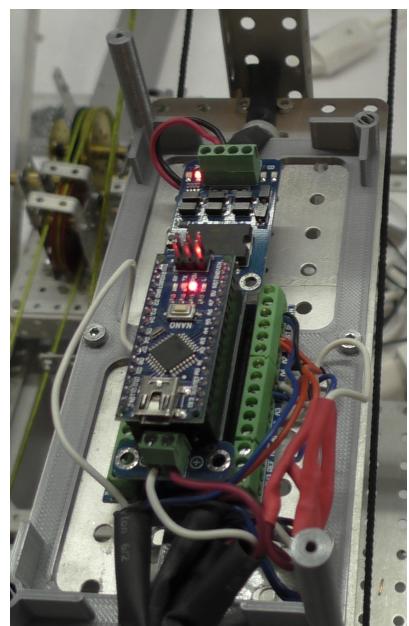
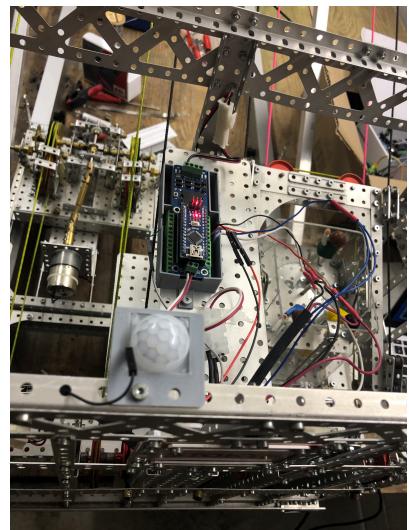
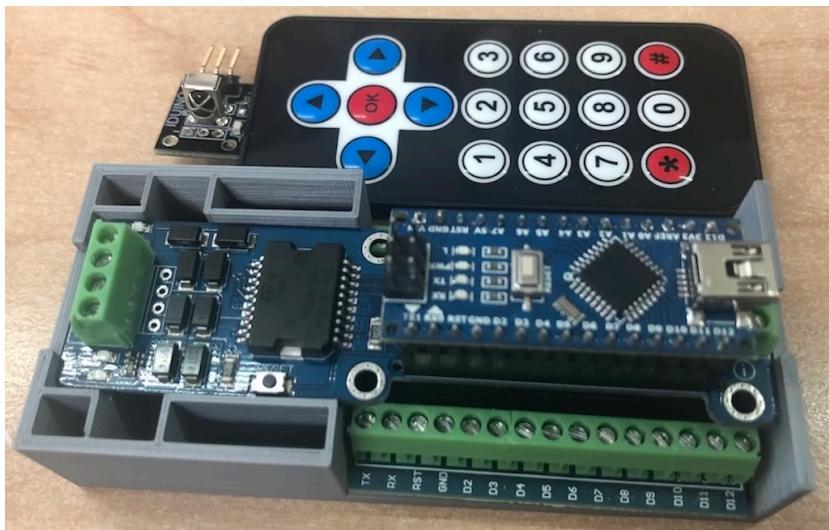


Programmierbare

Nano

Motorsteuerung

für Stokys - Modelle



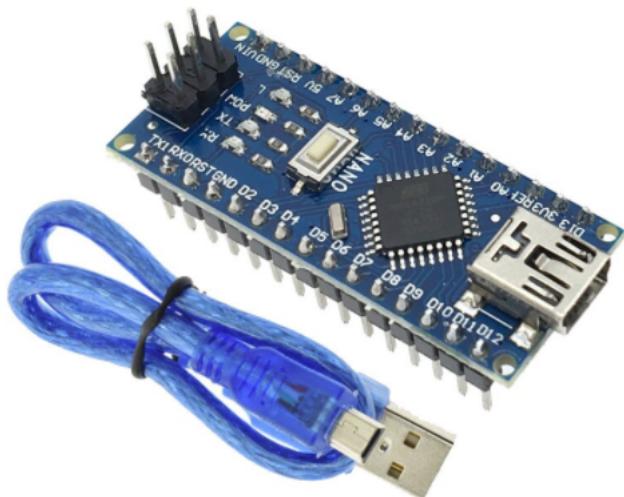
Inhaltsverzeichnis

Komponenten	3
Arduino Nano kompatibler Microcontroller	3
Erweiterungsboard	3
Die Infrarot - Fernbedienung	3
Motorcontroller	4
Montageadapter	5
Anschluss	6
Motoren und Stromversorgung	6
Fernbedienung	7
Funktionsweise	7
Einfaches Code - Beispiel	7
Eine Warnung!	8
Programmierung	9
Installation	9
Beispiel 1 (Richtung und Geschwindigkeit)	10
Beispiel 2 (Beschleunigen und Verzögern)	12
Beispiel 3 (Endschalter)	13
Beispiel 4 (langsam abfahren und ankommen)	15
Beispiel 5 (Steuerung mit Infrarot)	16
Weitere Funktionen	17
Die Infrarot - Fernbedienung	18
Vorbereitung	18
Remote0: Direkte Abfrage	18
Remote1: Wir lassen uns benachrichtigen	18
Remote2: Wurde die Taste losgelassen?	19

Komponenten

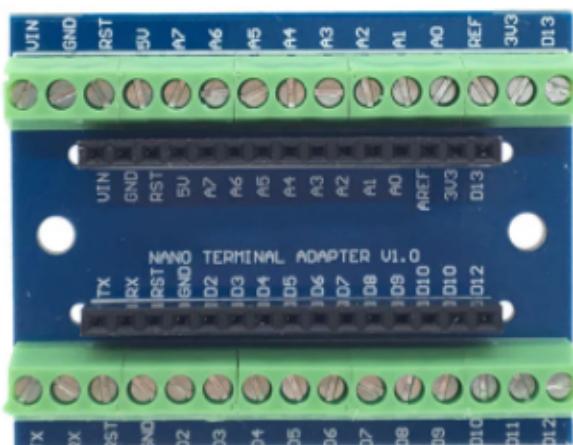
Die Nano Motorsteuerung erlaubt die Ansteuerung von 2 Gleichstrommotoren durch einen Microcontroller. Der Microcontroller ist im Set enthalten und ist kompatibel zu einem Arduino Nano. Mit einer Infrarotfernbedienung lassen sich einfache Kommandos übermitteln.

