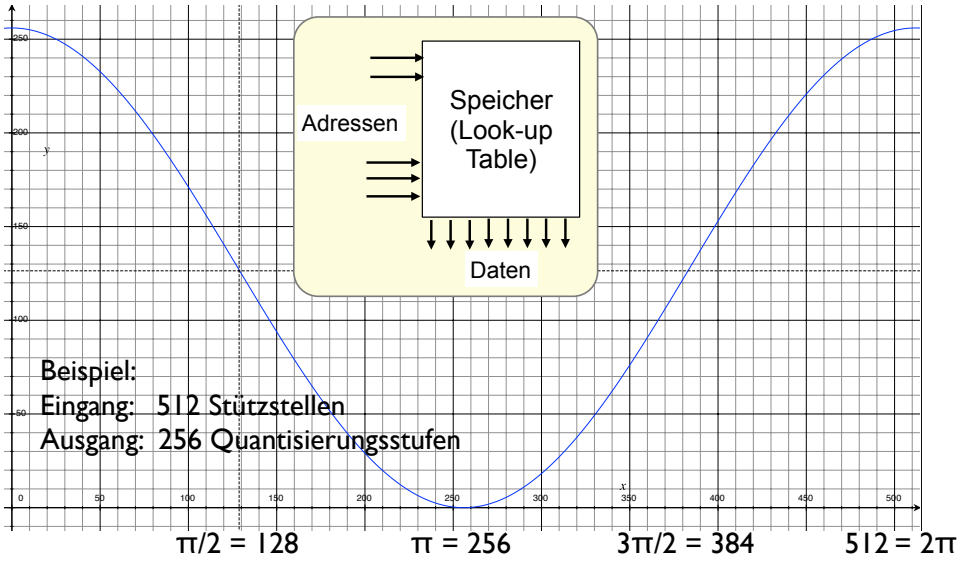


Bild 7.1 Funktionstabelle

Die Frequenz des Signals lässt sich ebenfalls variieren: Liest man z.B. nur jede zweite Stützstelle aus, so verdoppelt sich die Frequenz. Gemessen an der Taktrate, mit der man die Funktionstabelle ausliest, lassen sich durch Änderung der Schrittweite also Vielfache der Frequenz erzeugen, die sich bei Auslesen mit minimaler Schrittweite ergibt.

Ein Beispiel: Die Tabelle hat 512 Stützstellen. Das Intervall zwischen zwei Abfragen der Tabelle beträgt 1 ms. Die Tabelle wird mit Schrittweite eins ausgelesen. Zum Auslesen einer kompletten Periode werden also 512 ms benötigt, die Frequenz beträgt somit 1/(512 ms) = 1,95 Hz. Liest man die Tabelle mit Schrittweite 16 aus, ergibt sich die 16-fache Frequenz. Die größte mögliche Frequenz erreicht man mit Schrittweite 256: hier ergeben sich pro Periode nur noch 2 ausgegebene Stützstellen. Weniger als zwei Stützstellen pro Periode sind zum korrekten Abtasten einer periodischen Funktion nicht möglich.

7.2. Struktur und Funktionsblöcke des Signalgenerators

Zum Programmentwurf wird ein Blockschaltbild verwendet, das die benötigte Komponenten und deren Anordnung zeigt, siehe folgende Abbildung. Kern des Signalgenerators ist die Funktionstabelle

zum Nachschlagen der Funktionswerte (engl. Look-up Table). Das digitale Ausgangssignal wird in ein analoges Signal gewandelt. Ein Tiefpassfilter glättet das Ausgangssignal passend zum gewählten Frequenzbereich.

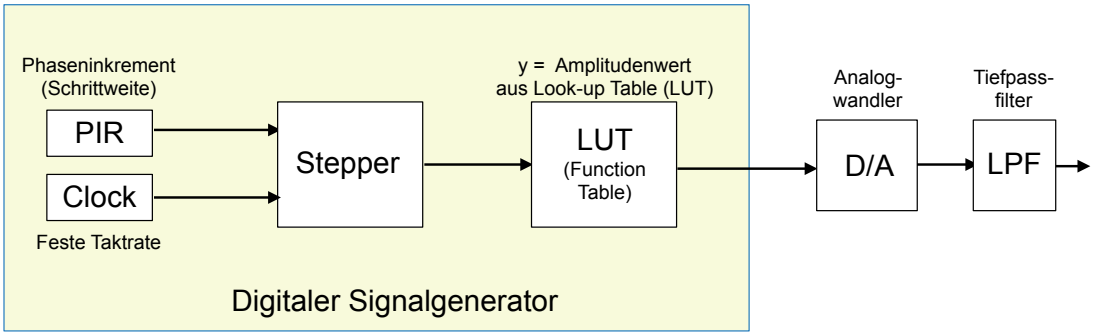


Bild 7.2 Blockschaltbild des Signalgenerators

Im Blockschaltbild erfolgt die Abfrage der Funktionstabelle durch den Funktionsblock „Stepper“. Die Abfrage erfolgt mit dem vorgegebenen Taktzyklus, der im Blockschaltbild als externer Takt (Clock) dargestellt ist. Aufgabe des Steppers ist es, mit diesem Takt die Adresse der Funktionstabelle um die vorgegebene Schrittweite hoch zu zählen. Die Schrittweite ist mit Bezug auf die Phase der Funktion in der Tabelle auch als Phaseninkrement bezeichnet. Das Phaseninkrement wird als Anzahl der Stützstellen angegeben und lässt sich über eine Periode von 2π in die Phase der Funktion umrechnen, wie in der Abbildung im ersten Abschnitt gezeigt.

7.3. Aufbau der Komponenten

Zunächst wird ein stark vereinfachter Aufbau gewählt. Als Ausgang soll eine LED dienen, die mit sehr langsamen Frequenzen angesteuert wird, so dass das periodische Signal mit bloßem Auge erkennbar ist. Hierzu genügen Intervalle von 100 ms für die Abfrage der Funktionstabelle. Mit insgesamt 512 Stützstellen ergeben sich als untere Grenzfrequenz somit 0,019 Hz und als obere Grenzfrequenz ca 5 Hz.

Für diese wenig zeitkritische Anordnung wird auf ein externes Taktsignal verzichtet, die Taktung kann mit dem internen Timer über die delay() Methode erfolgen. Die Frequenz soll allerdings einstellbar sein. Hierfür wird ein Potentiometer vorgesehen, aus dessen Stellung die Schrittweite (das Phaseninkrement) abgeleitet wird.

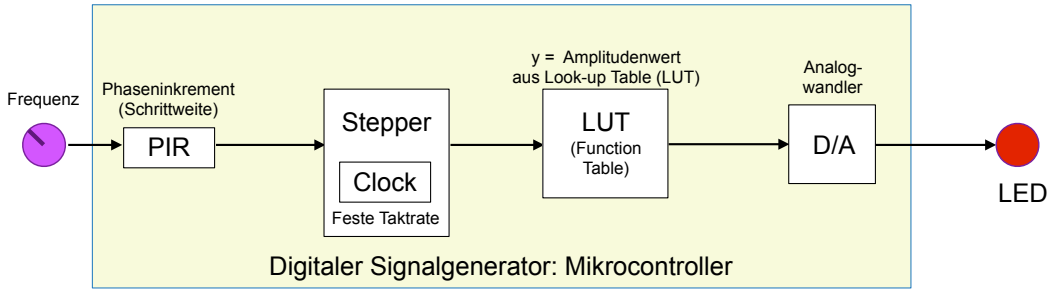
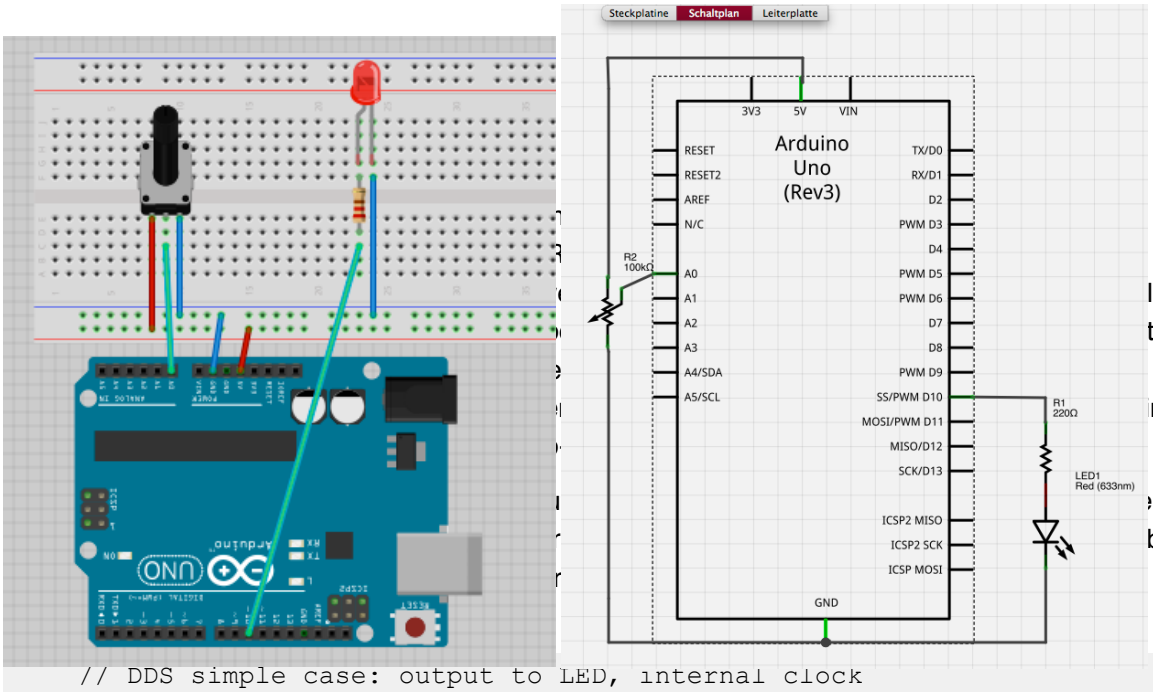


Bild 7.3 Blockschaltbild für den vereinfachter Aufbau

Dem Blockschaltbild ist die externe Beschaltung zu entnehmen: Zur Einstellung der Frequenz ist ein Potentiometer angeschlossen, sowie zur Ausgabe eine LED. Alle anderen Komponenten sind

als Software zu realisieren. Die LED wird an einen der digitalen Ports angeschlossen, an denen die Ausgabe analoger Werte per PWM möglich ist, also z.B. Port D10, wie im folgenden Schaltplan gezeigt. Die PWM kann hierbei als 1-Bit D/A-Wandler aufgefasst werden, der seriell mit erhöhter Datenrate betrieben wird. Das Tiefpassfilter wird in diesem Fall durch die Trägheit des menschlichen Auges realisiert. Das Potentiometer wird über den analogen Eingang A0 eingelesen.




```
// DDS simple case: output to LED, internal clock
// step 1: operate peripherals

int ledPin = 10;      // LED connected to digital pin 10
int sensorValue;      // value read from analog port A0 (AD converter)

// setup routine
void setup() {
  // initialize serial communication for tests
  Serial.begin(9600);
}

// program loop

void loop() {
 // read the input on analog pin 0:
 sensorValue = analogRead(A0); //sensorValues in range 0-1023

 // write values on serial monitor for test
 Serial.println(sensorValue); //test on serial monitor

 // write value to LED via PWM
 analogWrite(ledPin, sensorValue/4);  //scale to 0-255 for analogWrite

 // internal clock: next sample
```

```
 delay(100);        // delay time
}
```

Nach übersetzen und laden auf den Prozessor lässt sich mit der aufgebauten Schaltung mit Hilfe des Potentiometers die LED dimmen, die Ausgabe auf dem seriellen Monitor zeigt im vorgegebenen Takt die ausgelesenen Werte gemäß der Stellung des Potentiometers. Somit wären die externen Komponenten betriebsbereit.