Arduino Nano kompatibler Microcontroller



Prozessor: Microchip (Atmel) ATmega 328P
Betriebsspannung: 5V
Spannung an den Pins: 5V
USB-Interface: Mini USB (CH340)

Erweiterungsboard



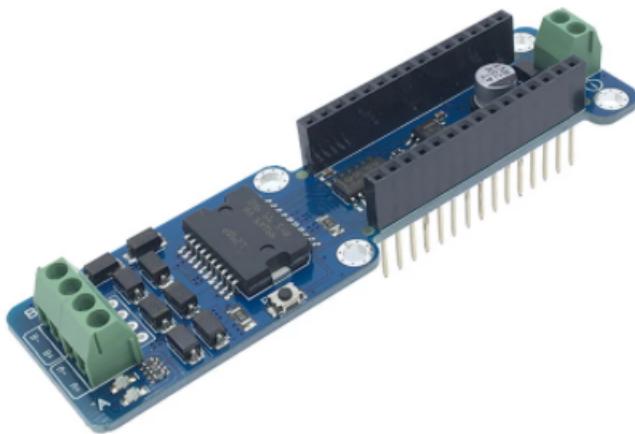
Die Pins des Microcontrollers werden durch Schraubanschlüsse zugänglich gemacht. Es handelt sich um eine Direktverbindung, es sind also keine Schutzmassnahmen dazwischengeschaltet.

Die Infrarot - Fernbedienung



Die Infrarot - Fernbedienung erlaubt eine einfache Steuerung des Modells.

Motorcontroller



L298 Dual Full H-Bridge Chip. Es können zwei DC - Motoren in Richtung und Geschwindigkeit gesteuert werden.

Motor A

Anschluss	Funktion
Microcontroller D12	Richtung
Microcontroller D3	Geschwindigkeit (PWM 0 .. 255)
Microcontroller A0	Stromsensor (in der Software nicht verwendet)
Microcontroller D9	Break (in der Software nicht verwendet)
Motoranschluss A+, A- (grüne Anschlüsse)	Verbindung zum Motor (max. 2A)

Motor B

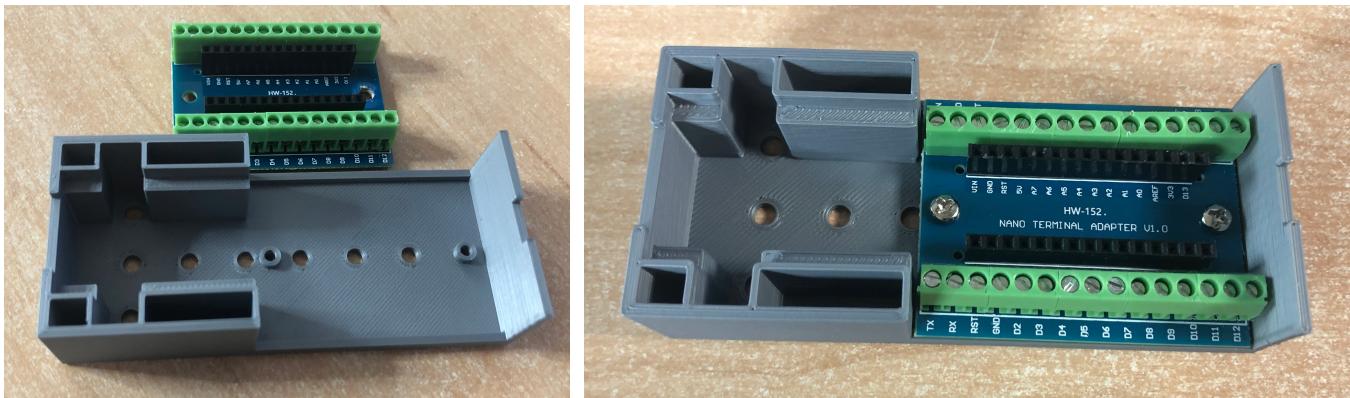
Anschluss	Funktion
Microcontroller D13	Richtung
Microcontroller D11	Geschwindigkeit (PWM 0 .. 255)
Microcontroller A1	Stromsensor (in der Software nicht verwendet)
Microcontroller D8	Break (in der Software nicht verwendet)
Motoranschluss B+, B- (grüne Anschlüsse)	Verbindung zum Motor (max. 2A)

Stromversorgung

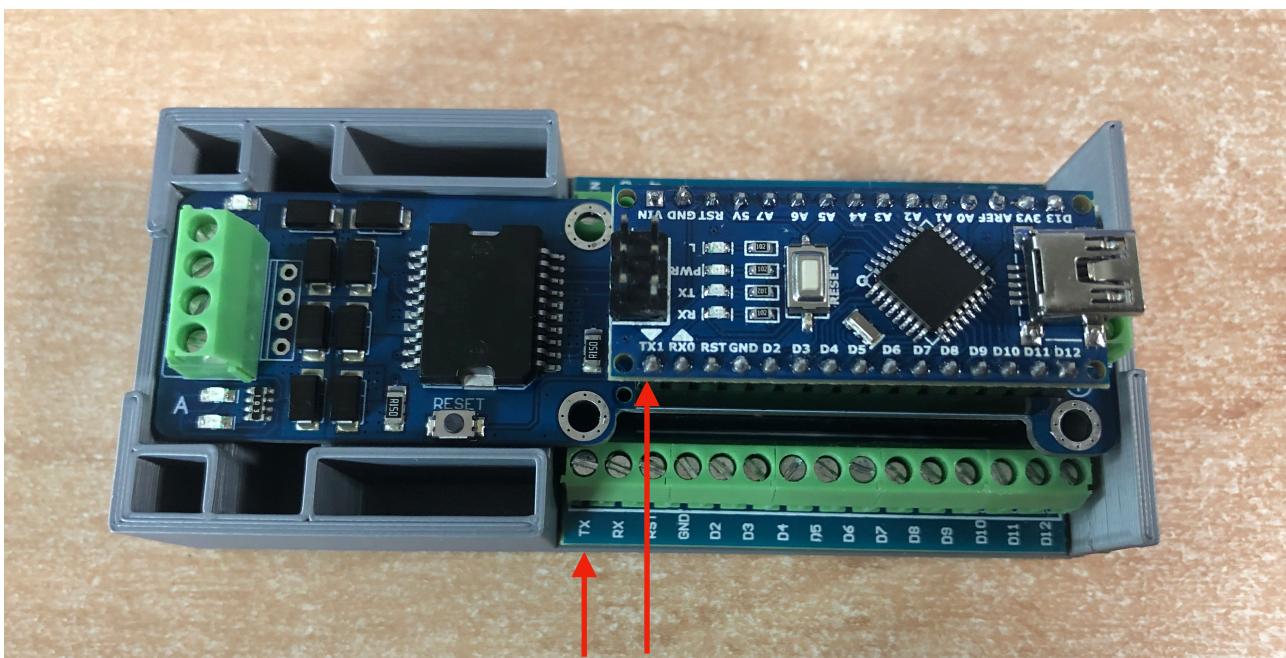
Anschluss	Funktion
Betriebsspannung +, - (grüne Anschlüsse)	7 - 12 V DC

Montageadapter

Der 3D - gedruckte Adapter erleichtert die Integration in ein Stokys - Modell. Die stl-Datei für den Druck ist im Github - Repository zu finden (<https://github.com/hobbyelektroniker/NanoMotorsteuerung>).



Die Löcher am Boden halten die Stokys Lochabstände ein. Hier können normale Stokys - Schrauben verwendet werden. Zur Befestigung des Erweiterungsboards sind zwei M3 Schrauben im Set enthalten. Diese werden in ein 3D gedrucktes Gewinde geschraubt. Die Schrauben dürfen nicht zu fest angezogen werden, damit das Gewinde nicht bricht.



Beschriftung muss übereinstimmen!

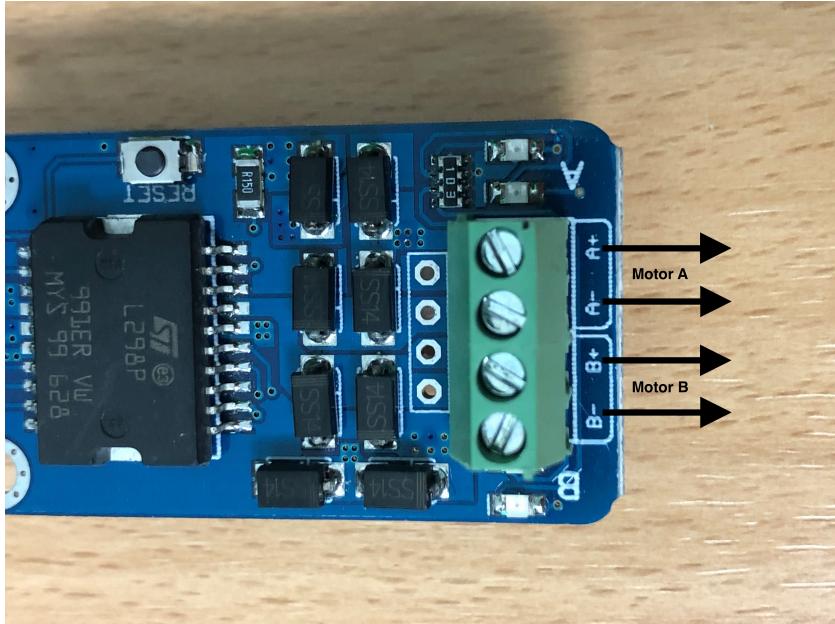
ACHTUNG: Das Erweiterungsboard muss korrekt eingebaut werden. Bei einer falschen Ausrichtung wird die Elektronik bei der Inbetriebnahme zerstört. Also unbedingt die Beschriftung vergleichen!

Der Rest der Elektronik kann dann einfach aufgesteckt werden.