Für den Kern des Programms ergibt sich mit den in der Aufzählung eingangs genannten Funktionen beispielsweise folgender Programmtext. Hierbei ist die Funktionstabelle mit 8 Bit Auflösung definiert, sowie mit 512 Stützstellen. Für den Test der Funktion genügt es, das Phaseninkrement in der Set-Up Routine als einstellige Zahl abzufragen. Diese Abfrage wird später durch die Abfrage des Potentiometers ersetzt.

```
// DDS simple case: output to LED, internal clock
// step 2: internal functions

int PIR;                    //phase increment
byte LUT[512];              //look up table (8 bit resolution)
static double pi = 3.14159; //constant pi
int i;                      //counter for LUT

// setup routine
void setup() {
  // initialize serial communication for tests
  Serial.begin(9600);

  // ask user to enter phase increment
  Serial.println("Pls. enter phase increment: ");
  while (Serial.available() == 0){};
  PIR = Serial.read() - 48;
  Serial.println(PIR);

  //set up function table (LUT)
  for (int i = 0; i < 512; i++) {
    LUT[i] = byte(127 * cos(2*pi*i/512.) + 128);
  }
}

// program loop
void loop() {

 // internal clock: next sample
 delay(100);        // delay time

 //calculate next step
 i = (i + PIR) % 512; //modulo 512 (rolls over i in LUT address range)

 // write values on serial monitor for test
```

```
    Serial.print(i);
    Serial.print(" : ");
    Serial.println(LUT[i]);
  }
```

Zur Abfrage der Funktionstabelle wird der Zählindex i in jeder Programmschleife um das Phaseninkrement erhöht. Damit der Zähler nach Überschreitung des Tabellenendes nach der 512-ten Stützstelle wieder in die Tabelle hinein läuft, erfolgt eine Modulo-Division durch 512. Die Funktionsweise der Zählschleife entspricht nun einem 9-Bit Zähler (0 bis 511).

Die folgende Abbildung zeigt einen Testlauf der internen Funktionen des Signalgenerators mit einem Phaseninkrement von 8. Nach Stützstelle 504 springt die Abfrage statt nach 512 wiederum an den Anfang der Tabelle, wodurch sich eine kontinuierliche periodische Funktion ergibt.

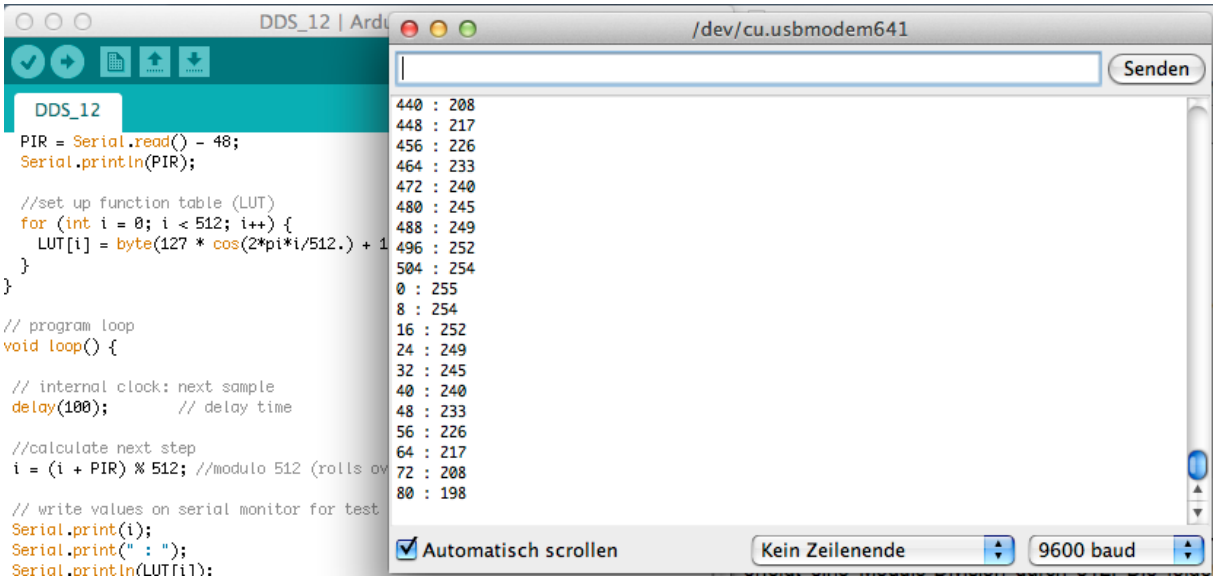


Bild 7.5 Testausgabe des Innenlebens mit Phaseninkrement 8

Nachdem sowohl die Eingänge und Ausgänge im ersten Schritt auf ihre Funktion getestet wurden, sowie die innere Architektur im zweiten Schritt, können beide Teile zu einem kompletten Signalgenerator zusammengesetzt werden. Hierzu lassen sich beide Programmteile zu einem neuen Programm zusammen kopieren. Als Eingang erfolgt nun die Abfrage des Potentiometers für das Phaseninkrement in der Hauptschleife. Hierbei der Wertebereich von 0 bis 1023 durch eine Verschiebeoperation um 2 binäre Stellen nach rechts auf einen sinnvollen Bereich von 0 bis 255 reduziert (wobei das Inkrement 0 ein konstantes Signal erzeugt).

Zunächst kann man die Testausgabe der Werte für PIR, i und LUT[i] noch im Programm belassen und die korrekte Funktion nochmals mit Hilfe des seriellen Monitors überprüfen. In folgendem Programmbeispiel wurden alle seriellen Testeingaben und Ausgaben beseitigt. Das Ausführen des Programms zeigt, dass sich mit Hilfe des Potentiometers ein periodisches harmonische Signal variabler Frequenz an der LED erzeugen lässt.

```
// DDS simple case: output to LED, internal clock
// integrate steps 1 and 2, eliminate serial test I/O
```

```
// input and output signals
int PIR;             // phase increment to be read from A0
int ledPin = 10;     // LED connected to digital pin 10

// internal architecture
byte LUT[512];              //look up table (8 bit resolution)
static double pi = 3.14159; //constant pi
int i;                      //counter for LUT

// setup routine
void setup() {
  //set up function table (LUT)
  for (int i = 0; i < 512; i++) {
    LUT[i] = byte(127 * cos(2*pi*i/512.) + 128);
  }
}

// program loop
void loop() {

 // internal clock: next sample
 delay(100);        // delay time in ms

 // read the input on analog pin 0 as phase increment:
 PIR = (analogRead(A0) >> 2); //scale values to (0-1023)/4

 // calculate next step
 i = (i + PIR) % 512; //modulo 512 (rolls over i in LUT address range)

 // write LUT value to LED via PWM
 analogWrite(ledPin, LUT[i]);  //scale to 0-255 for analogWrite

}
```

Der Frequenzbereich bewegt sich hierbei gemäß der Berechnung am Anfang des Abschnitts zwischen ca. 0,02 Hz und ca 5 Hz. Eine Frequenz von 0,02 Hz entspricht einer Periodendauer von 50 Sekunden. Das zugehörige Phaseninkrement von 1 lässt sich 1 nur mit Hilfe des seriellen Monitors feinfühlig genug einstellen. Mit eingeschaltetem Monitor zeigt sich auch, dass sich bei grossem Phaseninkrement, wie z.B. 255, eine Schwebung ergibt, da sich die Stützstellen im Intervall von 255 langsam in der Periode von den maximalen Amplituden in Bereiche kleinerer Amplituden verschieben. Dieser Effekt ist konstruktionsbedingt durch das DDS Verfahren.

Übung 7.1: Ergänzen Sie den Funktionsgenerator um eine oder mehrere weitere Signalformen, z.B. Dreieck, Sägezahn oder sonstige. Testen Sie die Funktion der Schaltung.

7.5. Tests und Weiterentwicklung

Für die Weiterentwicklung des Aufbaus wären folgende Themen interessant:

- Erweiterung für Signale im Audiobereich (bis 20 kHz): Statt der LED wäre nun ein Audio-Eingang zu betreiben. Allerdings müsste die PWM des seriellen D/A-Wandlers hierzu weit über die Grenzfrequenz reichen.
- Erweiterung für Signale im Audiobereich (bis 20 kHz): Als Alternative liessen sich die digitalen Ausgänge in Kombination mit einem externen D/A-Wandler betreiben.
- Statt der internen Timers wäre es interessant, wenn sich die Schaltung auf ein externes Taktsignal mit Hilfe von Interrupts synchronisieren liesse.

Die Frequenz der PWM lässt sich am Oszilloskop an Pin D10 messen. Die folgende Abbildung zeigt das Messergebnis, wobei hier ein PC-basierter Messadapter über USB mit passender Software verwendet wurde (siehe [7] im Literaturverzeichnis). Der Abstand zwischen zwei ansteigenden Flanken des pulsweitenmodulierten Signals beträgt ca. 2 ms, entsprechend einer Frequenz vom ca. 500 Hz. Hiermit bleibt der Audiobereich unerreichbar.

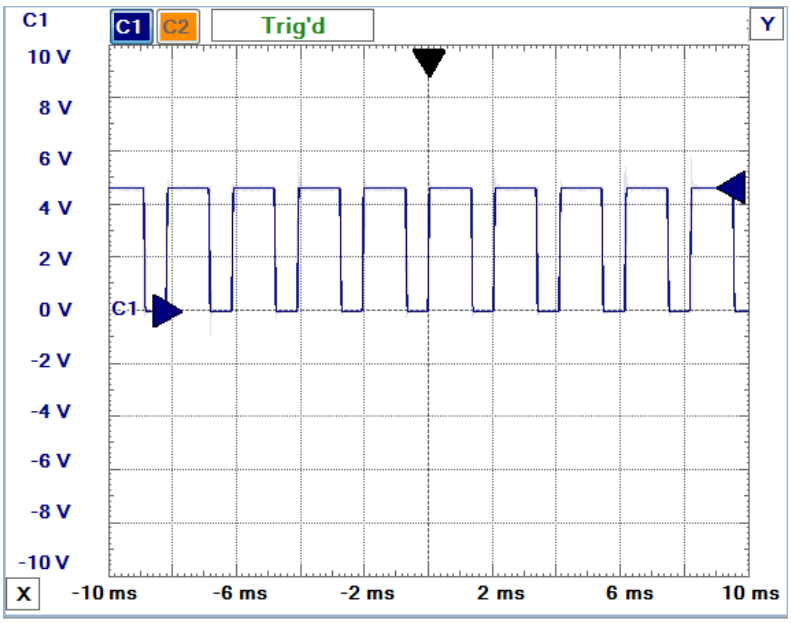


Bild 7.6 Messung am Ausgang D10

Pulsweitenmodulation durch den Prozessor zur D/A-Wandlung

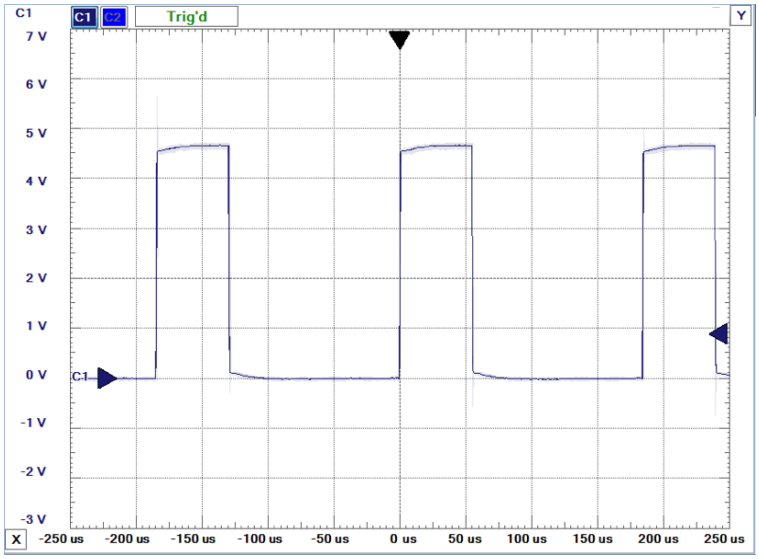
Für Anwendungen im Audio-Bereich scheidet dieses Konzept aus, wie eine überschlägige Berechnung zeigt. Um nur einen MP3-kompatiblen Frequenzbereich bis 10 kHz abdecken zu können, müsste die Taktrate der PWM für einen 10-Bit Wandler (=1024 Stufen Auflösung) bei ca. 10 MHz liegen, bzw. bei einer Auflösung von 8-Bit (=256 Stufen) bei ca. 2.5 MHz. Bei einer Taktfrequenz des Prozessors von 16 MHz ist dieses Konzept völlig unrealistisch.