Anschluss

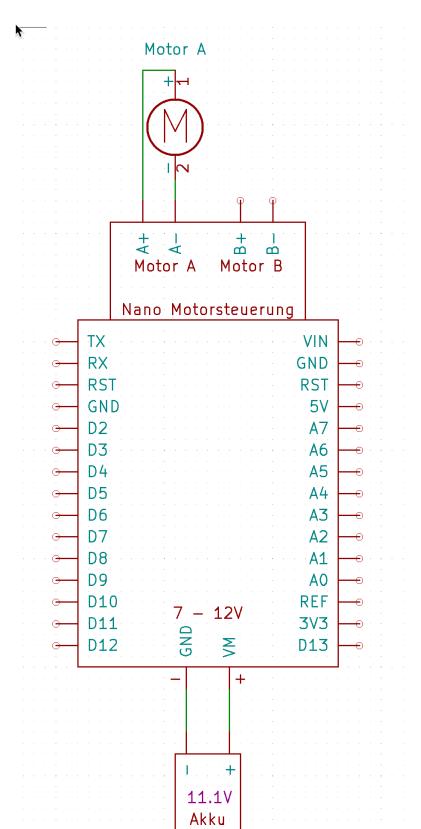
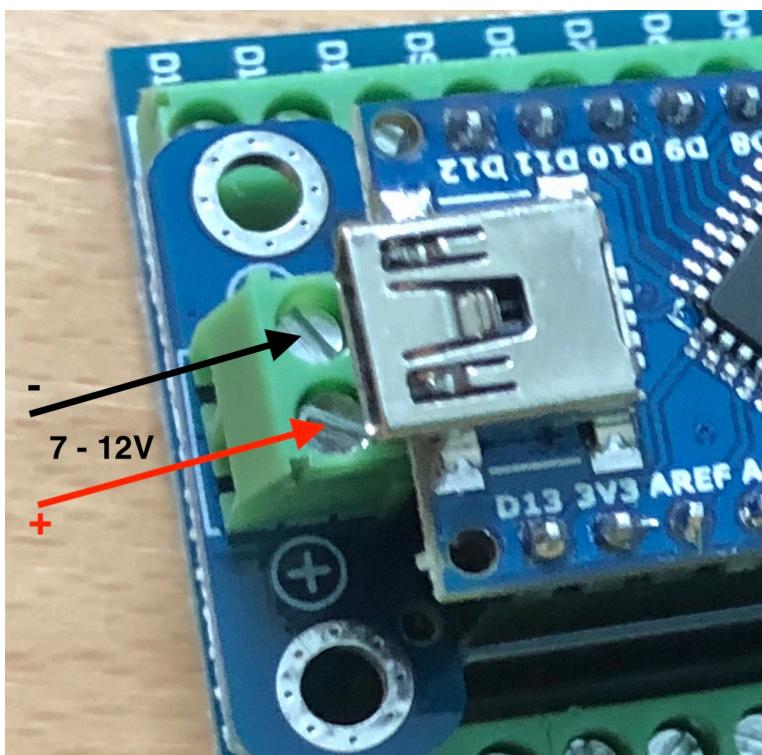
Motoren und Stromversorgung

Die Motoren können mit dem vierpoligen grünen Anschluss verbunden werden.

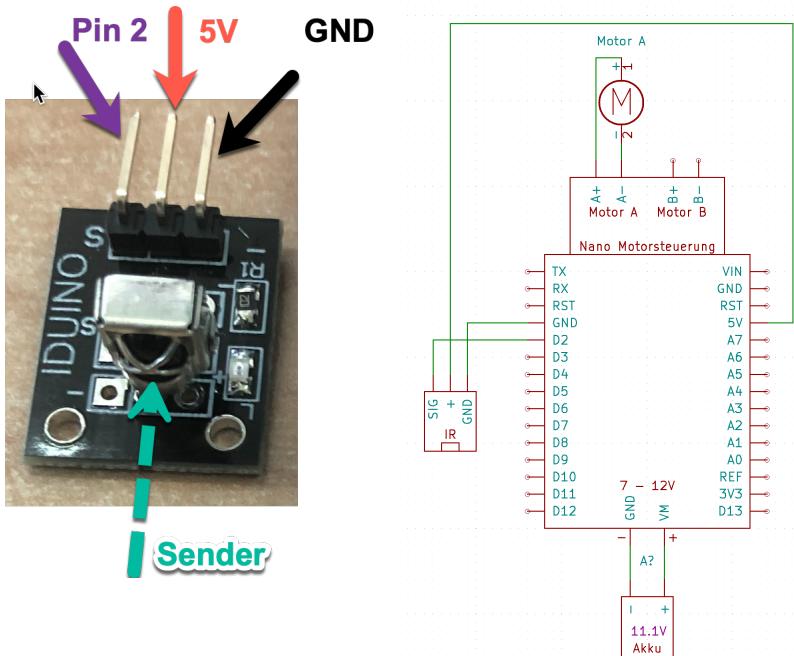


Als Spannungsquelle eignet sich ein Netzteil mit einer Spannung von 7 - 12V. Die Spannung sollte dem Motor entsprechen, da die volle Spannung an den Motor weitergegeben wird.

Der Microcontroller wird über seinen eigenen Spannungsregler mit 5V versorgt. Dieser ist auf max. 12V ausgelegt, darum sollten hier die 12V nicht überschritten werden.



Fernbedienung



Das Empfängermodul wird an Pin D2 angeschlossen. Dabei muss beachtet werden, dass der Sender und die Empfangsdiode Sichtverbindung haben.

Funktionsweise

Einfaches Code - Beispiel

Die Steuerung eines Motors erfolgt über drei Pins am Microcontroller. Die notwendigen Verbindungen sind innerhalb der Nano Motorsteuerung bereits vorhanden. Es muss also nur noch für die passende Programmierung gesorgt werden.

Für die folgenden Erklärungen wird Motor A verwendet. Sie dienen nur dem Verständnis. Für den praktischen Betrieb empfehlen wir vom Beispielprogramm auszugehen und die mitgelieferte Bibliothek zu verwenden.

Grundlegende Arduino - Kenntnisse werden vorausgesetzt. Falls keine Arduino - Kenntnisse vorhanden sind, findet man im Fachhandel genügend einführende Literatur. Eine Einführung geben auch diverse Youtube - Videos, wie zum Beispiel mein Arduino - Kurs:

Arduino Einführungskurs

Für Motor A werden die digitalen Pins D12 und D3 verwendet. Mit D12 wird die Richtung bestimmt. D3 ist ein PWM - Ausgang, der mit Werten zwischen 0 und 255 angesteuert wird. Um anzuhalten, wird der Wert an D3 auf 0 gesetzt. Der Motor erhält dadurch keinen Strom mehr und bleibt stehen. Er wird aber nicht aktiv abgebremst.