Auf der anderen Seite ist es eh unwirtschaftlich, den Prozessor mit der D/A Wandlung (PWM) komplett auszulasten. Es gibt Varianten des Prozessors mit integrierten D/A Wandlern, mit denen Audio-Anwendungen mit der DDS Methode durchaus machbar sind. Die beim Arduino Uno verwendete Prozessorvariante besitzt zwar integrierte 10-Bit A/D Wandler, leider aber keine D/A Wandler.

Betrieb mit externen D/A-Wandlern

Beim Anschluss eines parallel arbeitenden D/A Wandler werden die Ausgänge als digitale Ausgänge im Takt der Programmschleife ausgegeben. Für die 8-Bit Auflösung der Funktionstabelle wären also z.B. 8 digitale Ports als digitale Ausgänge zu schalten. Für Audio-Anwendungen sollte die Taktrate bei ca. 10 kHz liegen, entsprechend 100 µs für eine Periode mit zwei Abtastwerten. Somit sollte die Programmschleife im Takt von 50 µs arbeiten. Das wäre zu überprüfen.

Für eine Messung der Latenz wurde zunächst die LED als Messpunkt im Programm gelassen. Nach Beschreiben der LED wurde ein Intervall von 50 us gewartet und anschliessend die LED zurückgesetzt. Folgende Abbildung zeigt die Messung an Pin D10 zusammen mit der verwendeten Programmschleife.

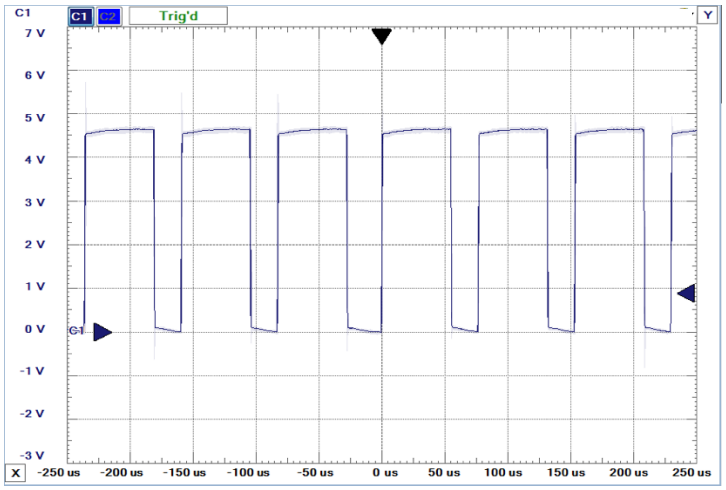



```
// program loop
void loop() {

 // read pin 0 as phase increment:
 PIR = (analogRead(A0) >> 2); //scale

 //calculate next step
 i = (i + PIR) % 512; //modulo 512

 // write LUT value to LED
  digitalWrite(ledPin, HIGH);
  delayMicroseconds(50);
  digitalWrite(ledPin, LOW);
}
```

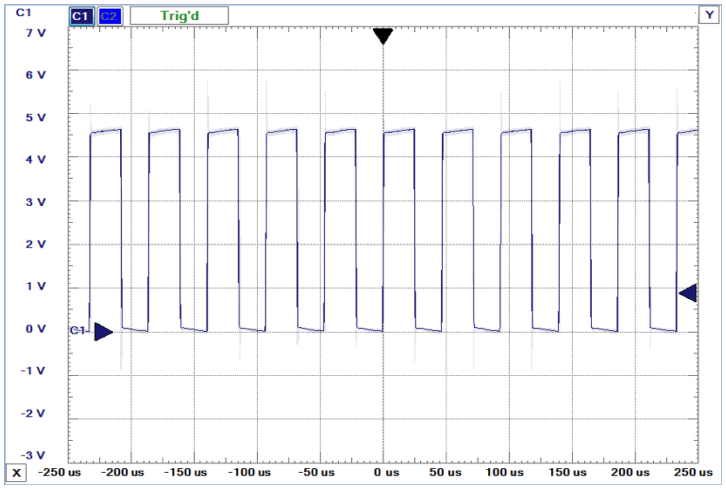
Bild 7.7 Messung der Latenz

Für das Auslesen des Potentiometers, die Modulo-Operation und das Beschreiben der LEDs werden insgesamt ca. 150 µs benötigt. Ursache für diese Verzögerung ist das Auslesen des Potentiometers. Verlagert man diese Funktion in die Set-up Routine, reduziert sich die Rechenzeit auf ca. 30 µs. Reduziert man das Verzögerungsintervall auf ca. 30 µs, so ist die gewünschte Taktung im Intervall von ca. 50 µs pro ausgegebenem Wert erreichbar, wie die in der folgenden Abbildung wiedergegebene Messung zeigt. Die Programmschleife mit den verwendeten Parametern ist jeweils rechts neben dem Oszillogramm dargestellt.

Die Verlagerung der Abfrage des Phaseninkrements in die Set-up Routine hat zur Folge, dass sich die Frequenz nur nach einem Reset der Schaltung (an der Reset-Taste des Arduino) ändern lässt. Hier wäre eine komfortablere Lösung gefragt.




```
// program loop
void loop() {

 //calculate next step
 i = (i + PIR) % 512; //modulo 512
 test = LUT[i];

 // write LUT value to LED
  digitalWrite(ledPin, HIGH);
  delayMicroseconds(50);
  digitalWrite(ledPin, LOW);
}
```




```
// program loop
void loop() {

 //calculate next step
 i = (i + PIR) % 512; //modulo 512
 test = LUT[i];

 // write LUT value to LED
  digitalWrite(ledPin, HIGH);
  delayMicroseconds(20);
  digitalWrite(ledPin, LOW);
}
```

Bild 7.8 Latenz nach Elimination des Lesens vom analogen Eingang

Ein 6-Bit D/A Wandler lässt sich mit Hilfe einer Widerstandskette aufbauen, wie in der nachfolgenden Abbildung gezeigt. Wegen der Bauteiletoleranzen erscheinen 6-Bit Auflösung ausreichend. Das Funktionsprinzip ist neben dem Schaltplan dargestellt. Mit Hilfe der Spannungsteilerregel lässt sich die Funktion des Wandlers in Abhängigkeit vom Schaltzustand der Bits nachvollziehen. Die Anwendung der Regel setzt voraus, dass der Ausgang nicht belastet wird. Damit der Ausgang hochohmig bleibt, ist ein Impedanzwandler am Ausgang vorzusehen.

Für die erzielbare Genauigkeit ist natürlich die Einhaltung des Verhältnisses R zu 2R wichtig. Hier kann man sich für die Realisierung damit behelfen, dass man nur Widerstände der Größe 2R verwendet. Für R werden dann 2 dieser Widerstände mit Wert 2R parallel geschaltet. Es bleiben dann allerdings noch die Effekte der Toleranzen der Bauteile.

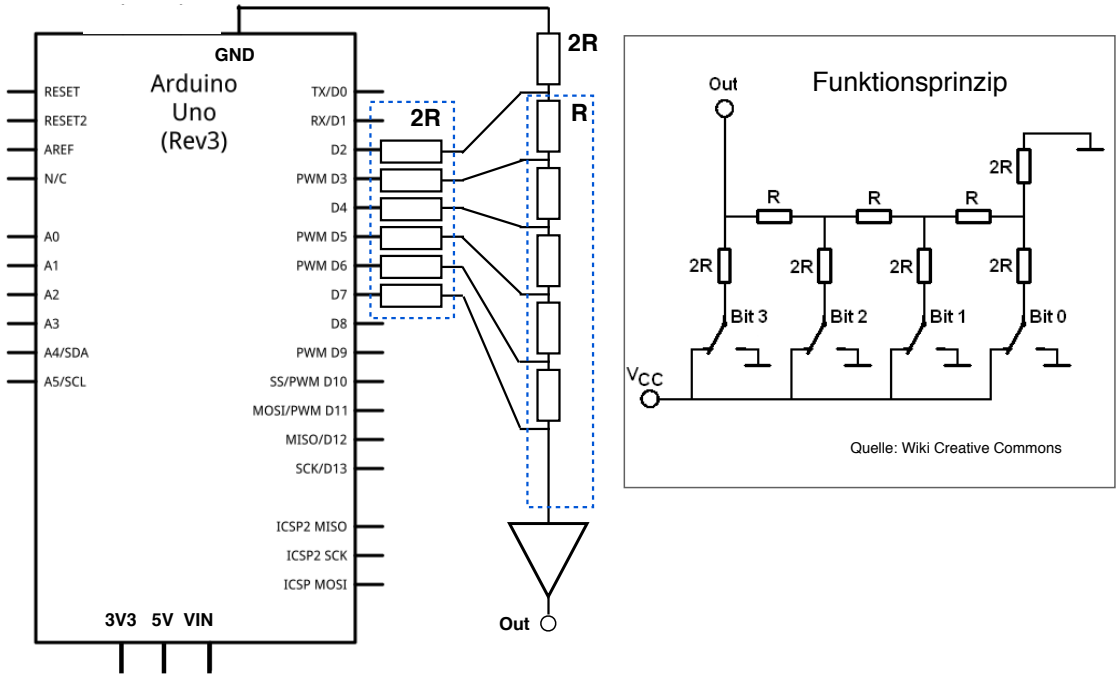


Bild 7.9 Eigenbau R-2R D/A Wandler

Für den Betrieb mit einem parallelen D/A Wandler sollte jedes Byte der Funktionstabelle 6 digitalen Ausgänge zugeordnet werden. Da in der gewählten Schaltung nur die oberen 6 Bits verwendet werden (Port D, Bits 2-7), kann der Wertebereich der LUT also bei 0 - 256 bleiben. Im folgenden Programmtext werden in der Set-Up Routine Bits 2-7 als Ausgänge konfiguriert. Statt wiederholter digitalWrites() auf jeden der Anschlusspins wurde in der Programmschleife das Portregister D direkt mit einem Byte beschrieben (siehe PORTD = LUT[i]).

```
// DDS with parallel DAC
// port map see http://www.arduino.cc/en/Reference/PortManipulation

// input and output signals
int PIR;             // phase increment to be read from A0

// internal architecture
byte LUT[512];              //look up table (8 bit resolution)
static double pi = 3.14159; //constant pi
int i;                      //counter for LUT
byte test;

// setup routine
void setup() {

  // set digital port to output for DAC
  DDRD = DDRD | B11111100;  // sets pins 7 to 2 as outputs

  //set up function table (LUT)
  for (int i = 0; i < 512; i++) {
```

```
      LUT[i] = byte(127 * cos(2*pi*i/512.) + 128);
    }

   // read pin 0 as phase increment:
   PIR = (analogRead(A0) >> 2); //scale to (0-1023)/4
  }

  // program loop
  void loop() {
   //calculate next step
   i = (i + PIR) % 512; //modulo 512

   // write value to D/A ports
    PORTD = LUT[i];
    delayMicroseconds(20);
  }
```

Zum Testen wurde ein Logik Analysator an die Pins 2-7 von Port D angesteckt (PC basiertes System, siehe [7]). Pin 7 kennzeichnet hierbei das höherwertigste Bit und findet sich in Kanal 5 der folgenden Abbildung. In drei Durchgängen wurde die Stellung des Potentiometers jeweils verändert, wobei nicht der gesamte Dynamikbereich ausgeschöpft wurde. Nach einem Reset der Schaltung mit Hilfe des Tasters auf der Arduino-Baugruppe wird das veränderte Phaseninkrement aufgenommen.

Betrachtet man eine Periode des höherwertigsten Bits, so ergeben sich von oben nach unten Perioden von ca. 950 µs, 600 µs und 200 µs. Diesen Periodenlängen entsprechen Frequenzen von ca. 1 kHz, 1,7 kHz und 5 kHz. Mit einer Taktperiode von ca 80 us für zwei Werte (siehe Bild 7.8) und maximal 512 Stützstellen pro Periode sollte der Frequenzumfang des Generators ca. 25 Hz bis 12.5 kHz umfassen.

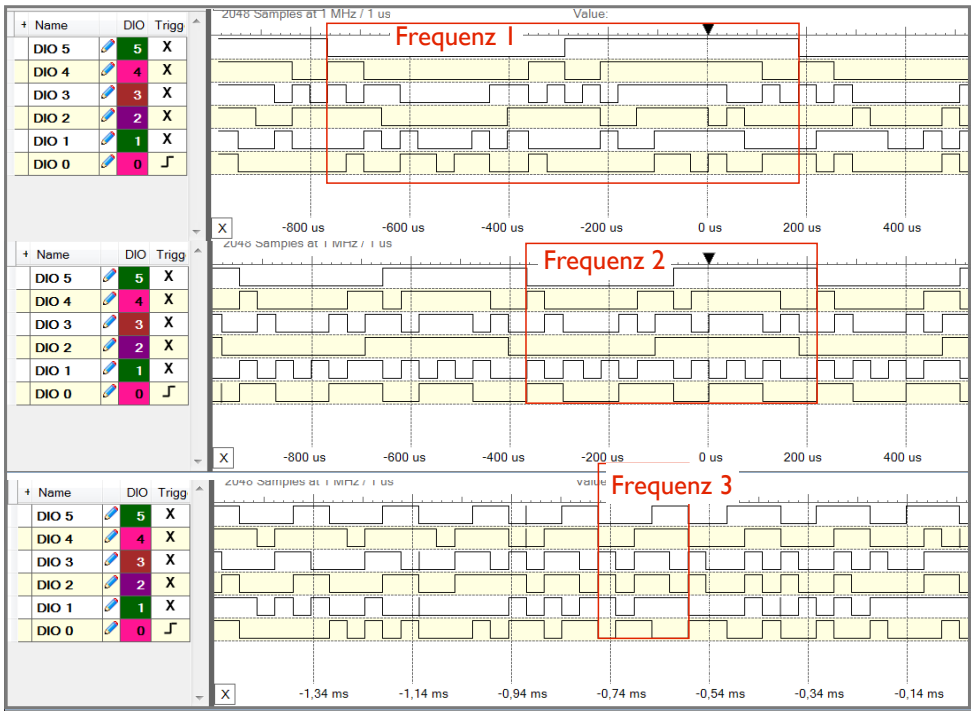


Bild 7.10 Test am Logik Analysator

Übung 7.2: Suchen Sie im Web nach externen D/A Wandlern für den Arduino. Wie werden diese an den Arduino angebunden? Erstellen Sie ein Konzept für den Betrieb eines solchen Wandlers, inklusive der Struktur der Software.

Übung 7.3: Suchen Sie nach Möglichkeiten, den Funktionsgenerator mit externen Signalen über Interrupts zu synchronisieren. Erstellen Sie hierzu ein Konzept und ein Muster der Software. Welche Vorteile hätte die Lösung gegenüber den bisher verwendeten internen Timern?

Übung 7.4: Bauen Sie eine Schaltung gemäß Übung 7.3 auf und nehmen Sie sie in Betrieb. Prüfen Sie die Funktion und messen Sie das Zeitverhalten.

8. Kommunikationsschnittstellen

In diesem Abschnitt wird das Konzept einiger weiterer in der Praxis relevanter Kommunikationsschnittstellen betrachtet, sowie deren Programmierung. Die serielle Schnittstelle stellt die Basis der meisten Kommunikationsschnittstellen aus Sicht des Anwendungsprogrammierers dar. Die serielle Schnittstelle bedient sich hierbei einer Abstraktion zum Beschreiben der Schnittstelle und zum Lesen von der Schnittstelle. Diese Abstraktion ist unabhängig von speziellen Medium nutzbar: ob die Schnittstelle über USB, GSM, WiFi oder ein anderes Medium geführt wird, macht hierbei keinen Unterschied. Viele Geräte bieten aus diesem Grund ebenfalls eine serielle Schnittstelle zur Kommunikation an.

Die serielle Schnittstelle mit den Methoden der Klasse Serial fand bereits in den vorausgegangenen Abschnitten hinreichend Verwendung. In diesem Anschnitt geht es um einige weitere Schnittstellen, bei denen das Verständnis der Funktion im Detail erforderlich ist. Oft bieten die Anbieter von Peripheriebausteinen Bibliotheken an, z.B. für die Arduino Entwicklungsumgebung, die die Programmierung auf einem höherem Abstraktionsniveau ermöglichen. Wenn keine Bibliothek zur Verfügung steht, muss man die Kommunikation ausgehend vom Datenblatt des Bausteins programmieren. Dann ist das Verständnis der Funktion dieser Schnittstellen hilfreich.

8.1. SPI - Serielle Schnittstelle für Peripherie

Das Serial Peripheral Interface (SPI) ist ein synchrones serielles Protokoll zur Kommunikation eines Mikrocontrollers mit peripheren Geräten. Mit serieller Kommunikation ist gemeint, dass die Daten nacheinander über eine gemeinsames Medium geschickt werden, im Unterschied zur parallelen Kommunikation. Synchrone Kommunikation bedeutet, dass Sender und Empfänger hierbei synchron arbeiten. In der Praxis wird hierzu eine Taktleitung verwendet. Folgende Abbildung zeigt das Prinzip der synchronen seriellen Kommunikation.

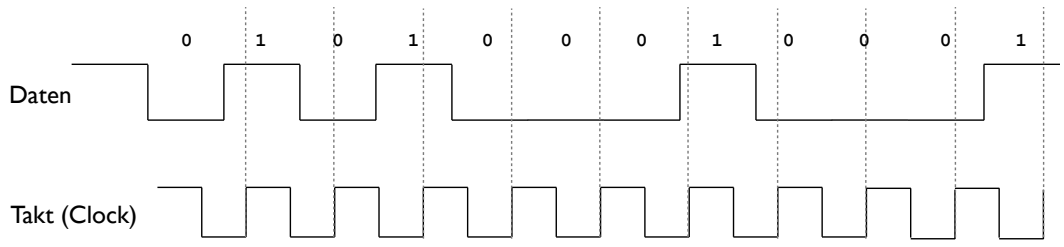
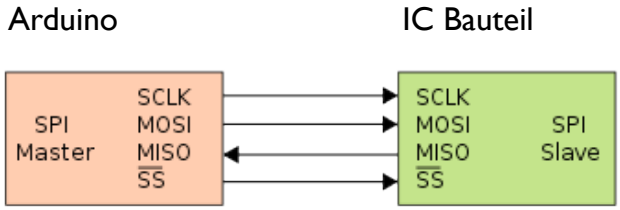


Bild 8.1 Synchrone serielle Kommunikation

Für die Daten ist eines der Systeme der Sender, das benachbarte System der Empfänger. Mit Hilfe des Taktes werden die Zeitpunkte definiert, an denen die Daten gültig sind und der Empfänger jeweils ein Datum übernehmen kann. Hierzu lassen sich z.B. die ansteigenden Flanken des Taktes verwenden, wie in der Abbildung gezeigt. Wer die Taktleitung betreibt, spielt für das Konzept keine Rolle. Allerdings muss es eine Stelle geben, die den Takt bereit stellt und die Kommunikation steuert.

Nach diesem Konzept werden also für eine bidirektionale Kommunikation zwischen Sender und Empfänger drei Leitungen benötigt: (1) die Taktleitung, (2) eine Leitung zum Senden, (3) eine Leitung zum Empfangen. Auf diese Weise miteinander verbundene Komponenten würden also Daten austauschen, solange ein Taktsignal gegeben wird. Gibt es mehr als zwei Kommunikationspartner, lässt sich der jeweils gewünschte Baustein durch ein weiteres signal auswählen. Bei SPI ist diese Auswahl durch eine weitere Leitung gelöst, wir in folgender Abbildung dargestellt.



Quelle: Wikimedia Commons

Bild 8.2 SPI als Kommunikationsschnittstelle

In der Abbildung zu sehen sind die bereits beschriebene Taktleitung (SCLK für SPI Clock), sowie die Datenleitungen in Senderichtung (MOSI für Master Out - Slave In), die Datenleitung in Empfangsrichtung (MISO für Master In - Slave Out). Die vierte Leitung ist mit SS bezeichnet für System Select (bzw. auch etwas neutraler als CS für Chip Select). Der Oberstrich auf der Bezeichnung ist als boolsche Negation zu interpretieren und bedeutet, dass das System durch den logischen Zustand LOW selektiert wird (als sogenannten active low Leitung).

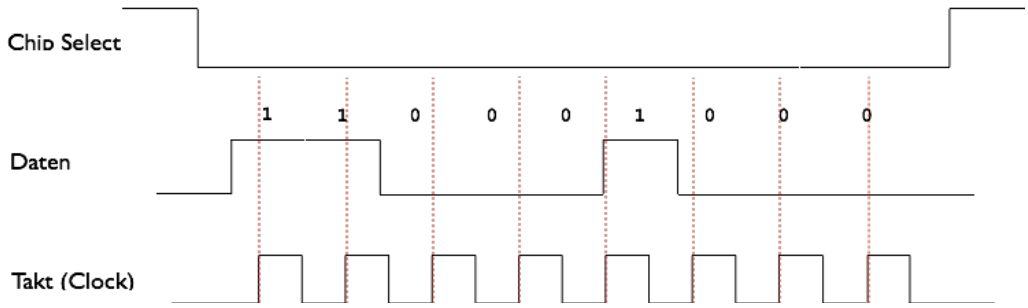
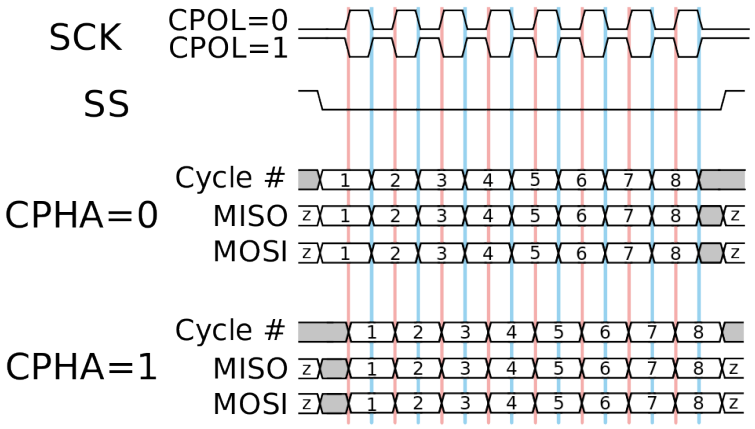


Bild 8.2 Ablauf der Kommunikation mit Hilfe der Chip Select Leitung

Eines der beiden Systeme muss die Taktleitung betreiben und die Kommunikation über die Chip Select Leitung organisieren. Diese Funktion ist als Master bezeichnet und ist in der Regel die Rolle des Mikrocontrollers. Die Bezeichnungen der beiden Datenleitungen beziehen sich auf dem Master der Kommunikation. Aus Sicht der Bauteile sind die Bezeichnungen jeweils DI (für Data In) oder SDI (für Serial Data In) bzw. DO (für Data Out) oder SDO (für Serial Data Out).

Die SPI-Spezifikation lässt Freiheitsgrade in der Interpretation des aktiven Taktsignals (Ruhewert bei LOW oder Ruhewert bei HIGH), sowie in der Wahl der aktiven Taktflanke (lesen der

Daten bei steigender oder bei fallender Flanke). Hierzu tauchen also folgende Begriffe im Zusammenhang mit der SPI-Schnittstelle auf: CPOL (Clock Polarity) und CPHA (Clock Phase). Die vier kombinatorischen Möglichkeiten stellen die möglichen Betriebsmodi der Schnittstelle dar. Die folgende Übersicht verdeutlicht die Begriffe und den Zusammenhang.