Im Programm kann das so realisiert werden:

```
const int dirAPin = 12;
const int speedAPin = 3;

void setup() {
  pinMode(dirAPin, OUTPUT);

  // Ausgangszustand
  analogWrite(speedAPin, 0);    // Anhalten
```

```
delay(2000);

// Schnell vorwärts fahren
// Die tatsächliche Richtung hängt vom Anschluss des Motors ab
// Falls diese nicht den Erwartungen entspricht,
// einfach Polarität tauschen.
digitalWrite(dirAPin,LOW);      // Richtung
analogWrite(speedAPin,255);    // Volle Geschwindigkeit

delay(2000);

// Schnell rückwärts fahren
digitalWrite(dirAPin,HIGH);    // Richtung
analogWrite(speedAPin,127);    // Halbe Geschwindigkeit

}

void loop() {}
```

Die Anschlüsse D9 und A0 werden hier nicht verwendet, sind aber belegt und stehen nicht zur freien Verfügung! Ebenso sind Pins für Motor B belegt (D13, D11, D8 und A1).

Eine Warnung!

Es sollten keine plötzlichen Richtungswechsel durchgeführt werden. Es sollte immer zuerst angehalten werden, bevor in der anderen Richtung wieder losgefahren wird. Da der Motor beim Anhalten noch etwas nachläuft, sollte genügend Zeit vorgesehen werden.

Im Ruhezustand sollte der Motor nicht manuell betätigt werden, ohne vorher die Verbindung zu trennen. Die vom Motor erzeugte Spannung wird an den Motorcontroller weitergegeben und kann dort Schaden anrichten!

Programmierung

Installation

Zur Programmierung wird die Arduino IDE verwendet.

<https://www.arduino.cc/en/software>

Wenn der USB - Port des Nanos nicht erkannt wird, muss ein USB - Treiber für den CH340/CH341 - Chip installiert werden.

Der Treiber sollte nur installiert werden, wenn der USB - Port des Nanos nicht erkannt wird!

Windows: http://www.wch-ic.com/downloads/CH341SER_EXE.html

Mac: http://www.wch-ic.com/downloads/CH341SER_MAC_ZIP.html

Linux: http://www.wch-ic.com/downloads/CH341SER_LINUX_ZIP.html

Die Download - Seite mit allen Treibern des Herstellers:

<http://www.wch-ic.com/downloads/category/30.html>

In der Arduino IDE muss das Board **Arduino Nano** eingestellt werden. Als Prozessor sollte **ATmega328P** oder **ATmega328P (Old Bootloader)** gewählt werden. Es sind momentan beide Versionen im Umlauf, also ausprobieren!

Von der Github Seite

<https://github.com/hobbyelektroniker/NanoMotorsteuerung>

kann das ganze Zip - File heruntergeladen werden. Darin sind alle besprochenen Beispiele enthalten.

Beispiel 1 (Richtung und Geschwindigkeit)

In diesem Beispiel wird gezeigt, wie man den Motor vorwärts oder rückwärts laufen lässt oder ihn anhält. Außerdem kann man lernen, wie die Geschwindigkeit eingestellt wird.

Das Hauptprogramm befindet sich in der Datei Beispiel1.ino. Zusätzlich wird die Datei motor.h benötigt. Beide Dateien befinden sich in einem Verzeichnis Beispiel1.

motor.h stellt eine Klasse für die Ansteuerung des Motors zur Verfügung. Sie muss im Normalfall nicht angepasst werden.

Das eigentliche Programm steht in der .ino - Datei.

Die Datei motor.h wird wie eine Bibliothek mit #include in das Programm eingebunden.

```
#include "motor.h"
```

Danach wird eine Variable angelegt, über die man auf den Motor zugreifen kann. Dabei wird angegeben, ob es sich um Motor A oder Motor B handelt.

```
DCMotor motor('A'); // an Motoranschluss A
```

Wenn beide Motoren angesprochen werden sollten, müssten zwei Variablen erstellt werden:

```
DCMotor motorA('A');
DCMotor motorB('B');
```

Wir bleiben aber in diesem Beispiel bei einem Motor.

In **setup()** wird dieser Motor bereit gemacht.

```
void setup() {
    motor.begin(); // Motor vorbereiten
}
```

In **loop()** rufen wir eine Demo auf. Wenn diese beendet ist, wird 10 Sekunden gewartet und dann wird sie neu gestartet.

```
void loop() {
    demo();           // Demo ablaufen lassen
    delay(10000);   // 10 Sekunden warten
}
```

Der Hauptteil des Programms läuft in der Funktion **demo()** ab.

Zuerst fahren wir mit mittlerer Geschwindigkeit vorwärts. Dazu werden zwei Motorbefehle benutzt: setForward() und setSpeed().

```
motor.setForward();
motor.setSpeed(127); // 0 .. 255 möglich
```

Mit **setForward()** wird die Richtung bestimmt. Mit diesem Aufruf sollte das Modell vorwärts fahren. Die tatsächliche Richtung hängt dabei vom mechanischen Aufbau des Modells ab. Falls die Richtung nicht stimmt, kann man einfach die beiden Anschlussleitungen zwischen Motorcontroller und Motor tauschen.

Für die Rückwärtsfahrt wird derselbe Befehl benutzt. Man sagt dann einfach, dass man nicht vorwärts fahren möchte (**setForward(false)**).

Mit **setSpeed()** wird die Geschwindigkeit festgelegt. Dabei können Werte von **0** bis **255** übergeben werden. Bei 0 bleibt der Motor stehen, bei 255 hat er die maximale Geschwindigkeit. Leider liegt es in der Natur der

PWM - Ansteuerung, dass der Motor bei niedrigen Geschwindigkeiten deutlich an Kraft verliert. So wird er je nach Belastung schon bei wesentlich höheren Werten als 0 nicht mehr drehen.