Quelle: Wikimedia Commons

Bild 8.2 Betriebsmodi der SPI-Schnittstelle

Weitere Protokolldefinitionen sind Sache des Herstellers von SPI Bausteinen. Hierzu gehört die Organisation der Daten für die Konfiguration des Bausteins (durch Setzen von Registern), die Datenformate für den Austausch von Informationen, die Organisation der Daten in Bytes, ggf. Quittungsnachrichten etc. Das generelle Vorgehen ist dabei immer gleich: (1) den Baustein konfigurieren, (2) Daten austauschen. Welche genauen Informationen in welchem Format benötigt werden findet sich im Datenblatt des Herstellers.

Die SPI Bibliothek

Die Entwicklungsumgebung des Arduino bietet eine Bibliothek für Bausteine, die über SPI angeschlossen werden. Die SPI Bibliothek enthält allgemein gültige Methoden für die Kommunikation über SPI. Zu diesen Methoden gehören folgende Anweisungen:

```
SPI.begin();           // initialisiert die SPI-Schnittstelle:
                    // SCK, MOSI, und CS als Ausgänge
                    // SCK auf low, CS und MOSI auf high

SPI.setBitOrder();     // setzt die Bit-Reihenfolge (msb/lsb zuerst)
                    // default: msb-lsb
SPI.setClockDivider(); // stellt die Übertragungsgeschwindigkeit ein
                    // default: 4 MHz (1/4 der Arduino Clock)
SPI.setDataMode();     // setzt den SPI-Modus (CPOL=0/1, CPHA=0/1
                    // default: Mode 0 (CPOL = 0, CPHA = 0)

SPI.transfer();    // überträgt ein Byte Daten (senden und empfangen)
```

Die Funktionen der Arduino SPI Bibliothek finden sich mit einigen Beispielen auf der Referenzseite [6] (unten unter Bibliotheken (library page), dann weiter unter SPI). Die Bedeutung und der Einsatz der verfügbaren Funktionen ist dort mit einigen Beispielen erklärt. Generell erfolgt die Verwen-

dung eines über SPI angebundenen Bausteins wie in folgendem Programmtext. Der Ablauf für den Datentransfer über die SPI-Schnittstelle in der Programmschleife lässt sich der Übersichtlichkeit halber auch in Unterprogramme auslagern.

```
#include <SPI.h>

/* pin assigment conventions (see http://arduino.cc/en/Reference/SPI):
 CS:   pin 10
 MOSI: pin 11
 MISO: pin 12
 SCLK: pin 13 */

// global variables
byte value;        // to be transferred over SPI

// set pin 10 as chip select
const int chipSelectPin = 10;

void setup() {
  // set the chipSelectPin as output
  pinMode (chipSelectPin, OUTPUT);

  // initialize SPI:
  SPI.begin();
}

void loop() {
...
  // take the SS pin LOW to select the chip
  digitalWrite(chipSelectPin, LOW);

  //  send the value via SPI
  SPI.transfer(value);  //write; read via value = SIP.transfer(0)

  // take the SS pin HIGH to de-select the chip
  digitalWrite(chipSelectPin, HIGH);
...
}
```

In aller Regel haben die verfügbaren Bausteine verschiedene Funktionen, die per Kommando eingestellt werden können, bzw. einen Adressbereich, an dessen Adressen Daten geschrieben werden können (oder von dessen AdressenDaten ausgelesen werden können). In diesem Fall sind zwei Datentransfers erforderlich. Folgender Programmtext ist ein Muster für ein Unterprogramm zum Schreiben von Daten, das in der Programmschleife aufgerufen werden kann.

```
void writeSpiChip(byte address, byte value){

  // take the CS pin LOW to select the chip
```

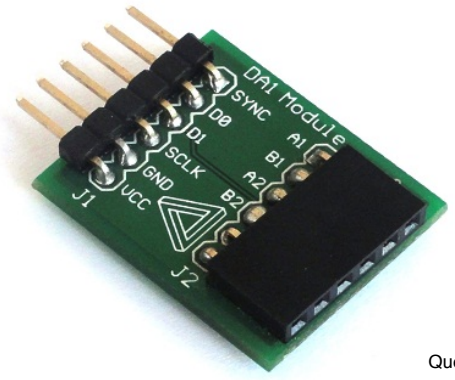
```
  digitalWrite(chipSelectPin, LOW);

  //  send the value via SPI
  SPI.transfer(address);      //send address or command byte
  SPI.transfer(value);        //send value

  // take the SS pin HIGH to de-select the chip
  digitalWrite(chipSelectPin, HIGH);
}
```

Das Lesen eines Wertes von einer bestimmten Adresse kann mit einem Unterprogramm erfolgen, wie in folgendem Programmtext beschrieben. Der Transfer der Adresse bzw. einer Anweisung an den Chip erfolgt wie im vorausgegangenen Beispiel. Beim Lesen wird ein Blind-Byte (Dummy Byte) gesendet, das der Chip ignoriert. Das genaue Vorgehen ist abhängig vom jeweiligen Chip. Informationen hierüber finden sich im Datenblatt des Herstellers.

```
byte readSpiChip(byte address){
  byte value;

  // take the CS pin LOW to select the chip
  digitalWrite(chipSelectPin, LOW);

  //  read the value via SPI
  SPI.transfer(address);              //send address or command byte
  value = SPI.transfer(0x00);         //read value

  // take the SS pin HIGH to de-select the chip
  digitalWrite(chipSelectPin, HIGH);

  return value;
}
```

Ein praktisches Beispiel

Als Beispiel sei ein D/A Wandler Modul gewählt, hier das PmodDA1 des Herstellers Digilent, das an der DHBW im Einsatz ist. Für eigene Experimente stellt Ihnen der Dozent gerne ein Exemplar zur Verfügung. Folgende Abbildung zeigt das Modul und das Blockschaltbild.



Quelle: Digilent Inc.

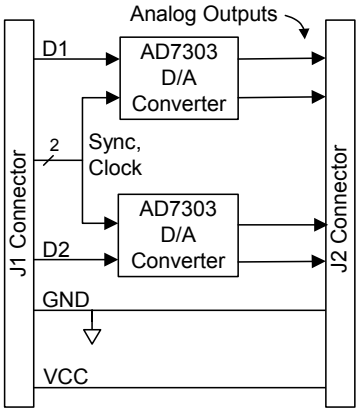
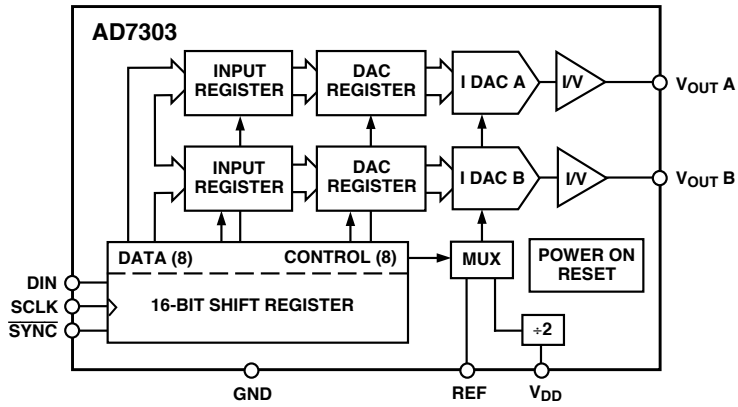


Bild 8.3 D/A Wandler Modul PmodDA1

Auf dem Modul finden sich zwei D/A Wandler Bausteine AD7303. Der Weg führt also weiter zum Datenblatt des Herstellers Analog Devices. Dort findet sich das Blockschaltbild des Bausteins und ein Zeitdiagramm der SPI-Schnittstelle, wie in folgender Abbildung gezeigt.



Quelle: Analog Devices

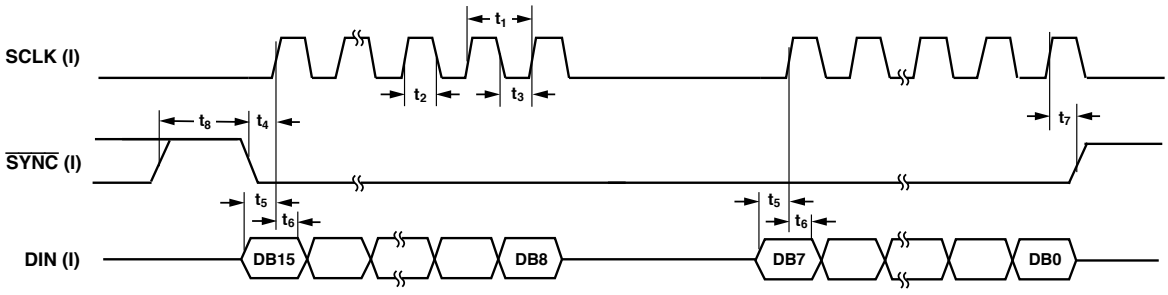


Bild 8.4 Blockschaltbild und Zeitdiagramm des D/A Wandler Bausteins

Man erkennt, dass der Baustein zwei Bytes an Daten erwartet: (1) ein Kommando (Control mit 8 Bit) zur Steuerung der Betriebsart, (2) Daten (mit ebenfalls 8 Bit). Das Schieberegister am Eingang des Chips ist dementsprechend 16 Bit lang. Das höchstwertigste Bit DB15 wird zuerst gesendet, d.h. die Reihenfolge ist MSB-LSB. Ausserdem erkennt man am Zeitdiagramm, dass der Ruhezustand der Clock 0 ist und die aktive Flanke die ansteigende Flanke (entsprechend CPHA = 0). Der SPI-Modus wäre demnach 0.

Auf dem Datenblatt finden sich auch die benötigten Angaben für die Konfiguration bzw. für die Ansteuerung der unterschiedlichen Betriebsarten des Bausteins. Ein Auszug aus dem Datenblatt findet sich in der Anlage. Mit dem Konfigurationsbyte 0x00 sollten demnach beide D/A Wandler parallel arbeiten. Für einen Test der Schnittstelle wird folgendes Programm verwendet.

```
// DAC connection over SPI
// Target: Digilent PmodDA1 DAC module

#include <SPI.h>

/* pin assigment conventions (see http://arduino.cc/en/Reference/SPI):
 CS:   pin 10
 MOSI: pin 11
 MISO: pin 12
 SCLK: pin 13 */

// set pin 10 as chip select
const int chipSelectPin = 10;

void setup() {
  // set the chipSelectPin as output
  pinMode (chipSelectPin, OUTPUT);

  // initialize SPI:
  SPI.begin();
}

void loop() {

  for (byte i = 0; i < 256; i++){

     // take the CS pin low to select the chip
     digitalWrite(chipSelectPin, LOW);

     //  send the value via SPI
     SPI.transfer(0x00);      //send command byte
     SPI.transfer(i);         //send value

      // take the SS pin high to de-select the chip
     digitalWrite(chipSelectPin, HIGH);

     delayMicroseconds(50); // 10 kHz sampling rate
   }

}
```

Im Programm wurde vorgegeben, dass alle 50 Mikrosekunden ein Abtastwert an den D/A Wandler gegeben werden soll. Mit zwei Abtastwerten pro Periode errechnet sich eine Periodenlänge

von 100 Mikrosekunden und eine Grenzfrequenz von 10 kHz. Als Signalform wurde mit Hilfe eines Zählers eine Rampe vorgegeben.

Nach Aufstecken des PmodDA1 Moduls auf dem Steckbrett erfolgt die Verdrahtung nach den Konventionen der Arduino SPI Umgebung, d.h. CS an Pin 10, MOSI an Pin 11 und SCLK an Pin 13. Da vom D/A Wandler nicht gelesen wird, wird MISO (Pin 12) nicht benötigt. Durch Verbindung von MOSI mit D0 oder D1 lässt sich einer der beiden AD7303 Bausteine auf dem Modul auswählen.

Zum Test der Funktion wurden die 3 Leitungen der SPI-Schnittstelle mit einem Logic Analysator überprüft. Folgende Abbildung oben zeigt das Ergebnis. Im oberen Diagramm beträgt das Raster der Zeitleiste 50 Mikrosekunden. Getriggert wurde auf die fallende Flanke des Chip Select (CS) Signals, das in der ersten Zeile des Diagramms aufgezeichnet wurde. Man erkennt, dass das CS Signal nur sehr grob dem vorgegebenen Intervall von 50 Mikrosekunden folgt, da die Programmschleife zusätzlich Rechenzeit benötigt. Dieses Intervall lässt sich durch Verringerung der Vorgabe for den internen Timer des Arduino reduzieren (delayMicroseconds(50)). Für eine wirklich genaue Abtastrate wäre aber ein externer Trigger vorzuziehen.

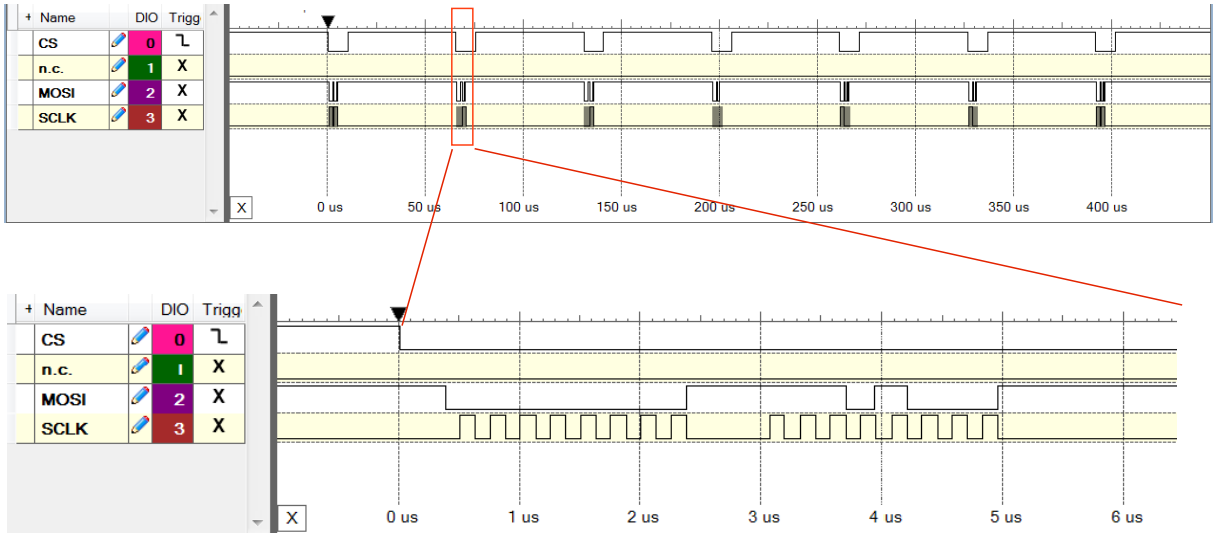


Bild 8.5 Ansteuerung des D/A Wandlers über SPI

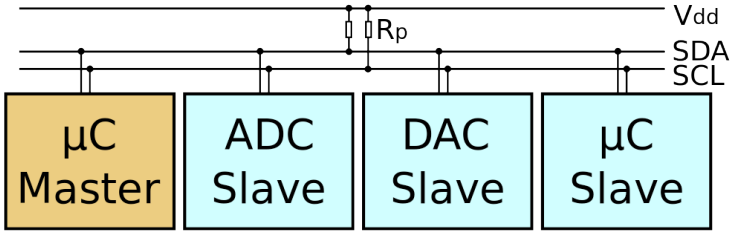
Auf Kanal 2 und 3 sind die Signale MOSI (serielle Daten zum D/A Wandler) und SCLK (SPI Clock) dargestellt. Durch Wahl eines kürzeren Zeitrasters von 1 Mikrosekunde pro Einheit erkennt man im unteren Diagramm die Kommunikation über die Schnittstelle. Nach der fallenden Flanke des Signals Chip Select beginnt der Datentransfer (MOSI). Die Clock liefert zunächst 8 Takte für das erste Byte, dann 8 weitere Takte für das zweite Byte.

Man erkennt ausserdem, dass die Übertragung eines Bytes etwa 2 Mikrosekunden dauert. Von der Schnittstelle her betrachtet wären somit also Datenraten von 500.000 Bytes/s möglich, ausreichend für eine Grenzfrequenz von 250 kHz. Die Dauer der Übertragung eines Bits (bzw. die Dauer einer Taktperiode) beträgt 1/4 Mikrosekunde. Die zugehörige Taktfrequenz liegt also bei 4 MHz, gemäss der vorgegebenen Einstellung der SPI Bibliothek. Für die serielle Schnittstelle stellt die Arduino CPU nur die Daten in einem Schieberegister des Arduino Schnittstellenbausteins bereit. Das Auslesen erfolgt dann unabhängig von der CPU. Der D/A Wandler auf dem externen Baustein kann mit Taktfrequenzen bis zu 30 MHz betrieben werden. Die hohe Geschwindigkeit serieller Schnittstellen erklärt, warum man heute auf parallele Ports verzichtet.

8.2. I²C Bus

Während SPI insgesamt 4 Leitungen für eine bidirektionale Übertragung benötigt (bzw. 3 Leitungen für eine unidirektionale Übertragung), ist kommt der I²C Bus mit insgesamt 2 Leitungen aus. Den Unterschied macht die Adressierung der angeschlossenen Geräte: SPI benötigt für jedes Gerät am Controller (Master) eine eigene Adressleitung (Chip Select). I²C arbeitet mit Geräteadressen und benötigt daher keine eigene Leitung zur Auswahl eines Gerätes. Ausserdem wird nur eine einzige Datenleitung für beide Richtungen verwendet.

Ein I²C System hat die in folgender Abbildung gezeigte Struktur. Wegen der gemeinsamen Datenleitung spricht man hier auch von einer Bus-Struktur. Da die Schnittstelle seriell arbeitet, ist die Funktion der beiden Busleitungen eindeutig: eine der Leitungen ist die Taktleitung (SCL - Serial Clock Line), die andere Leitung ist die Datenleitung (SDA - Serial DAta Line).

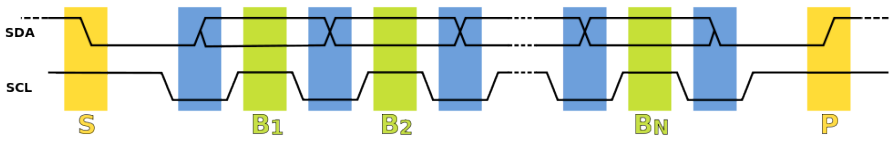


Quelle: Wikimedia Commons

Bild 8.6 I2C Zweitdraht Bussystem

Den Nachrichtenaustausch steuert beim Zweidrahtbus I²C ebenfalls ein Master, in aller Regel der Mikrocontroller. Elektrisch haben alle angeschlossenen Bauteile offene Kollektorausgänge und sind über die Pull-up Widerstände an beiden Leitungen auf definiertem Potential. Dieser Anschluss stellt eine verdrahtete ODER-Verknüpfung der Ausgänge dar und wird zur Signalisierung benutzt. Hohe Datenraten sind bei diesem Konzept nicht vorgesehen. Die Taktrate liegt üblicherweise bei maximal 100 kHz im Standardmodus, bzw. bei maximal 400 kHz im sogenannten Fast Mode.

Für die Kommunikation gibt der Master den Takt vor. Das Protokoll gibt vor, dass jeweils Einheiten von 8 Bits übertragen werden. Bei der moderaten Taktrate lassen sich Phasenübergänge der Signale auf beiden Leitungen zur Steuerung der Kommunikation nutzen. Ein Startsignal gibt der Master durch eine fallende Flanke auf der Datenleitung, währen die Taktleitung noch im Ruhezustand ist (Ruhezustand = HIGH). Als Stopp-Signal zieht der Master die Datenleitung auf HIGH, nachdem die Taktleitung bereits in den Ruhezustand übergegangen ist. Folgende Abbildung zeigt den Ablauf der Kommunikation.



Quelle: Wikimedia Commons

Bild 8.7 Kommunikation über den Zweidrahtbus I²C

Auffällig ist, dass Daten nicht bei fallenden oder steigenden Taktflanken interpretiert werden, sondern im Zustand HIGH der Taktleitung. Während dieses Zustandes dürfen sich die Daten auf der Datenleitung nicht ändern. Nach 8 Datenbits wird ein weiteres Protokollbit übertragen, das sogenannte Bestätigungsbit (Acknowledge Bit), das den Empfänger der Nachricht (vorausgegangene Bits) zu

einer Quittierung des Empfangs motivieren soll. Wurde die Nachricht korrekt empfangen, legt der Empfänger die Datenleitung auf LOW und hält die Datenleitung in diesem Zustand während der folgenden HIGH Phase der Taktleitung. Die Taktleitung verbleibt in diesem Zustand (Ruhezustand). Der Übergang der Datenleitung in den Zustand LOW muss vor dem Übergang der Taktleitung in den Zustand HIGH erfolgen, um Missverständnisse zu vermeiden. Der Master hebt im Anschluss an die Interpretation der Quittung die Datenleitung auf HIGH und meldet hiermit das Stopp-Signal.

Wie werden nun Geräte adressiert? I²C verwendet üblicherweise 7-Bit Adressen, die im Byte Format als Nachricht verschickt werden. Das achte Bit kennzeichnet, ob die folgende Nachricht vom gewünschten Empfänger gelesen werden soll, oder ob der Empfänger eine Nachricht an den Master senden soll. Für Geräte, die entweder nur lesen oder nur schreiben können, spielt diese Unterscheidung allerdings keine Rolle. Die Geräteadressen werden vom Hersteller festgelegt, wobei einige Adressbits vom Anwender durch Verdrahtung konfiguriert werden können, z.B. um mehrere baugleiche Geräte zu betreiben.

Die Wire Bibliothek

In der Arduino Entwicklungsumgebung findet sich die I2C Bibliothek unter dem Stichwort Wire. Die Bibliothek ist eng mit der seriellen Schnittstelle verwand, daher dürften die meisten Methoden geläufig sein. Serielle Schnittstellen verwenden allerdings üblicherweise nur Punkt-zu-Punkt Verbindungen, man beschreibt immer das Gerät auf der anderen Seite der Verbindung (bzw. liest von diesem Gerät). Für ein serielles Bussystem kommt nun die Adressierung mehrerer Geräte dazu. Die Bibliothek stellt folgende Funktionen für einen Busmaster bereit.

```
Wire.begin();              // initialisiert die I2C-Schnittstelle
                     // default: als Master teilnehmen

// Daten von einem Gerät lesen:
Wire.requestFrom();        // Daten von einem Gerät anfordern
                     // Wire.requestFrom(address, quantity)

Wire.available();          // Weitere Nachrichten zum Lesen verfügbar
                     // Bsp. while (Wire.available()) {
                                char c = Wire.read()};

Wire.read();               // Lesen einer Nachricht vom Gerät
                     // Bsp. char c = Wire.read();


//Daten auf ein Gerät schreiben:
Wire.beginTransmission();        // Gerät adressieren
                          // Wire.beginTransmission(address);

Wire.write();                    // Nachricht an das Gerät schicken
                          // Bsp. write(value);

Wire.endTransmission();          // Übertragung beenden
```

In der Bibliothek finden sich einige weitere Funktionen, die von Geräten (Slaves) genutzt werden können. Muster für alle Anwendungen finden sich in der Bibliothek ebenfalls, darunter einen schreibenden Master (master_writer) und einen lesenden Master (master_reader), die als Vorlagen für eigene Programme verwendet werden können. Auf der Arduino Baugruppe findet sich die I²C Schnittstelle an den Pins der analogen Eingänge A4 (SDA) und A5 (SCL).