Zum Anhalten wird der Befehl **stop()** verwendet. Dieser Befehl setzt die Geschwindigkeit auf 0. Er hat also keine Bremswirkung, er entzieht dem Motor einfach die Stromversorgung.

Beispiel 2 (Beschleunigen und Verzögern)

Dieses Beispiel zeigt, wie man langsam anfahren und wieder anhalten kann.

Das Anfahren wird über den Befehl `speedUp()` gestartet. Dabei wird die gewünschte Endgeschwindigkeit angegeben.

```
// auf volle Geschwindigkeit (255) beschleunigen  
motor.speedUp(255);
```

Mit `slowDown()` kann die Geschwindigkeit wieder verringert werden.

```
// auf halbe Geschwindigkeit reduzieren  
motor.slowDown(127);
```

Beide Befehle sind blockierend. Die Funktion wird also erst wieder verlassen, wenn die gewünschte Geschwindigkeit erreicht ist. Die Geschwindigkeit wird in festgelegten Schritten durchgeführt, zwischen den Schritten wird eine bestimmte Pause eingelegt.

Diese Werte können angepasst werden. Dadurch kann festgelegt werden, wie stark die Beschleunigung oder Verzögerung ist. Standardmäßig wird die Geschwindigkeit jeweils um 5 geändert und zwischen den Schritten eine Pause von 100 ms eingelegt.

Mit `setAcceleration()` kann das geändert werden.

```
// Ein Schritt verändert die Geschwindigkeit um 2, die Pause wird nicht geändert  
motor.setAcceleration(2);
```

```
// Ein Schritt verändert die Geschwindigkeit um 10, die Pause wird auf 300 ms festgelegt  
motor.setAcceleration(10, 300);
```

Beispiel 3 (Endschalter)

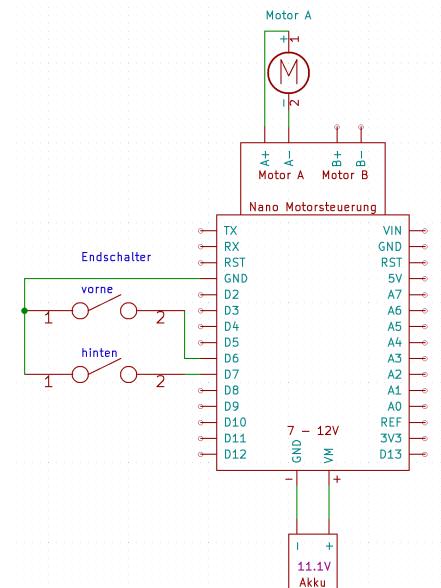
Wir fahren zwischen zwei Stationen hin und her. An jeder Station wird durch ein Endschalter angezeigt, dass wir angekommen sind. Die Endschalter gehen von einem Arduino - Pin aus und verbinden bei Betätigung mit GND.

Zuerst werden die beiden Pins als Konstanten festgelegt.

```
// Pins für Endschalter definieren
// Die Endschalter verbinden bei
// Betätigung den Pin mit GND
const int vornePin = 6;
const int hintenPin = 7;
```

In **setup()** müssen die Pins vorbereitet werden.

```
// Endschalter vorbereiten
pinMode(vornePin, INPUT_PULLUP);
pinMode(hintenPin, INPUT_PULLUP);
```



Es handelt sich um INPUT - Pins, mit **INPUT_PULLUP** schalten wir einen internen PULLUP - Widerstand hinzu. Wenn der Taster nicht gedrückt wird, erhält der Pin 5V über den PULLUP-Widerstand. Er ist dadurch auf HIGH - Level. Wenn der Taster gedrückt ist, wird der Pin mit GND verbunden und ist dann auf LOW. Dadurch können wir den Taster wie folgt abfragen:

```
if (!digitalRead(vornePin)) ...; // Ist der Taster gedrückt?
if (digitalRead(vornePin)) ...; // Ist der Taster NICHT gedrückt?
```

Die Logik ist also genau umgekehrt, deshalb muss genau auf das ! geachtet werden.

Das benutzen wir gleich in **setup()** um festzulegen, in welche Richtung wir abfahren müssen.

```
// Wir testen, ob wir schon an einem Ende stehen
if (!digitalRead(vornePin)) {
    // wir stehen hinten, müssen also nach vorne losfahren
    motor.setForward(false);
} else if (!digitalRead(hintenPin)) {
    motor.setForward();
} else {
    // wir stehen irgendwo in der Mitte und fahren nach vorne los
    motor.setForward();
}
motor.setSpeed(255); // Wir starten mit voller Geschwindigkeit
```

Wenn wir bereits an einer Station stehen, verlassen wir diese. Falls wir irgendwo dazwischen sind, fahren wir vorwärts.

Während der Fahrt testen wir laufend, ob wir schon auf einer Seite angekommen sind. Dieser Code muss in **loop()** stehen. Falls wir vorwärts fahren und den Endschalter vorne betätigt haben, sind wir vorne angekommen.

```
if (motor.getForward() && !digitalRead(vornePin)) { ..
```

oder wir fahren rückwärts und der hintere Endschalter wird betätigt:

```
if (!motor.getForward() && !digitalRead(hintenPin)) { ..
```

Jetzt sind wir an einer Station angekommen, halten an, warten eine Weile und fahren dann in umgekehrter Richtung wieder los.

Zwischen zwei Abfragen machen wir mindestens 50 ms Pause. Beim Nano wäre das nicht unbedingt notwendig, es gibt aber Prozessoren, die benötigen diese Pause.

```
void loop() {
    if (motor.getForward() && !digitalRead(vornePin)) {
        // Wenn wir nach vorne fahren und vorne angekommen sind
        // halten wir an und fahren nach 3 Sekunden Pause nach hinten los
        motor.stop(); // anhalten
        delay(3000); // 3 Sekunden Pause
        motor.setForward(false); // Richtung nach hinten
        motor.setSpeed(255); // los geht's
    } else if (!motor.getForward() && !digitalRead(hintenPin)) {
        // Wenn wir nach hinten fahren und hinten angekommen sind
        // halten wir an und fahren nach 3 Sekunden Pause nach vorne los
        motor.stop(); // anhalten
        delay(3000); // 3 Sekunden Pause
        motor.setForward(); // Richtung nach vorne
        motor.setSpeed(255); // los geht's
    }
    delay(50); // Eine kleine Pause zwischen den Abfragen schadet nichts
}
```

Beispiel 4 (langsam abfahren und ankommen)

In diesem Beispiel fahren wir langsam los und beschleunigen auf Maximalgeschwindigkeit. Die Fahrzeit dauert 8 Sekunden, dann beginnen wir langsamer zu fahren. Wir fahren dann mit einer Minimalgeschwindigkeit ein und halten an, wenn der Endschalter erreicht wird. Nach 3 Sekunden Halt beschleunigen wir wieder in die andere Richtung.

Bei der Abfahrt merken wir uns die Startzeit in einer eigenen Variablen **startZeit**. Die Beschleunigung erfolgt über **speedUp()**. Dazu erstellen wir eine eigene Funktion.

```
// Startzeit
unsigned long startZeit;

void abfahrt(bool nachVorne) {
    startZeit = millis();
    motor.setForward(nachVorne);
    motor.speedUp(255); // Auf volle Geschwindigkeit beschleunigen
}
```

Nach einer bestimmten Fahrzeit werden wir langsamer und halten bei Erreichen des Endschalters an. Nach 3 Sekunden fahren wir wieder in der entgegengesetzten Richtung los. Dazu erstellen wir die Funktion **ankunftUndAbfahrt()**.

```
void ankunftUndAbfahrt() {
    if (motor.getForward()) {
        // Wir sind fahren vorwärts
        motor.slowDown(50, testVorne);
        while (!testVorne()) delay(5);
        motor.stop(); // anhalten
        delay(3000); // 3 Sekunden Pause
        abfahrt(false);
    } else {
        motor.slowDown(50, testHinten);
        while (!testHinten()) delay(5);
        motor.stop(); // anhalten
        delay(3000); // 3 Sekunden Pause
        abfahrt(true);
    }
}
```

Das Abbremsen auf die Minimalgeschwindigkeit dauert eine Weile. Wir sind dabei in der Funktion **slowDown()**. Auch während dem Abbremsen, sollten wir auf den Endschalter testen. Aus diesem Grund geben wir der Funktion **slowDown()** eine Testfunktion für den Endschalter mit. Dazu müssen wir Testfunktionen für die beiden Endschalter schreiben.

```
bool testVorne() {
    return (motor.getForward() && !digitalRead(vornePin));
}

bool testHinten() {
    return (!motor.getForward() && !digitalRead(hintenPin));
}
```

slowDown() wird dann sofort abbrechen, wenn der Endschalter erreicht ist. Andernfalls läuft die Funktion bis die Endgeschwindigkeit erreicht ist. Wir müssen also vor dem Stopp testen, ob der Endschalter wirklich erreicht ist. Wenn nicht, wird einfach gewartet.

Durch die Funktion **ankunftUndAbfahrt()** vereinfacht sich **loop()** sehr.

```
void loop() {
    if (millis() - startZeit > 8000) ankunftUndAbfahrt();
    delay(50); // Eine kleine Pause zwischen den Abfragen schadet nichts
}
```

Beispiel 5 (Steuerung mit Infrarot)

Diesmal verzichten wir auf die Endschalter und arbeiten mit der Infrarot - Fernbedienung.

Zuerst müssen wir zusätzlich zur Bibliothek ***motor.h*** noch ***Fernbedienung.h*** einbinden. Der Empfänger steht dann automatisch zur Verfügung, muss aber in ***setup()*** initialisiert werden. Außerdem benötigen wir noch die Dateien ***SimpleIRReceiver.h*** und ***digitalWriteFast.h***.

```
void setup() {
    Serial.begin(115200);
    // Motor vorbereiten
    motor.begin();
    zustand = stehend;
    // Fernbedienung vorbereiten
    empfaenger.begin(2);
    empfaenger.setNoRepeat();
    empfaenger.setCallback(newCmd);
}
```

Für den Empfänger erstellen wir eine Callback - Funktion (***newCmd***), die alle empfangenen Kommandos abarbeitet. Diese wird dem Empfänger in ***setup()*** zugewiesen. Die Funktion ***newCmd()*** müssen wir nie selbst aufrufen. Der Empfänger wird das selbstständig tun, wenn eine Taste auf der Fernbedienung gedrückt wird.

Damit wir auch ein Signal erhalten, wenn eine Taste losgelassen wird, muss in ***loop()*** regelmässig ***refresh()*** aufgerufen werden.

```
void loop() {
    empfaenger.refresh();
    delay(10); // Eine kleine Pause zwischen den Abfragen schadet nichts
}
```

Jedes Kommando von der Fernbedienung ruft automatisch ***newCmd()*** auf . Das empfangene Kommando wird mit switch case ausgewertet. Die vordefinierten Konstanten (***IRCMD_***) mit den Codes findet man in der Datei ***Fernbedienung.h***.

Dieses einfache Beispiel arbeitet im ***Mode 2*** der Empfänger - Klasse. Dabei wird jeder Tastendruck sofort signalisiert. Sobald während mindestens 200 ms keine Taste mehr gedrückt wird, erhalten wir eine ***IRCMD_NONE*** - Meldung. Wir haben in ***setup()*** ***empfaenger.setNoRepeat()*** aufgerufen, daher wird jedes Kommando nur einmal signalisiert.