8.3. Ethernet und IP

Ethernet findet sich in lokales Netzen. Das Internet mit dem Internet Protokoll (IP) umspannt die ganze Welt. Unter einem Netz (engl. network) versteht man einen gemeinsamen Adressraum, den alle Teilnehmer des Netzes nutzen. Ein Beispiel wäre das Postnetz, dessen Adressen sich aus Postleitzahlen, Strassen und Hausnummern zusammensetzen. Ein anderes Beispiel wäre das Telefonnetz, wobei die Telefonnummern die Netzadressen darstellen.

Die Netzadressen dienen der Zustellung von Nachrichten. Bei grossen Netzen ist der Adressraum hierarchisch organisiert. Ein Beispiel wären die Postleitzahlen: Um von Stuttgart aus eine Sendung an die Postleitzahl 24937 oder 87561 zuzustellen, genügt von Stuttgart aus ein Blick auf die erste Ziffer. Die Sendung mit der 2 am Anfang muss weiter in nördliche Richtung, die mit der 8 am Anfang Richtung Süden. So kann eine Sendung effizient vom Groben ins Feine verarbeitet werden. Strasse und Hausnummer werden schliesslich als lokale Adressen vor Ort genutzt.

Wären die Postleitzahlen zufällig unter den Orten verteilt worden, müsste man in jedem Postamt bei jeder Sendung durch den gesamten Katalog blättern, um die nächste Station für die Zustellung zu ermitteln und die Sendung weiter zu transportieren. Netze mit Adressräumen finden sich auch am Arbeitsplatz bzw. zuhause, wie in folgender Abbildung gezeigt.

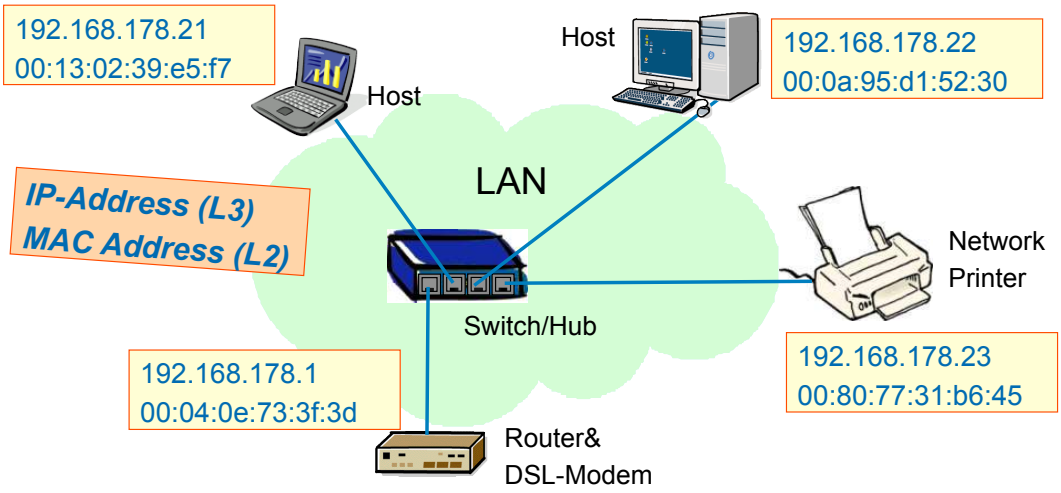


Bild 8.8 Ethernet und IP im Heimnetz

Hier sind einige Geräte zu einem Netz zusammen geschaltet, darunter ein Netzwerkdrucker, ein Laptop Computer und ein fest installierter Rechner. Ob für die Zusammenschaltung kabelgebundenes Ethernet oder WiFi verwendet wird, spielt aus Sicht des Netzes keine Rolle. Die untere Box stellt die Verbindung zum Internet her (als Router mit DSL Modem bzw. Kabelmodem bezeichnet).

Ethernet verwendet Geräteadressen als Netzwerkadresse. Die Geräteadresse wird vom Hersteller fest an jedes Gerät vergeben. Die Adressen hierbei so organisiert, dass keine Konflikte

durch mehrfach vorhandene Adressen auftreten. Der Adressraum im lokalen Netz ist also flach: ein Netzknoten (Ethernet Switch) muss jedes einzelne angeschlossene Gerät kennen, damit es Nachrichten an den korrekten Port zustellen kann. Da das Netz als lokales Netz betrieben wird, ist das weiter kein Problem. Ethernet Adressen (MAC Adressen) sind 48 Bit lang und werden üblicherweise in hexadezimaler Schreibweise wiedergegeben, wie in der Abbildung zu sehen.

Die IP Adressen in der Abbildung sind ebenfalls lokale Adressen. Diese Adressen werden von einem Adressserver vergeben (dem sogenannten DHCP-Server, der sich z.B. auf der Anschlussbox zum Internet befindet). Sobald ein neues Gerät ins Netz gebracht wird, erhält es eine lokale IP-Adresse. Die Anschlussbox hat ausser ihrer eigenen lokalen IP Adresse eine im Internet gültige öffentliche Adresse. Diese Adresse erhält Sie vom Internet Anbieter. IP-Adressen sind 32 Bit lang und werden üblicherweise im dezimalen Format dargestellt, wie ebenfalls in der Abbildung zu sehen. Ausser den 32 Bit breiten IP Adressen (den sogenannten IPv4 Adressen, wobei v4 auf die 4. Version des Protokolls hindeutet), sind inzwischen auch 128 Bit breite IP Adressen im Einsatz (die sogenannten IPv6 Adressen).

Wenn ein Gerät wie ein Arduino im Netz teilnehmen möchte, benötigt er sowohl eine MAC Adresse als auch eine IP Adresse. Die MAC Adresse hat der Hersteller des Ethernet Adapters (Ethernet Shield) bereits vergeben. Die Zuweisung der IP Adresse geschieht entweder manuell durch den Anwendungsprogrammierer, oder automatisch mit Hilfe des Zuweisungsprotokolle (DHCP) durch den diesbezüglichen Server (DHCP Server der Anschlussbox).

Weiterhin wird eine Anwendung benötigt, die die anwendungsorientierten höheren Protokolle unterstützt: Für einen Web-Server bzw. einen Web-Client muss das Protokoll HTTP bedient werden. Ebenso gibt es Anwendungsprotokolle für E-Mail, Telefonie oder Datentransfer. Viele Anwendungen basieren inzwischen auf HTTP, daher ist speziell der Web-Server bzw. Web-Client von Interesse.

Die Begriffe Client und Server sind folgendermassen definiert: Der Client ist immer derjenige, der eine Anfrage startet. Der Server ist immer derjenige, der eine Anfrage bedient. Damit er Anfragen von Clients bedienen kann, muss der Server vorher bereit stehen und warten. In einem Restaurant wäre ein Gast ein Client (Kunde), der Kellner der Server (engl. waiter). Wenn Sie einen Telefonanruf entgegen nehmen, sind Sie der Server. Wenn Sie jemanden anrufen, sind Sie der Client.

Übung 8.1: Erkunden Sie die eigene Netzwerkumgebung. Wenn Sie einen Windows PC verwenden, öffnen Sie das Konsolenfenster (Eingabeaufforderung) und geben das Zeilenkommando netstat ein. Untersuchen Sie auch die Optionen netstat -an und netstat -ano. Analysieren Sie die Ergebnisse. Auf einem Mac öffnen Sie unter Programme den Ordner Dienstprogramme (utilities) und starten das Netzwerkdienstprogramm (network utility) mit der Funktion Netstat. Wenn Sie ein Linux System verwenden, kennen Sie das vermutlich bereits alles.

Die Ethernet Bibliothek

In der Arduino Bibliothek finden sich die Web-Anwendungen unter dem Stichwort Ethernet. Die Bibliothek unterstützt Klassen für:

- die Zuweisung von MAC und IP Adressen
- die Initialisierung der Netzwerkprotokolle
- einen Web-Server (Server Class)
- einen Web-Client (Client Class)
- einen IP basierten Telegrammdienst (EthernetUDP)

- Muster für verschiedene Web Anwendungen.

Ethernet Adapter werden über die SPI Schnittstelle an die Arduino Baugruppe angebunden. Daher wird im Programmtext auch die SPI Bibliothek eingebunden. Von SPI Funktionen sieht der Anwendungsprogrammierer allerdings wenig. Verwendet wird die SPI Bibliothek durch die Ethernet Bibliothek, die somit eine höhere Abstraktionsebene bereit stellt. Da die Ethernet Bibliothek recht gut dokumentiert ist, wird an dieser Stelle auf die diesbezügliche Dokumentation verwiesen (siehe [6]).

Messwerte auf einer Web-Seite anzeigen

Als Anwendungsbeispiel wird ein Web-Server realisiert, der einen Messwert zur Abfrage durch Web-Browser bereit stellt. Das Beispiel basiert auf der Musteranwendung WebServer aus der Arduino Bibliothek. Um die Musteranwendung zu starten bzw, eigene Anwendungen zu entwickeln, benötigen Sie ein Arduino Ethernet Shield oder ein vergleichbares Produkt.

Zur Teilnahme im lokalen Netz sind zwei Adressen nötig für MAC und IP. Wo bekommt man die MAC-Adresse her? Auf der Unterseite des Ethernet Shields befindet sich ein Aufkleber mit der MAC Adresse. Wo bekommt man eine IP-Adresse her? Hierzu gibt es zwei Möglichkeiten: (1) entweder im lokalen Netz nachprüfen, welche Adressen verwendet werden und eine freie Adresse manuell vergeben (Netstat aufrufen, siehe Übung 8.1), (2) eine IP Adresse vom DHCP Server automatisch beziehen. In folgendem Programmtext wurde die IP-Adresse zunächst manuell vergeben.

```
    // Simple Web Server
    // target: Arduino Ethernet Shield
    // target: temperature sensor TI LM35

    #include <SPI.h>
    #include <Ethernet.h>

    // pin assigment: Ethernet shield to SPI Pins 10, 11, 12 and 13

    // device specific MAC address and local IP address
    byte mac[] = { 0x90, 0xA2, 0xDA, 0x0E, 0xAE, 0xA3 };
    IPAddress ip(192,168,178,177);

    //global variables
    int sensorValue;    // value read from analog port A0
    double temp;        // temperature

    // HTTP Header and HTML document parts
    #define HTTP_RESPONSE "HTTP/1.1 200 OK \n Content-Type: text/html \n
Connection: close Refresh: 5 \n"
    #define HTML_TOP     "<!DOCTYPE HTML> \n <html> <head> <title> Arduino
Web-Server </title> </header> \n <body>"
    #define HTML_BOTTOM "</body> \n </html>"

    // Initialize Ethernet server at HTTP port 80
    EthernetServer myServer(80);
```

```
void setup() {

  //// use Serial interface for debugging (not mandatory)
  Serial.begin(9600);

  analogReference(INTERNAL); // use internal reference 1.1 V on ADC

  // start the Ethernet connection and the server:
  Ethernet.begin(mac, ip);
  myServer.begin();

  ////debugging output to serial
  Serial.print("server is at ");
  Serial.println(Ethernet.localIP());
}

void loop() {
  // read temperature
   sensorValue = analogRead(A0); // values in range 0-1023
   temp = (sensorValue * 100.0) / 1024;

  // wait for incoming clients
  EthernetClient client = myServer.available();
  if (client) {

    Serial.println("new client");

    // an HTTP request ends with a blank line
    boolean currentLineIsBlank = true;
    while (client.connected()) {
      if (client.available()) {
        char c = client.read();
        Serial.write(c);       //copy HTTP Request to serial monitor
        // read until end of line, which indicated by '\n'
        if (c == '\n' && currentLineIsBlank) {

          // send a standard http response header
          client.println(HTTP_RESPONSE);
          client.println(HTML_TOP);

          // read analog pin and write value to client
            sensorValue = analogRead(A0);
            temp = (sensorValue * 100.0) / 1024;
            client.print("the temperature is: ");
            client.print(temp, 1);
            client.print(" C");
            client.println("<br />");

          client.println(HTML_BOTTOM);
```

```
            break;
          }
          if (c == '\n') {
            // you're starting a new line
            currentLineIsBlank = true;
          }
          else if (c != '\r') {
            // you've got a character on the current line
            currentLineIsBlank = false;
          }
        }
      }
      // give the web browser time to receive the data
      delay(1);
      // close the connection:
      client.stop();
      Serial.println("client disonnected");
    }
  }
```

Übung 8.2: Analysieren Sie den Programmtext. Wie ist der grundsätzliche Ablauf? Welche Komponenten gehören zur Verarbeitung der HTTP-Anfrage (HTTP-Request)? Welche Komponenten gehören zur Beantwortung der Anfrage (HTTP-Response)?