Bedienung

Mit **VOR** und **ZURÜCK** (***IRCMD_UP***, ***IRCMD_DOWN***) auf der Fernbedienung steuern wir den Motor direkt. Er fährt sofort mit maximaler Geschwindigkeit in die angegebene Richtung. Sobald wir die Taste loslassen, hält der Motor sofort an. Wir erhalten dazu ein Kommando ***IRCMD_NONE*** von der Fernbedienung.

Mit **LINKS** und **RECHTS** (***IRCMD_LEFT***, ***IRCMD_RIGHT***) auf der Fernbedienung steuern wir den Motor indirekt. Er fährt langsam los und beschleunigt in die angegebene Richtung. Der Motor wird erst durch Druck auf **OK** (***IRCMD_OK***) verlangsamt. Er verringert dann die Geschwindigkeit, bis er steht.

Falls ein sofortiger Stopp notwendig ist, können wir mit der **Taste 0** (***IRCMD_0***) einen Notstopp auslösen.

Weitere Funktionen

Es ist möglich, eine Stromüberwachung einzurichten. Die Pins A0 und A1 erhalten eine Spannung, die proportional zum Motorstrom ist. Diese Werte können mit ***analogRead()*** abgefragt werden. In der Motorklasse steht dazu die Funktion ***motor.getMonitor()*** zur Verfügung.

Es wird hier bewusst auf ein Beispiel verzichtet. Diese Werte sind nicht sehr genau und man sollte wissen, was man tut, bevor man diese Möglichkeit verwendet.

Die Infrarot - Fernbedienung

Vorbereitung

Damit die Fernbedienung angesprochen werden kann, müssen die Dateien **Fernbedienung.h**, **SimpleIRReceiver.h** und **digitalWriteFast.h** in das Projektverzeichnis geladen werden. Durch das Einbinden von Fernbedienung.h wird uns ein Objekt mit dem Namen **empfaenger** zur Verfügung gestellt.

Das Empfängermodul wird, wie oben beschrieben, an **Pin2**, 5V und GND angeschlossen.

In **setup()** wird der Empfänger mit **empfaenger.begin()** vorbereitet. Dabei kann ein Mode (0 bis 2) gesetzt werden. Diese drei Modi werden im Folgenden anhand von drei Beispielprogrammen besprochen.

Remote0: Direkte Abfrage

Im **Mode 0** fragen wir die gedrückte Taste selbst ab.

In **setup()** rufen wir **empfaenger.begin()** oder **empfaenger.begin(0)** auf. Dadurch wird der Empfänger in den Mode 0 versetzt.

Die aktuell gedrückte Taste fragen wir mit **empfaenger.getCmd()** ab. Wenn keine Taste gedrückt ist, erhalten wir **IRCMD_NONE**, was einem Wert von 0 entspricht. Andernfalls erhalten wir einen Wert, der einem der anderen IRCMD_- Werte entspricht.

Wenn wir die Taste gedrückt halten, werden wir mehrere Codes der gedrückten Taste erhalten. Diese Autorepeat - Funktion kann durch Aufruf von **empfaenger.setNoRepeat()** in **setup()** unterdrückt werden. Das kann mit **empfaenger.setNoRepeat(false)** wieder rückgängig gemacht werden.

Remote1: Wir lassen uns benachrichtigen

Im **Mode 1** lassen wir uns über eine Callback - Funktion direkt benachrichtigen. **getCmd()** liefert hier keine Resultate mehr zurück.

In **setup()** rufen wir **empfaenger.begin(1)** auf. Dadurch wird der Empfänger in den Mode 1 versetzt. Zusätzlich müssen wir eine durch uns geschriebene Callback - Funktion angeben (**empfaenger.setCallback(show)**). Diese Funktion muss einen festen Aufbau haben.

Wir schauen uns die Funktion aus dem Beispiel an:

```
void show(int cmd) {  
    Serial.println("Callback: " + cmd);  
}
```

Es muss sich immer um eine **void** - Funktion handeln. Sie hat einen **Parameter** vom Typ **int**. Damit wären auch schon alle Bedingungen erfüllt. Die Namen sind frei wählbar. Bei jedem Tastendruck wird diese Funktion automatisch aufgerufen und wir erfahren durch den übergebenen Parameter, welche Taste gedrückt wurde.

Auch hier können wir mit **empfaenger.setNoRepeat()** die Autorepeat - Funktion unterdrücken..

In diesem Mode werden wir nie erfahren, dass eine Taste losgelassen wurde.

Remote2: Wurde die Taste losgelassen?

Im **Mode 2** lassen wir uns ebenfalls über eine Callback - Funktion direkt benachrichtigen. `getCmd()` liefert hier keine Resultate mehr zurück. Zusätzlich erhalten wir ein **IRCMD_NONE** - Signal über die Callback - Funktion, sobald eine Taste losgelassen wird. Das Signal wird erzeugt, sobald für 200 ms keine Taste mehr gedrückt wird.

In `setup()` rufen wir **`empfaenger.begin(2)`** auf. Dadurch wird der Empfänger in den Mode 2 versetzt.

Zusätzlich müssen wir eine durch uns geschriebene Callback - Funktion angeben
(**`empfaenger.setCallback(show)`**). Diese Funktion muss wie in Mode 1 aufgebaut sein.

Bei jedem Tastendruck wird diese Funktion automatisch aufgerufen und wir erfahren durch den übergebenen Parameter, welche Taste gedrückt wurde. Wenn wir die Taste loslassen und für mindestens 200 ms keine Taste mehr drücken, wird ein **IRCMD_NONE** - Signal ausgelöst.

Damit das funktioniert, müssen wir regelmässig (in Abständen von weniger als 200 ms) **`empfaenger.refresh()`** aufrufen.

Auch hier können wir mit **`empfaenger.setNoRepeat()`** die Autorepeat - Funktion unterdrücken..