Auf der folgenden Seite ist das Ergebnis eines Testlaufs dargestellt. Oben rechts neben dem Fenster der Arduino Entwicklungsumgebung findet sich das Fenster des seriellen Monitors. Nach der Statusmeldung des Servers mit seiner lokalen E-Mail Adresse findet sich hier der Text der Anfrage des Clients (HTTP-Request). Wie man sieht, gibt der Client sich zu erkennen mit Nennung des verwendeten Browser, dem verwendeten Rechner, sowie einigen Browser-Einstellungen.

Darunter findet sich das Browser-Fenster des Clients, in den die IP-Adresse des Servers eingegeben wurde. Der Web-Server auf dem Arduino meldet sich mit seinem Namen im Titel des Fensters und im Browser-Fenster mit der gemessenen Temperatur.

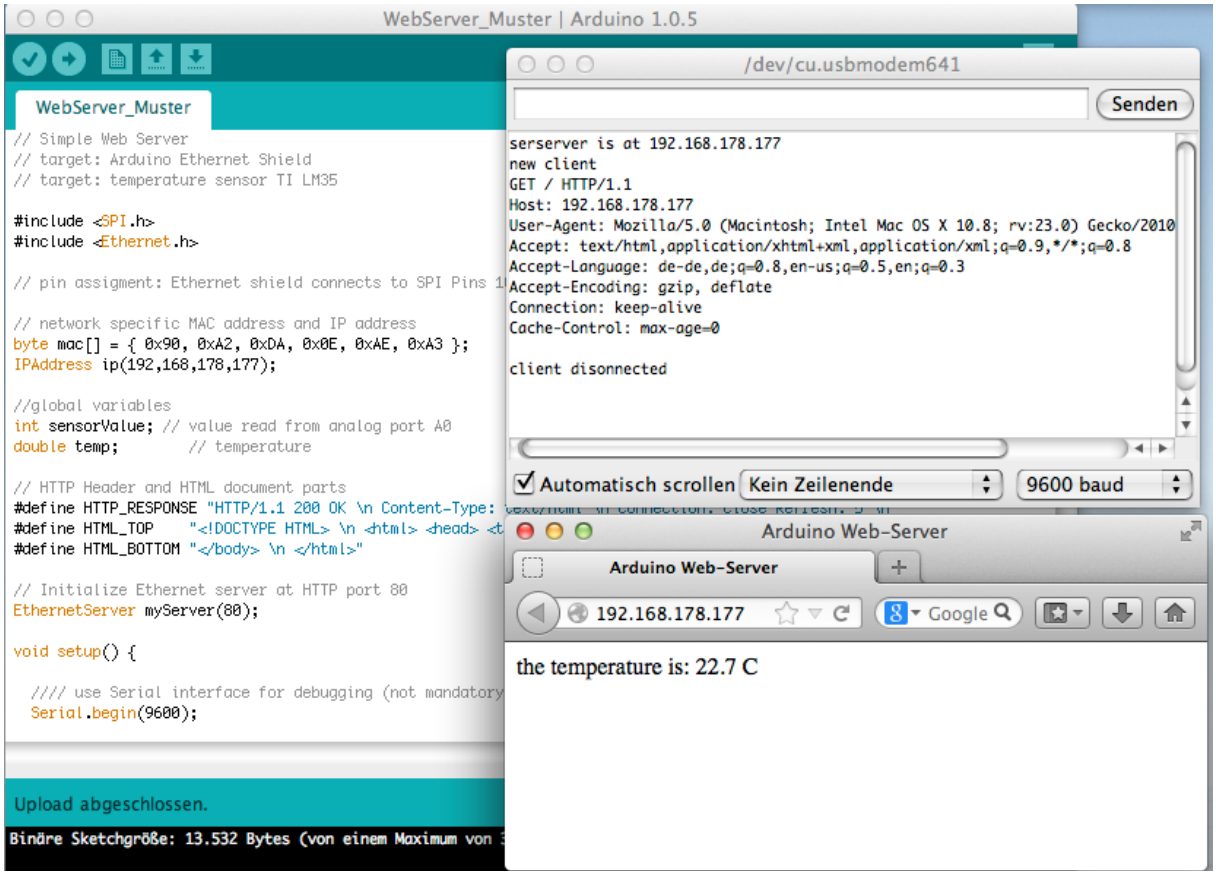


Bild 8.9 Arduino Web-Server

Der automatische Bezug einer lokalen IP-Adresse vom DHCP-Server funktioniert ebenfalls. Hierzu wird keine IP-Adresse manuell vergeben, auch der Blick in die Netzwerkumgebung mit Hilfe von Netstat kann entfallen. Ein Programmbeispiel für diese Art der Adressverteilung findet sich in der Arduino Ethernet Bibliothek unter der Methode Ethernet.localIP().

9. Übungen

Werden im Laufe des Semesters ergänzt.

9.1. Schrittmotor ansteuern

...

9.2. Regelung für Servomotor

...

9.3. Audio-Verarbeitung (Digitales Filter)

...

9.4. CAN Bus

...

Englisch - Deutsch

Activity Diagramm	Aktivitätsdiagramm
Class Diagram	Klassendiagramm
Event	Ereignis
Failed	fehlerhaft (bei Testfällen)
Interrupt	Alarmmeldung
Laptop	Klapprechner
Library	Bibliothek
Look up Table	Funktionstabelle
Low Pass Filter	Tiefpassfilter
Passed	bestanden (bei Testfällen)
Network	Netz
State Diagram	Zustandsdiagramm
State Event Table	Zustandsübergangstabelle

...

Abkürzungen

CPU	Central Processing Unit
DDS	Direkte Digitale Synthese
IDE	Integrated Development Environment
IP	Internet Protokoll
LUT	Look up Table
UML	Unified Modelling Langauge

...

Literatur

(1) Erik Bartmann, Die elektronische Welt mit Arduino entdecken, O'Reilly, 2011, ISBN-13: 978-3897213197

(2) Günter Schmitt, Mikrocomputertechnik mit Controllern der Atmel AVR-RISC-Familie: Programmierung in Assembler und C - Schaltungen und Anwendungen, Oldenbourg Wissenschaftsverlag, 2010, ISBN-13: 978-3486589887

(3) Arduino Programmierumgebung: zu laden unter http://www.arduino.cc

(4) Fritzing Editor für Steckbrett und Schaltpläne: zu laden unter http://fritzing.org

(5) Sprachreferenz der Arduino Programmierumgebung (Strukturen, Datentypen, Funktionen): http://arduino.cc/en/Reference/HomePage

(6) Digilent Analog Discovery Kit (Taschenlabor mit Messeingängen und Signalausgängen, lässt sich als Messplatz und zur Erzeugung von Testsignalen verwenden), siehe Trenz Electronic

Anhang A - D/A Wandler mit SPI Schnittstelle

Auszug aus dem Datenblatt, Quelle: Analog Devices

AD7303

DB15 (MSB)															DB0 (LSB)
$\overline{\text{INT}}$/EXT	X	LDAC	PDB	PBA	$\overline{\text{A}}$/B	CR1	CR0	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
Control Bits								Data Bits							

Figure 24. Input Shift Register Contents

Bit Location	Mnemonic	Description
DB15	$\overline{\text{INT}}$/EXT	Selects between internal and external reference.
DB14	X	Uncommitted bit.
DB13	LDAC	Load DAC bit for synchronous update of DAC outputs.
DB12	PDB	Power-down DAC B.
DB11	PDA	Power-down DAC A.
DB10	$\overline{\text{A}}$/B	Address bit to select either DAC A or DAC B.
DB9	CR1	Control Bit 1 used in conjunction with CR0 to implement the various data loading functions.
DB8	CR0	Control Bit 0 used in conjunction with CR1 to implement the various data loading functions.
DB7–DB0	Data	These bits contain the data used to update the output of the DACs. DB7 is the MSB and DB0 the LSB of the 8-bit data word.

CONTROL BITS

LDAC	$\overline{\text{A}}$/B	CR1	CR0	Function Implemented
0	X	0	0	Both DAC registers loaded from shift register.
0	0	0	1	Update DAC A input register from shift register.
0	1	0	1	Update DAC B input register from shift register.
0	0	1	0	Update DAC A DAC register from input register.
0	1	1	0	Update DAC B DAC register from input register.
0	0	1	1	Update DAC A DAC register from shift register.
0	1	1	1	Update DAC B DAC register from shift register.
1	0	X	X	Load DAC A input register from shift register and update both DAC A and DAC B DAC registers.
1	1	X	X	Load DAC B input register from shift register and update both DAC A and DAC B DAC registers outputs.

$\overline{\text{INT}}$/EXT	Function
0	Internal $V_{DD}/2$ reference selected.
1	External reference selected; this external reference is applied at the REF pin and ranges from 1 V to $V_{DD}/2$.

PDA	PDB	Function
0	0	Both DACs active.
0	1	DAC A active and DAC B in power-down mode.
1	0	DAC A in power-down mode and DAC B active.
1	1	Both DACs powered down.

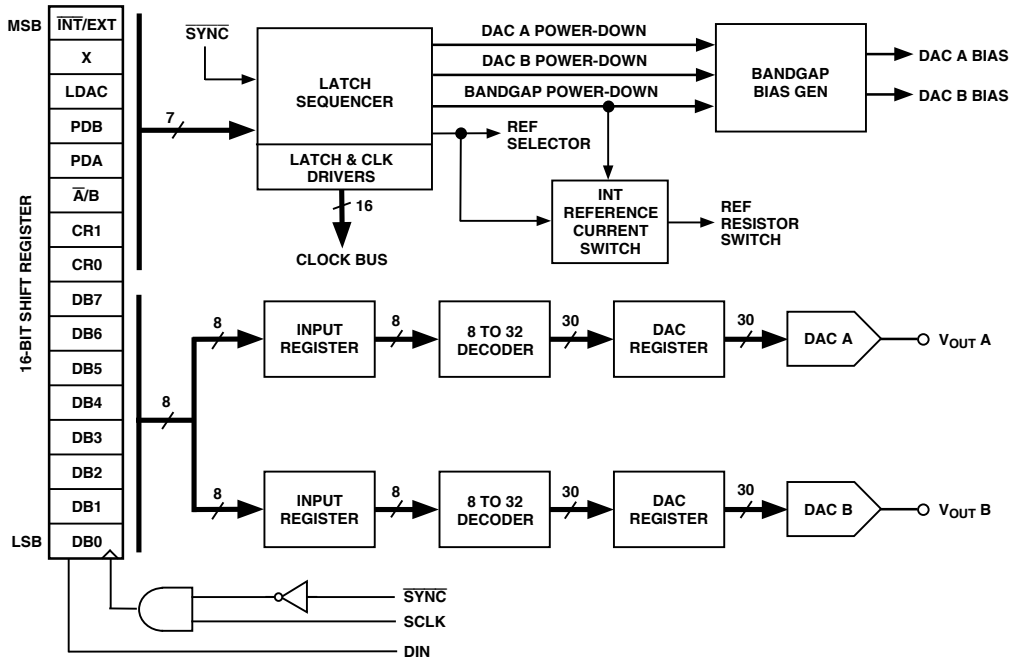
Betriebsarten:

(1) Command = 0b00000000: Parallelbetrieb beider DACs

(2) Beide Kanäle mit unterschiedlichen Signalen nutzen:

Command_1 = 0b00000001: Register A aus Schieberegister laden

Command_2 = 0b00100100: Register B aus Schieberegister laden, beide Ausgänge aktualisieren



Quelle: Analog Devices